Advanced Joins on GPUs

by

Christos Bellas

This study was supported by the Hellenic Foundation for Research and Innovation (HFRI) under the HFRI PhD Fellowship grant (Fellowship Number: 1154).

Σύνοψη

Τα τελευταία χρόνια υπάρχει μία μεγάλη αύξηση στη χρήση των μονάδων επεξεργαστών γραφικών (ΜΕΓ) για την επίλυση προβλημάτων. Συγκεκριμένα, πέρα από την απόδοση γραφικών, μέσω του ετερογενούς προγραμματιστικού μοντέλου που ονομάζεται Γενικού Σκοπού ΜΕΓ, μία ή περισσότερες ΜΕΓ μπορούν να χρησιμοποιηθούν συνεργατικά με παραδοσιακές κεντρικές μονάδες επεξεργασίας (ΚΜΕ) για την επίλυση προβλημάτων. Το κύριο πλεονέκτημα των ΜΕΓ είναι ο υψηλός βαθμός παραλληλισμού που προσφέρουν έναντι πολύ χαμηλού ενεργειακού κόστους λόγω της αρχιτεκτονικής τους. Πέρα από το χαμηλό ενεργειακό κόστος, η χρήση των ΜΕΓ έχει οδηγήσει και σε πάρα πολύ μεγάλες αυξήσεις των επιδόσεων, ειδικά στον τομέα της μηχανικής μάθησης και της βαθιάς γνώσης. Ο λόγος που συμβαίνει αυτό είναι επειδή οι τομείς αυτοί ενθυλακώνουν ομαλούς αλγορίθμους. Δηλαδή αλγορίθμους στους οποίους ο έλεγχος ροής αλλά και αναφορές σε θέσεις μνήμης δεν εξαρτώνται από δεδομένα εισόδου. Χαρακτηριστικά παραδείγματα ομαλών αλγορίθμων είναι ο πολλαπλασιασμός πινάκων και η συνέλιξη. Σε αυτά τα προβλήματα, το παράλληλο περιβάλλον των ΜΕΓ ευνοείται καθώς διαβάζει μεγάλους όγκους δεδομένων, κάνει τις απαραίτητες πράξεις και γράφει σε προκαθορισμένες θέσεις μνήμης χωρίς να υπάρχουν σύνθετες διακλαδώσεις.

Στον αντίποδα, για μη ομαλούς αλγορίθμους, όπου ο έλεγχος και οι αναφορές σε θέσεις μνήμης καθορίζονται από τα δεδομένα εισόδου και τις δομές δεδομένων, η κατάσταση είναι πιο περίπλοκη. Συγκεκριμένα, λόγω της φύσης τους, αυτοί οι αλγόριθμοι απαιτούν αλληλοεπικοινωνία μεταξύ των εκτελούμενων νημάτων στην ΜΕΓ, ανάγνωση και γράψιμο σε μη προκαθορισμένες θέσεις μνήμης, χαρακτηριστικά που δεν ευνοούν το παράλληλο περιβάλλον της ΜΕΓ. Αυτό έχει ως αποτέλεσμα την πολύ μικρότερη αύξηση των επιδόσεων σε σχέση με τους ομαλούς αλγορίθμους.

Ένα χαρακτηριστικό παράδειγμα μη ομαλών αλγορίθμων είναι οι συνδέσεις, όπου αντικείμενα από δύο ή περισσότερες συλλογές συνδέονται βάσει μίας συνθήκης σύνδεσης. Οι συνδέσεις αποτελούν θεμέλιο λίθο στη διαχείριση δεδομένων με χαρακτηριστική εφαρμογή σε πεδία όπως οι βάσεις δεδομένων, η εξόρυξη δεδομένων και η ανάλυση οντοτήτων. Ο πιο μελετημένος τύπος σύνδεσης σε ΜΕΓ είναι οι συνδέσεις ισότητας για τις οποίες έχουν αναφερθεί σημαντικές αυξήσεις των επιδόσεων έναντι υλοποιήσεων που εκτελούνται σε ΚΜΕ.

Υπάρχουν πιο περίπλοκοι τύποι συνδέσεων για τους οποίους αφενός δεν υπάρχει ενδελεχής μελέτη σε ΜΕΓ, αφετέρου παραλληλοποιούνται πιο δύσκολα. Το κίνητρο της παρούσας διατριβής ήταν η μελέτη τέτοιων τύπων συνδέσεων και στα πλαίσια αυτής τους χαρακτηρίζουμε προηγμένους. Για την ανάπτυξη των τεχνικών χρησιμοποιήθηκε η κλειστού κώδικα προγραμματιστική πλατφόρμα της CUDA η οποία είναι ιδιοκτησία της NVIDIA.

Συγκεκριμένα μελετήθηκαν και αναπτύχθηκαν λύσεις στο παράλληλο περιβάλλον μιας ΜΕΓ που αφορούν τις θ-συνδέσεις όπου η συνθήκη σύνδεσης είναι αυθαίρετη και οι συνδέσεις ομοιότητας σε σύνολα, όπου βάσει μιας απόστασης ομοιότητας και ενός κατωφλιού, δύο σύνολα μπορούν να θεωρηθούν όμοια. Επιπρόσθετα, μελετήθηκαν και δύο τελεστές πάνω σε σύνολα: (α) ο τελεστής εύρεσης τομής μεταξύ δύο συνόλων, και (β) ο τελεστής σύνδεσης περιεκτικότητας μεταξύ δύο συνόλων, ο οποίος επιστρέφει το βαθμό που ένα σύνολο εμπεριέχεται σε ένα άλλο.

Συγκεκριμένα, το πρώτο πρόβλημα που μελετήθηκε αφορούσε τις θ-συνδέσεις που είναι η πιο γενική μορφή σχεσιακών συνδέσεων, καθώς επιτρέπουν τη χρήση τελεστών ανισότητας στη συνθήκη σύνδεσης. Σε μια γενική θ-σύνδεση μεταξύ δύο σχέσεων, όλοι οι πιθανοί συνδυασμοί πλειάδων, δηλαδή το καρτεσιανό γινόμενο τους, μπορεί να χρειαστεί να ελεγχθεί. Το γεγονός αυτό οδηγεί σε τετραγωνική χρονική πολυπλοκότητα. Επιπρόσθετα, χώρος τετραγωνικής πολυπλοκότητας απαιτείται να δεσμευ-

τεί στη χειρότερη περίπτωση όπου η συνθήκη σύνδεσης είναι αληθής για όλους τους συνδυασμούς. Στην παρούσα διατριβή μελετήθηκαν και υλοποιήθηκαν λύσεις για δύο ερωτήματα θ-συνδέσεων ανάμεσα σε δύο σχέσεις: (α) ένα συναθροιστικό, του οποίου το αποτέλεσμα είναι μόνο μία τιμή, και (β) ένα γενικό ερώτημα το οποίο παράγει πολλαπλές πλειάδες εξόδου. Μέσα από μία σειρά βελτιστοποιήσεων που εκμεταλλεύονται την ιεραρχία μνημών της ΜΕΓ και αποδοτικού επιμερισμού του φόρτου εργασίας ανάμεσα στα εκτελούμενα νήματα της ΜΕΓ, οι τεχνικές της διατριβής καταφέρνουν αύξηση της απόδοσης αναφορικά με το χρόνο εκτέλεσης μέχρι και μία τάξη μεγέθους.

Το δεύτερο πρόβλημα που μελετήθηκε αφορούσε τις συνδέσεις ομοιότητας σε σύνολα. Συγκεκριμένα, δοθέντων δύο συλλογών από σύνολα και ενός κατωφλιού, η σύνδεση ομοιότητας συνόλων υπολογίζει για κάθε ζεύγος συνόλων αν είναι όμοια. Δηλαδή αν ο βαθμός ομοιότητας τους βάσει μίας απόστασης είναι μεγαλύτερος ή ίσος από το κατώφλι. Στη βάση τους οι συνδέσεις ομοιότητας συνόλων είναι τετραγωνικής πολυπλοκότητας. Οι υπάρχουσες τεχνικές της βιβλιογραφίας είναι κυρίως ευριστικές και εστιάζουν στο πρώτο τους στάδιο όπου χρησιμοποιούν ένα μηχανισμό φιλτραρίσματος για να μειώσουν τον αριθμό ζευγών που πρέπει τελικά να ελεγχθούν για το αν είναι όμοια. Πιο συγκεκριμένα, για κάθε ζεύγος συνόλων ο βαθμός ομοιότητας μεταφράζεται σε έναν αριθμό που υποδηλώνει πόσα κοινά στοιχεία πρέπει να έχουν τα δύο σύνολα για να θεωρηθούν όμοια. Η πιο διαδεδομένη απόσταση που χρησιμοποιείται στη βιβλιογραφία και χρησιμοποιήθηκε και στην διατριβή είναι η Jaccard. Η συνεισφορά της παρούσας διατριβής για τις συνδέσεις ομοιότητας σε σύνολα είναι τρίπτυχη.

Στο πρώτο στάδιο, αναπτύχθηκε μια συνεργατική τεχνική στην οποία μία ΚΜΕ και μία ΜΕΓ συνεργάζονται για την επίλυση τους προβλήματος της σύνδεσης ομοιότητας σε σύνολα. Συγκεκριμένα, η ΚΜΕ κάνει το φιλτράρισμα και έπειτα περνά στην ΜΕΓ για έλεγχο τα ζεύγη συνόλων που περνάνε όλα τα φίλτρα. Αυτό μπορεί να οδηγήσει σε επικάλυψη της εκτέλεσης μεταξύ της ΚΜΕ και της ΜΕΓ, γεγονός που οδηγεί σε σημαντική αύξηση των επιδόσεων, έως και 2.6 φορές έναντι των καλύτερων τεχνικών που τρέχουν σε ΚΜΕ, όταν τα πλήθη των ζευγών που ελέγχονται είναι στην τάξη των δισεκατομμυρίων. Επιπρόσθετα, ο χρόνος εκτέλεσης της ΜΕΓ κρύβεται τελείως από το χρόνο εκτέλεσης της ΚΜΕ. Οι επιδόσεις της συνεργατικής τεχνικής που προτείνεται στη διατριβή φράζονται από το χρόνο εκτέλεσης του φιλτραρίσματος στην ΚΜΕ.

Στο δεύτερο στάδιο, πραγματοποιήθηκε μία εκτενής συγκριτική αξιολόγηση των καλύτερων τεχνικών της βιβλιογραφίας που αντιμετωπίζουν το πρόβλημα της σύνδεσης ομοιότητας σε σύνολα είτε εξ' ολοκλήρου σε μία ΚΜΕ ή ΜΕΓ, είτε με τη συνεργατική τεχνική που προτείνεται στην παρούσα διατριβή. Για την αξιολόγηση των τεχνικών χρησιμοποιήθηκαν δεδομένα πραγματικού κόσμου αλλά και τεχνητά. Το κύριο συμπέρασμα της συγκριτικής μας μελέτης είναι ότι δεν υπάρχει ξεκάθαρα καλύτερη λύση. Βάσει των χαρακτηριστικών που διέπουν τα δεδομένα εισόδου, όπως το μέγεθος της συλλογής των συνόλων, το μέσο μέγεθος των συνόλων της συλλογής αλλά και του βαθμού ομοιότητας που ορίζεται στο ερώτημα, η συμπεριφορά κάθε τεχνικής αλλάζει. Συνοπτικά, σε πολύ υψηλές τιμές του βαθμού ομοιότητας η χρήση της ΚΜΕ φαίνεται να είναι αρκετή με εξαίρεση όταν έχουμε μεγάλες συλλογές συνόλων και μεγάλο μέσο μέγεθος συνόλου όπου η συνεργατική λύση μπορεί να μειώσει τους χρόνους εκτέλεσης. Σε χαμηλότερες τιμές του βαθμού ομοιότητας και σε μικρές ή μεσαίου μεγέθους συλλογές συνόλων, η χρήση της ΜΕΓ αποδίδει καλύτερα. Αντίθετα, σε μεγαλύτερες συλλογές συνόλων, η συνεργατική λύση φαίνεται στην πλειοψηφία των περιπτώσεων να αποδίδει καλύτερα. Για πολύ χαμηλές τιμές του βαθμού ομοιότητας, όπου το φιλτράρισμα παύει να είναι ισχυρό, τις καλύτερες επιδόσεις έχουν τεχνικές ωμής βίας που εκτελούνται σε ΜΕΓ.

Στο τρίτο στάδιο, τα αποτελέσματα της συγκριτικής αξιολόγησης μαζί με το γεγονός ότι άλλα παράλληλα περιβάλλοντα όπως το MapReduce δεν μπορούν να προσφέρουν αποδοτικότερες λύσεις στο

πρόβλημα, αποτέλεσαν εφαλτήριο για την ανάπτυξη ενός υβριδικού εργαλείου που θα ενθυλακώνει τις καλύτερες ΚΜΕ και ΜΕΓ τεχνικές. Στόχος του υβριδικού εργαλείου είναι η διάσπαση του φόρτου εργασίας ανάμεσα σε μία ΚΜΕ και μία ΜΕΓ, αλλά και η μείωση του ανενεργού χρόνου των επεξεργάστων όσο το δυνατόν περισσότερο γίνεται. Η διάσπαση του φόρτου γίνεται μέσω δύο διαφορετικών στρατηγικών. Αρχικά, στη στρατηγική της ουράς, στην οποία όλο το πρόβλημα διασπάται σε συνδέσεις μικρότερου μεγέθους, οι οποίες εισάγονται σε μία ουρά από την οποία η ΚΜΕ και η ΜΕΓ τις εξάγουν για να έχουν συνεχώς φόρτο εργασίας. Η δεύτερη στρατηγική είναι η διχοτόμηση, στην οποία γίνεται η διάσπαση του φόρτου σε δύο μεγάλα τμήματα με τη ΚΜΕ και τη ΜΕΓ να αναλαμβάνουν από ένα. Το σημείο διχοτόμησης καθορίζεται από παράμετρο εισόδου μαζί με το βαθμό ομοιότητας. Έπειτα από εκτενή πειραματική αξιολόγηση, το υβριδικό εργαλείο που προτείνεται καταφέρνει να βελτιώσει τους χρόνους σε όλες τις περιπτώσεις. Γενικά, η στρατηγική της διχοτόμησης είναι πιο εύρωστη γιατί έχει μικρότερες απαιτήσεις σε μνήμη. Σε σύγκριση με μία πρόσφατη παράλληλη υλοποίηση που τρέχει εξ' ολοκλήρου σε ΚΜΕ, το υβριδικό εργαλείο καταφέρνει και πετυχαίνει ακόμα μεγαλύτερη αύξηση των επιδόσεων.

Το τελευταίο πρόβλημα που μελετήθηκε αφορούσε την αποδοτική υλοποίηση δύο τελεστών, αυτόν της εύρεσης τομής, όπου δοθέντων δύο συνόλων παράγει ένα τρίτο που περιέχει όλα τα κοινά στοιχεία, και αυτόν της σύνδεσης περιεκτικότητας, όπου η έξοδος είναι ο βαθμός περιεκτικότητας του ενός συνόλου στο άλλο. Το πρόβλημα της εύρεσης τομής έχει μελετηθεί έμμεσα στο περιβάλλον της ΜΕΓ μέσω εργασιών που αφορούν γράφους και συγκεκριμένα στο πρόβλημα της μέτρησης τριγώνων ενός γράφου. Όλες οι τεχνικές στη βιβλιογραφία για ΜΕΓ είναι αποδοτικές σε σύνολα μικρού μεγέθους, καθώς ο μέσος βαθμός των γράφων του πραγματικού κόσμου είναι αρκετά μικρός. Όταν αυξάνονται όμως τα μεγέθη των συνόλων, στην τάξη των εκατοντάδων χιλιάδων ή εκατομμυρίων, χρειάζεται καλύτερη αξιοποίηση της ΜΕΓ για να υπάρχει κλιμάκωση. Ως εκ τούτου, μία από τις βασικές συνεισφορές της διατριβής είναι η εξαγωγή των υπαρχουσών τεχνικών της βιβλιογραφίας, η τροποποίηση τους, καθώς και η ανάπτυξη υβριδικών τεχνικών που μπορούν να διαχειρίζονται πολύ μεγάλα σύνολα πιο αποδοτικά. Η αξιολόγηση των τεχνικών και για τους δύο τελεστές έγινε σε δύο προβλήματα, αυτό του απλού ζεύγους, όπου ο τελεστής τρέχει πάνω σε ένα μόνο ζεύγος συνόλων, και αυτό των πολλαπλών ζευγών, όπου ο τελεστής εφαρμόζεται σε κάθε πιθανό ζεύγος ανάμεσα σε k σύνολα. Για το πρόβλημα του απλού ζεύγους, το κατώτερο όριο χρονικής πολυπλοκότητας είναι γραμμικό, ενώ για το πρόβλημα των πολλαπλών ζευγών, η χρονική πολυπλοκότητα είναι τετραγωνική ως προς k. Η πειραματική αξιολόγηση των τροποποιημένων τεχνικών αλλά και των υβριδικών που προτείνονται στη διατριβή φανερώνει σημαντική αύξηση των επιδόσεων έναντι τεχνικών που τρέχουν στην ΚΜΕ, η οποία μπορεί να φτάσει και τη μία τάξη μεγέθους.

Συμπερασματικά, σύμφωνα με τα αποτελέσματα και τη γνώση που αποκομίστηκε κατά την εκπόνηση της διατριβής, για να επιτευχθεί καλή αύξηση των επιδόσεων σε προβλήματα διαχείρισης δεδομένων με χρήση ΜΕΓ, οι λύσεις πρέπει να υλοποιούνται προσεκτικά σε χαμηλό επίπεδο και να εκμεταλλεύονται στο έπακρο όλο το οπλοστάσιο της ΜΕΓ, από την ιεραρχία μνημών έως και τη διατήρηση ενός υψηλού βαθμού παραλληλισμού ανάμεσα στα εκτελούμενα νήματα. Επιπρόσθετα, για την επιτεύξη ακόμα καλύτερων επιδόσεων, ΚΜΕ και ΜΕΓ πρέπει να λειτουργούν συνεργατικά και συμπληρωματικά έτσι ώστε η ετερογενεία που προσφέρουν ως όμαδα να αξιοποιείται από λύσεις που στοχεύουν στην επίλυση μεγάλων προβλημάτων.

Abstract

Over the past years, the rise of General Purpose GPU (GPGPU) paradigm has become more evident in high-performance computing. The massive parallelism that GPUs offer at low cost is the catalyst for its adoption in numerous computational intensive applications, where tremendous speedup gains are reported due to the ease of parallelization of the algorithms they encapsulate. This thesis studied more advanced data management problems such as inequality and set similarity joins, along with the set intersection and containment operators on the GPGPU paradigm. Due to the inherent quadratic complexity of these problems, direct performance gains are unachievable. However, as shown in this thesis by designing co-processing techniques that take advantage of the different and complementary characteristics offered by CPUs and GPUs, tangible speedup gains are achieved over standalone implementations and other multi-threaded CPU solutions.

More specifically, for the inequality or theta joins, efficient implementations on a GPU are provided along with several optimizations to further improve performance. The best performing technique adopts a sliding window approach, and by resource reuse and through memory hierarchy exploitation, manages to achieve speedups of an order of magnitude for the aggregation query and of 2.7X for the select query over other CPU alternatives.

For the set similarity join, a co-processing CPU-GPU technique is proposed with the goal to utilize both processors. As a result, due to the execution overlap between the CPU and GPU tasks, significant speedups, which can reach up to 2.6X, are achieved over other alternatives in several cases. Furthermore, an empirical evaluation of the state-of-the-art GPU-enabled set similarity techniques is also presented, along with several key insights into under which circumstances each technique performs better. The main outcome is that there is no dominant solution; each technique has its own pros and cons based on specific dataset characteristics. Based on these findings, a hybrid framework for the set similarity join problem his proposed, which encapsulates the state-of-the-art CPU and GPU set similarity join techniques. By introducing two workload allocation strategies, the hybrid framework manages to utilize both the CPU and the GPU with high efficiency and achieves speedups of up to 3.25X over standalone techniques and of an order of magnitude over other parallel CPU solutions.

Finally, by adapting techniques initially proposed for graph analytics and matrix multiplication on the GPU, several hybrid GPU techniques for the set intersection operator on large sets are proposed. In addition, the first known technique for the set containment operator in the GPGPU paradigm, which involves a co-processing scheme between the CPU and the GPU, is introduced. In a thorough experimental evaluation, the proposed techniques for both operators manage to achieve performance speedup of an order of magnitude over the respective multi-threaded CPU alternatives.

Acknowledgements

This study was supported by the Hellenic Foundation for Research and Innovation (HFRI) under the HFRI PhD Fellowship grant (Fellowship Number: 1154). In addition, I would like to gratefully acknowledge the support of NVIDIA through the donation of the GPU (GPU Grant Program).

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Anastasios Gounaris, whose sincerity and encouragement I will never forget. His work ethic and constant unbiased search for the truth, were and are the cornerstones for my development and progress as an academic researcher.

I am extremely thankful to Dr. Georgia Kougka and Dr. Athanasios Naskos who welcomed me and made me feel familiar in my beginning at the former DeLab (now DataLab) research group. I would also like to thank my fellow lab members and friends for creating a friendly and positive environment to work in. More specifically, many thanks to Thodoris Toliopoulos, Georgia Baltsou, Ioannis Christoforidis, Nikodimos Nikolaidis and Kostis Varvoutas for the countless office hours I've shared with them and to Valentina Michailidou for her help, especially during my last year of doctoral studies. Last but not least, very special thanks to my friend Dr. Andreas Kosmatopoulos to whom I reached out for help numerous times and he always responded.

Finally, I want to thank my family, Michalis, Tania, Zois and Nikos for their continuous support, love and guidance. Knowing that I can always count on you is a source of strength and motivation for me every day.

Contents

1 Introduction

In the last decade, it has become widely accepted that the performance of modern processors is no longer limited by transistor density, but by power consumption. As a result, there has been a strong shift towards more heterogeneous environments, such as General Purpose GPU (GPGPU) [11]. Essentially, GPGPU enables co-processing on both traditional multi-core CPUs with many-core processors and GPUs. The main advantage of GPUs is the massive parallelism they provide at low cost. Moreover, the GPU runs slower, at about a third of the CPU's frequency, but in each clock cycle it can perform 100X more calculations in parallel than the CPU. This renders the GPU much faster than the CPU for tasks with lots of parallelism, and thanks to its energy efficiency, less expensive [14]. As of 2021, 6 out of the top 10 fastest supercomputers use GPUs as co-processors. In addition, those that are GPU-enabled are the most power-efficient ones[91].

Apart from the power efficiency, the main impact of GPGPU on high performance computing can be seen in the orders of magnitude of speedups achieved in domains such as machine learning and deep learning [67]. The main reason is that these domains involve embarrassingly parallelizable algorithms, also referred to as *regular* algorithms. The definitive feature of regular algorithms is that control flow and memory references are not data dependent and thus problems can be easily divided into components that can be executed concurrently [38]. Typical examples of regular algorithms include matrix multiplication and convolution computation, where programs running on GPUs benefit from performing similar independent operations on large amounts of data. More specifically, along with the massive parallelism they offer, GPUs provide almost 10X more memory bandwidth than CPUs, but have higher memory latency in accessing data. This makes GPUs better for predictable calculations, i.e. where the data needed from memory can be anticipated and brought to the processor at the right time, and worse at unpredictable ones.

On the contrary, in *irregular* algorithms, the input alongside the use of complex data structures such as graphs, determine the control flow and memory references. Moreover, irregular algorithms are generally less amenable to parallelization than regular ones, since their execution requires a lot of intercommunication among threads and shared resources, e.g., memory. Thus, parallelizing irregular algorithms on GPUs is challenging because, on the one hand, it presupposes in-depth knowledge of the hardware and, on the other hand, good GPU-enabled program design is often counter-intuitive. As a result, it is difficult to achieve significant speedups of the same order of magnitude for irregular algorithms with GPUs as for regular algorithms.

A typical example of irregular algorithms are joins, where objects of two or more collections are matched based on a join condition. Joins are a fundamental operation in data management, used in a variety of domains such as database systems [65], data mining [90], data cleaning [15] and entity resolution [3].

Over the past years, due to the need of processing the increasing amounts of data faster, there has been a plethora of research on how the massive parallelism provided by the GPUs could be

leveraged to improve the performance of joins. Because of its popularity and wide application, the most studied join type on GPUs is the equality join, for which speedups have been demonstrated in practice. Moreover, GPU hash-based and sort-merge algorithms achieve speedups of up to 5.5X and 10.5X, respectively, over CPU alternatives, as shown in [83].

On the other hand, there are several other types of joins that can benefit from GPU acceleration and have not been thoroughly investigated. The motivation of this thesis is to investigate less studied forms of GPU-accelerated joins, noted as *advanced*. The designation *advanced* is due to the fact that these type of joins are far more difficult to be implemented efficiently, compared with equality joins, especially in the complex parallel environment of the GPU.

The two advanced joins that the thesis focuses on are: (i) theta or inequality joins, where the join condition can be arbitrary, and (ii) set similarity joins, where, the records are set of elements, based on a similarity distance and a threshold, two sets may be considered similar. Additionally, because of the latter, the set intersection and set containment operators are also investigated independently in the GPU setting.

In summary, the thesis contributions are as follows:

- Efficient implementations of theta-join queries between two relations on a GPU are provided, along with a series of optimizations which yield performance improvements of an order of magnitude over CPU alternatives.
- A co-processing CPU-GPU technique to tackle the exact set similarity problem is proposed, which in several cases performs better than the state-of-the-art in due to efficient workload balance.
- A thorough evaluation of state-of-the-art GPU-enabled and CPU algorithms addressing the set similarity join problem is conducted, showing that no dominant solution exists and each different technique has its own sweet spot depending on specific dataset and query characteristics.
- A hybrid framework for the set similarity join problem is proposed. The framework encapsulates the best performing GPU-enabled and CPU algorithms and techniques, along with a partitioning mechanism to utilize both the CPU and the GPU appropriately. After an extensive experimental evaluation, it is shown that the proposed hybrid framework manages to achieve tangible performance speedups over the best standalone techniques, and even greater against multi-threaded CPU alternatives.
- Several hybrid GPU techniques for the set intersection operator on large sets are proposed. In addition, the first known technique for the set containment operator in the GPGPU paradigm, which involves a co-processing scheme between the CPU and the GPU, is introduced. In a thorough experimental evaluation, the proposed techniques for both operators manage to achieve performance speedup of an order of magnitude over the respective multi-threaded CPU alternatives.

Along with the contributions mentioned above, a short discussion is also provided on three common misconceptions that originate from the early years of GPGPU computing or that do not apply to modern data management on GPUs, and were tackled throughout the course of this thesis:

- *GPU can directly achieve a speedup by orders of magnitude*. This perception stems from the performance gains from using GPUs on easily parallelizable algorithms.
- *Data transfer makes GPU too expensive for data analytics*. The basic claim is that GPU's computational advantage is outweighed by existing slow interconnect between graphics cards and host devices.
- *Full occupation is required for efficient GPU algorithms*. Since GPUs are bandwidth-optimal, it is commonly believed as more beneficial to launch thousands of threads to hide memory latency.

The CUDA framework [68], which is proprietary to NVIDIA, was used to develop the proposed techniques. Moreover, all implementations of the proposed techniques are open source and available at `https://github.com/chribell`. Thus, anyone can evaluate and extend the work of this thesis.

The rest of this thesis is organized as follows. Chapter 2 introduces some preliminaries on GPGPU computing along with some common misconceptions, which are directly related to the technical challenged encountered. Chapter 3 describes the proposed solution to tackle theta-joins. In Chapter 4, a co-processing CPU-GPU technique for the set similarity join problem is introduced. In Chapter 5 an empirical evaluation of the state-of-the-art GPU-enabled techniques, along with the introduced hybrid co-processing framework for the set similarity join problem, are presented. Finally, Chapter 7 concludes the thesis and discusses possible future work. The list of publications is in the Appendix A.

2 Background

In this Section, preliminaries on GPGPU computing are given. NVIDIA's CUDA [68], on top of which every algorithm and technique proposed in this thesis was developed, is used as reference for the GPU architecture and programming model. The same basic principles apply to other vendors such as AMD [1] and to other programming models such as OpenCL [71]. The main advantages of CUDA are its ongoing support and development, its larger community, and the fact that it is more programmer friendly [69].

2.1 Architecture

A modern CPU contains several cores, each capable of executing instructions independently. The instruction set allows for multiple threads to execute simultaneously, but has limited parallelism due to data dependencies between threads. To increase computational power, CPUs have been equipped with even more cores. However, as the number of cores increases, so does the time required to switch from one thread to another. This phenomenon is known as latency. Latency grows linearly with the number of cores, and thus quickly becomes a bottleneck limiting performance. On the other hand, GPUs are designed for parallel computation and differ fundamentally from CPUs in that they use more hardware resources for computation and less resources for caches and flow control. As shown in Figure 2.1, a GPU has much more processing units, also referred to as cores, but less cache and control units compared to a CPU. This design decision favors high-throughput, compute-intensive operations as opposed to operations with complex branching.

GPUs divide the cores into groups called *Streaming Multiprocessors (SMs)*. Each SM includes also other units such as ALUs, instruction units, memory caches for load/store operations, and follows the *Single Instruction Multiple Threads (SIMT)* parallel processing paradigm. Further details on SIMT are presented in Section 2.4. Overall, GPU cores execute the same instruction on different data items, whereas CPU cores typically execute different instructions thus raising

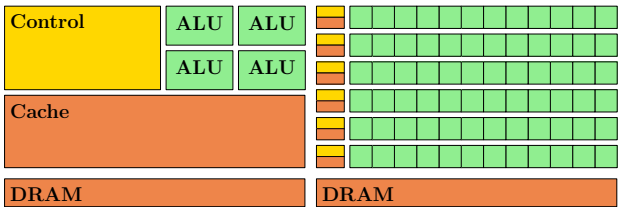


Figure 2.1: Architecture difference between CPU (left) and GPU (right) [69].

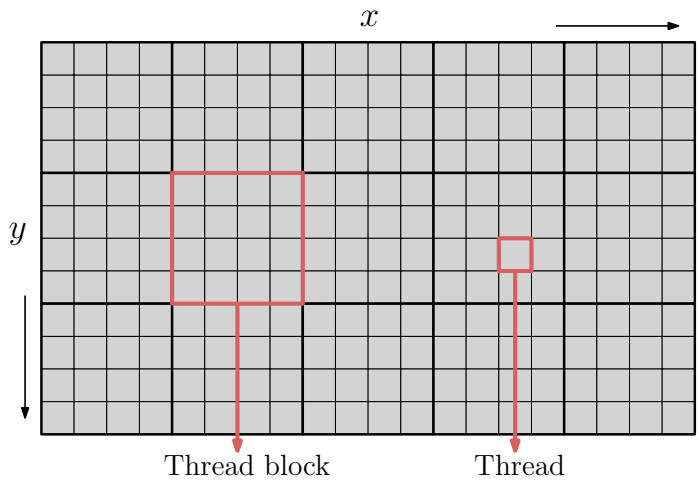


Figure 2.2: An example kernel grid with $5 \times 3 = 15$ blocks, each consisting of 16 threads.

the need for more advanced program flow control. In general, modern CPUs are designed to minimize the latency of individual threads, whereas GPUs target throughput maximization [66].

In CUDA terminology, the CPU and the main memory are referred to as *host*, while the GPU and its own memory are referred to as *device*. Throughout the thesis, the terms CPU and host (resp. GPU and device) are used interchangeably.

2.2 Thread Organization

In CUDA, threads are organized in logical blocks called *thread blocks*. Essentially, each thread block is an array of threads with up to three dimensions. Every function to be processed in parallel by the GPU is called a *kernel*. Each kernel is executed by multiple thread blocks, which form the *kernel grid*. The kernel grid can be regarded as an array of thread blocks with up to two dimensions. An example of a kernel grid is illustrated in Figure 2.2. All threads in the same block are executed on the same SM and are allowed to inter-communicate. CUDA can schedule blocks to run concurrently on a SM according to the capacity of each SM in terms of concurrent blocks and active threads.

Scheduled blocks are further partitioned in groups of 32 threads called *warps*. Threads within a warp are called *lanes* and share the same instruction counter, thus they are executed simultaneously in a *Single Instruction Multiple Data (SIMD)* manner. As an example, the SM in Figure 2.3 has 128 cores; hence it supports up to 4 warps to be executed concurrently. A 2D thread block of size 16×5 to be executed on this SM, requires a total of 80 threads. Therefore, three warps are to be launched and the third warp will contain only 16 threads (*half-warp*). As a result 48 cores in the SM will be idle during the thread block's execution.

In general, warps are the smallest unit of execution. At any time, a SM chooses the warps that are ready for execution, so that long waiting times (i.e., waiting for other warps to fetch operands) are hidden. There are two cases of thread divergence, which degrade performance, namely *inter-warp*,

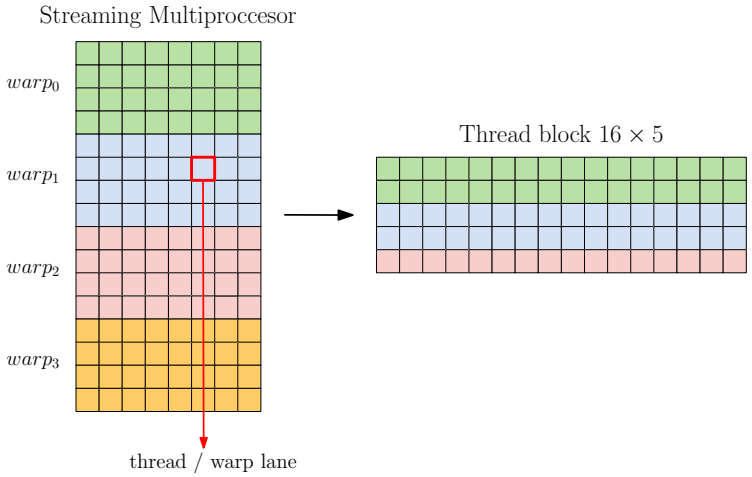


Figure 2.3: A warp-to-thread block mapping example.

when concurrent warps run unevenly and *intra-warp*, when warp lanes take different execution paths. The latter is also simply referred to as *warp divergence* [69].

The kernel grid and thread block sizes are defined as kernel launch parameters. Capacity restrictions may play an important role in kernel execution. There are hardware constraints on the number of (A) resident (or else, scheduled or active) blocks per SM, (B) the number of resident warps per SM, (C) the number of scheduled threads per SM, and (D) the number of threads per block. As such, the number of active threads in a kernel cannot exceed the quantity $min\{(A) \times (D), 32 \times (B), (C)\}$ multiplied by the number of SMs on the GPU.

2.3 Memory Hierarchy

The memory structure on a GPU is hierarchical, as depicted in Figure 2.4. Moreover, GPU memories are distinguished between off-chip and on-chip. Off-chip memories include the *global, constant, texture* and *local* ones. The global memory is the largest but slowest memory, and it is the place where data is transferred from the main memory through a PCI-E link or more recently, with faster interconnects such as NVLink [60]. Data resident in global memory is visible to all threads.The contents of the global memory can also be cached in the L1 and L2 on-chip caches. The constant memory is much smaller (e.g., 64 KBs), but leads to significantly shorter access times. It is used for read-only data and is cached in a different place than L1 and L2 caches. The key reason for the short times is that a single thread can read once a value from constant memory and broadcast the value read to all the other threads in the same half-warp. The texture memory is also read-only, and is optimized for access patterns where subsequent accesses are spatially close. The local memory is part of the global memory and is used when the registers needed for a thread are full or cannot hold the required data. This is called *register spilling*. Each thread gets allocated its own part of the local memory.

The on-chip memory modules consist of the *shared* memory, the caches, and the registers. Shared memory is the second fastest memory type and its latency is two orders of magnitude lower

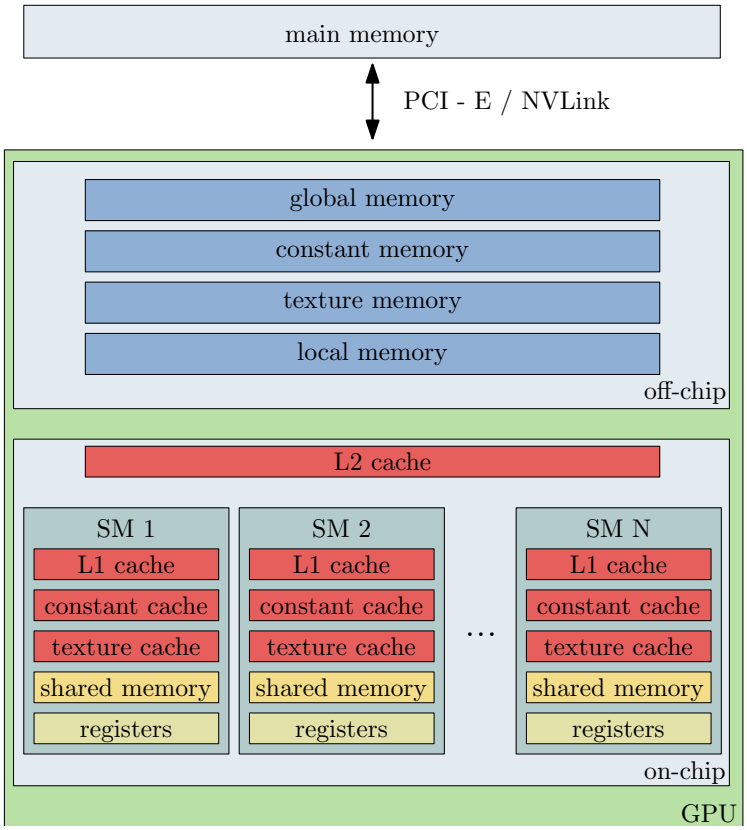


Figure 2.4: Memory hierarchy for CUDA-enabled GPUs.

than that of the global memory. Each SM has its own shared memory. Moreover, the contents of shared memory are accessible by all threads in the same block. This feature renders shared memory particularly useful since it also enables cooperation between threads within a block. The L1 and L2 caches serve the global memory, and can spill data to the local memory. There are also caches for the constant and the texture memory. Finally, the registers are the fastest memory type, similarly to CPU chips. More specifically, they contain the instructions of a single thread and the local variables during the lifetime of that thread. If a thread requires more registers than a threshold, this leads to a reduction of the threads allowed to run concurrently.

CUDA can schedule blocks to run concurrently on a SM depending on the shared memory and registers used per block. Increasing either of these factors can lead to limited concurrent block execution, which results in low occupancy. *Occupancy* is defined as the ratio of active warps on a SM to the maximum allowed active warps per SM. Maximizing occupancy is a good heuristic approach but it does not always guarantee performance gain. On the contrary, maintaining high warp *execution efficiency*, i.e. the average percentage of active threads in each executed warp, is a more robust approach for data-management tasks.

2.4 SIMT Programming Model

SIMT can be regarded as a more flexible version of SIMD in Flynn's taxonomy [69]. Moreover, both SIMT and SIMD approach parallelism through broadcasting the same instruction to multiple execution units. However, they differ fundamentally in how that broadcast is implemented. In a SIMD architecture, each instruction applies the same operation in parallel to many data items. SIMD is typically implemented using processors with vector registers and execution units; a thread issues vector instructions that are executed in a SIMD fashion. In a SIMT architecture, rather than a single thread issuing vector instructions applied to data vectors, multiple threads issue common instructions for arbitrary data. To support the SIMT programming model at a high level, CUDA executes threads as bundles of 32, i.e. warps. Each thread can access its own registers, load and store from divergent addresses, and follow divergent control flow paths [69].

When a warp is selected to be executed, SIMD hardware executes the same instruction for all threads in the same warp. If all threads within a warp follow the same execution path when working their data, execution efficiency is high. When threads within a warp take different control flow paths, the SIMD hardware will take multiple passes through these divergent paths. For example, in an if-else scenario, one pass will execute those threads that follow the if part and another pass will execute those that follow the else part. During each pass, the threads that follow the other path are not allowed to take effect. These passes are sequential, thus will add to the execution time and decrease Instruction-level Parallelism, also noted as *ILP*, which is a measure of parallelism within threads [49].

2.5 Common Misconceptions

Having established a background on the GPGPU paradigm, the afore-mentioned "misconceptions" in introduction are further analyzed. The fact that it is common for someone to expect that GPUs can directly achieve a speedup by orders of magnitude, it has been debunked as a myth since the last decade [52]; it is more common to expect speedups at the order of a few factors or even less. This is verified by the latest advances in GPU-enabled data analytics when taking into account the overhead to load data on GPU's main memory [87]. So, why does a 100-fold increase in the number of cores is not accompanied by a proportional performance improvement? First, it is well-known that the improvements of any parallel algorithms are bounded by the Amdahl's law, and actually, this law gives a very loose upper bound on potential improvements [38]. Multiple additional factors, such as costs to transfer data between different types of memory and expensive memory access patterns, thread divergence, hidden serialization of instructions, which are specific to the nature of GPU render this bound much lower in practice. Second, the speedups refer to comparisons against optimized multi-threaded CPU solutions that can be effectively parallelized in many scenarios. That is, they most commonly computed as the ratio t_{cpu}/t_{gpu}, where t_{cpu} (resp. t_{gpu}) denotes the execution time on a CPU (resp. GPU). In this ratio, the nominator is typically low. Third, not all algorithms naturally lend themselves to parallelization, e.g., they may require updating a single set of points or encapsulate multiple points of synchronization, or more importantly, the data values determine the instruction flow (i.e., which exactly data need to be accessed). The latter is commonly encountered in data management operations, such as joins, and prevents the development of parallel solutions with blocks running independently of each other.

The points above may lead someone to believe that the overheads are too high, e.g., data transfer makes GPU too expensive for data analytics as claimed in [87]. First, it should be considered that bottlenecks, such as slow PCI-E, are expected to be surpassed by hardware advances in interconnecting technology. As an example, NVLink 2.0 is reported to scale to arbitrary data volumes exploiting cache-coherence and facilitating data co-processing between CPU and GPU [60]. But even with the current technology of slower CPU-GPU interconnects, the data transfer costs are not necessarily higher than the cost to process the whole dataset on the CPU exclusively. However, this is not always the case. Even if the transfer cost is high, this does not imply that the GPUs should not be used for data joins. Depending on the problem this cost may be mitigated to a large extent thanks to the overlapping between execution and transferring data, always keeping in mind that the expected speedups may not be very high.

So, what is the right way to employ a GPU on a problem that cannot easily be parallelized? A common statement is that for efficient GPU algorithms, full occupancy is required. This stems from the fact that GPUs are optimized for memory bandwidth and therefore launching thousands of threads is the correct approach to hiding memory latency. The actual answer is "it depends". More specifically, the first step towards optimized GPU parallelization is to understand where most execution time is spent. Hence, a problem may be memory-bounded, i.e. most time is spent on memory transactions, compute-bounded, i.e. most time is spent on compute operations, or a mixture of both. For memory-bounded problems, memory coalescing and the exploitation of the fastest on-chip memories are required to alleviate costly memory transactions. On the other hand, for compute-bounded problems, reducing branch divergence and optimizing ILP at warp level are required to improve performance. Overall, even though full occupancy is a good starting-point heuristic, the fact that there may be memory accesses determined by unforeseen data values calls for striking a balance between fully utilizing the GPU cores and allowing threads to operate on on-chip memory spaces; the latter, in practice, decreases the number of running threads but also increases the ILP and improves execution efficiency [94].

2.6 Discussion

A general rule of thumb in GPU computing in terms of performance is to aim for good ILP and maximize memory throughput. This can be regarded as striving to achieve a balance between these two aspects, which is a prerequisite for effective occupancy. For regular algorithms, finding this equilibrium is straightforward and requires less effort. For irregular algorithms, on the other hand, achieving effective occupancy becomes more challenging due to non-trivial memory access patterns and branching. The advanced joins investigated in this thesis fall into the category of irregular algorithms.

Developing high performing GPU programs for irregular algorithms entails a sequence of coherent design choices. For example, one must carefully choose factors such as thread block size, the total number of blocks and the amount of work per thread that needs to be performed. These decisions have a significant impact on overall program behavior, both in terms of latency and bandwidth utilization. In addition, they may also lead to resource constraints and inability to fully utilize the fastest on-chip memories. A characteristic example is the excessive use of registers,

which, when a predefined limit is exceeded, leads to register spilling to the slow global memory and ultimately to an increase in memory traffic.

To tackle such issues, it is sometimes necessary to reform the program at the algorithmic level. For example, it is common to reorder certain tasks in order to split the algorithm across multiple kernels. The main drawback of this approach is the additional materialization overhead on the slow global memory. Another approach that minimizes the global memory footprint involves the use of a single kernel where each thread block performs every algorithm step on a tile of items exploiting the memory hierarchy [87].

3 Theta Joins

The most general but less studied form of binary joins is *theta (or inequality)* joins. A theta-join is an inner join operation, which combines tuples from different relations, if and only if a given theta (θ) condition is satisfied. It generalizes equality joins (or equi-joins) as it allows the use of inequality operators within the join condition. In a generic theta-join, all possible combinations (i.e., the cartesian product) may need to be evaluated, which leads to $O(n^2)$ time complexity. Additionally, the worst case space complexity is when $O(n^2)$ combinations of records satisfy the theta condition. In practice, a theta-join over a couple of hundred thousands of records may take hours in a modern DBMSs [48], whereas an equi-join may complete in a few seconds or less.

GPUs have been proposed for simpler relational operators, such as sorts, selects, aggregates and equi-joins, e.g., in [4, 35, 88, 114], but no solution tailored to theta-joins on GPUs has been proposed. In this thesis, the first proposal that explicitly targets theta-joins using a GPU is proposed. The closest work to the presented proposal is the consideration of non-indexed nested loops in [35], which can cover theta-joins as well as equi-joins. Compared to [35], the present work elaborates on all practical details, including the handling of join output, whereas in [35], nested loops are discussed at a high level only.

The key features of the proposed solution are summarized as follows: (i) no assumptions are made about the θ condition; therefore, there is no kind of preprocessing or knowledge about data distribution and selectivity; this renders the proposed solution as generally applicable as possible, (ii) a high processor utilization is achieved, despite the inherent presence of thread branching, i.e., the threads follow a different execution path depending on the result of the predicate evaluation, which typically returns both true and false values, and (iii) a series of optimizations that address issues related to data reuse, minimizing accesses to slow device memory, limited memory, utilization of memory hierarchies on a typical GPU, and data layouts that facilitate memory accesses are provided. Overall, it is shown experimentally that these optimizations can lead to speedups of up to an order of magnitude.

3.1 Background: theta-joins in MapReduce

Theta-joins have been previously investigated for MapReduce in a shared-nothing parallel environment [70]. Despite being significantly different from the GPGPU context, the MapReduce techniques can be leveraged to build a solution for GPUs, when the input is too large to fit in the device memory.

In a MapReduce setting, multiple machines work in parallel and inter-communicate through the network. The main problem addressed in [70] is that each machine may have memory limitations, so that the goal is to allocate to each machine as small workload chunk in terms of memory requirements as possible. To this end, it is helpful to represent the work of a theta-join as a matrix,

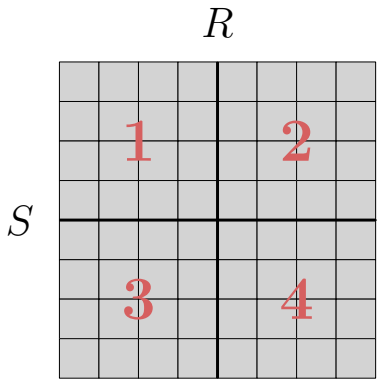


Figure 3.1: A simple 8×8 join matrix and its partitioning into 4 square regions.

the *join matrix (JM)*. A join matrix has as many rows (resp. columns) as the number of records in the first (resp. second) relation to be joined. Figure 3.1 shows a matrix that represents the theta-join between two relations, R and S, each having eight records. Each cell in a JM represents a unit of work in the theta-join, which is equal to the evaluation of the theta predicate of a single pair of records.

The key contribution of the work in [70] is the proof that, given the number of the reducer machines available, the memory requirements are minimized, when each reducer machine gets an amount of the workload, which is visually represented as a square region in the JM. For example, in Figure 3.1, four reducers are assumed and the workload is split into four square regions. Note that in the generic case, JM is not square. The special case where one input relation is much smaller than the other is also considered.

The main difficulties in transferring the solution in [70] to the GPUGPU setting is that, if on the one side, each GPU thread becomes responsible for a single cell, then the amount of cells required may well exceed the GPU capacity. On the other side, if a GPU thread becomes responsible for multiple JM cells, then load balancing becomes even more difficult. Finally, if the theta-join, and consequently the JM, is processed by multiple kernels, this aggravates the overhead of sending data from the host to the device due to data replication. In the example show in Figure 3.1, each record of each relation is transferred to two reducers. The smaller the reducer regions, the higher the replication factor. Consequently, there are significant tradeoffs between the various workarounds.

3.2 Proposed solution

In this section, first a concept mapping between MapReduce and GPGPU is presented. Based on this mapping, the design choices on how to divide the work between the host and the device are presented, along with a high level overview of the proposed solution. Last, the types of queries studied are presented.

3.2.1 Concept Mapping

In order to transfer results from the solution for the MapReduce paradigm discussed in Section 3.1 to the GPGPU paradigm, it is a prerequisite to understand the association and the differences between the concepts of the two frameworks.

The map process in [70] is responsible for enforcing the desired data partitioning and does not contribute to actual theta-join processing. The latter is performed by the reducers in parallel using a single phase. In the GPGPU setting, the responsibility for data partitioning rests with the host. However, workload allocation to the device can be performed in multiple iterations due to memory limitations. This issue is further elaborated in Section 3.2.2.

A second difference is that in the MapReduce paradigm, a reducer machine is a distinct processing unit with its own main memory and processing elements, to which part of the workload is allocated. All reducers run in parallel, whereas interaction between reduce-side processing of different key-value pairs on the same reducer machine is possible [57]. At the hardware level, a CUDA-enabled GPU has a number of mutually independent SMs. As such, SMs can be regarded as analogous to reducer machines. However, at the software level, the workload of a kernel is divided in thread blocks. Thread blocks are processed in parallel by the SMs and multiple blocks can be simultaneously resident on the same SM, as already explained. Consequently, the division of workload needs to be at a finer granularity than in MapReduce-based theta-joins, where it is adequate to divide the workload into as many pieces as the number of reducer machines.

3.2.2 Memory Limitations

GPUs are more suitable for problems of fixed input and output size, so that the exact amount of required memory can be accurately estimated before execution and consequently, no extra logic to co-ordinate writes and handle conflicts is required. In theta-joins, the primary focus is on output because the memory requirements of the input are much lower. For example, consider a join between 50K$\times$25K tuples of 8 bytes each. The whole input can fit into less than 1MB, but the cartesian product requires approximately 20GBs. A 10-fold increase in each of the input relations incurs a 100-fold increase in the output, i.e., the input fits into 10MBs, while the output exceeds 1TB. In generic theta-joins, there is no prior knowledge of the exact output size. So, any solution should account for the worst case as well.

To tackle this issue, two possible approaches are investigated. The first one assumes that the device processes a whole JM in a single execution. The most straightforward implementation is to split the device memory that is available (i.e., not occupied by the input) into as many buffers as the number of SMs. When a buffer becomes full, the execution is suspended until a flushing operation, which transfers the partial output to the host memory through the PCI-E bus, is completed. However, a basic requirement to this action, is to know the SM that executes a specific thread at a given moment. CUDA API does not provide this kind of information directly, but can retrieve the SM identifier indirectly by executing a PTX[1] command. The main drawback of this approach is that it leads to poor performance because the parallel execution is suspended frequently, and also, the device code becomes more complex.

[1] PTX is a pseudo-assembly language used for code translation in the CUDA framework.

Parameter	Description
R, S	Relations to be joined
O	Output relation
r, s, o	Tuple sizes in bytes
$\|R\|, \|S\|, \|O\|$	Cardinalities of relations
$\|\|R\|\|, \|\|S\|\|, \|\|O\|\|$	Sizes of relations in bytes
$\|\|O\|\|_{max}$	Maximum space needed in bytes
$\|R'\|, \|S'\|$	Cardinalities of sub-relations in each iteration
M_h	Host memory
M_d	Device memory
T	Number of threads executed
$B = B.x \times B.y$	Thread block size and its dimensions
p	Chunk size

Table 3.1: Notation.

In the second approach, which is the one followed in the proposed solution, the host iteratively feeds the device with as many input data as the GPU memory can support (i.e., their cartesian product fits into the remaining global memory). In this way, control flow activities are assigned to the host side and thus, complex memory handling operations on the device are avoided.

3.2.3 High-level Approach

In this section, the high-level summary of the proposed solution for $R \bowtie_\theta S$ joins is presented. The main notation used throughout the rest of the chapter is shown in Table 3.1.

Host Role

In each iteration, the host sends to the device a single partition with as few data as possible in order to maximize the amount of memory available for the output, while balancing the workload across iterations. Each partition is processed in parallel by the GPU SMs, however different partitions are processed sequentially. More specifically, in each iteration, sub-relations $R' \subseteq R$ and $S' \subseteq S$ are sent to the device so that their cartesian product fits in the device memory M_d. The lower bound of number of partitions, which equals to the lower bound on iterations, is calculated as follows: $t = \lceil ||O||_{max}/M_d \rceil$, where $||O||_{max} = |R| \times |S| \times o$.

There are several ways to perform the partitioning. Leveraging the theorems presented in [70], which ensure that the memory requirements for holding the input are minimized, three cases are distinguished: (i) both R and S are multiples of $\sqrt{|R||S|/t}$; (ii) one relation is much smaller than the other one, and more specifically $|S| < |R|/t$; and (iii) one relation is less or equal than the other one and $|R|/t \leq |S| \leq |R|$ is satisfied.

The first two cases are treated similarly to [70]. In the first case, the JM is partitioned into squares of side $\sqrt{|R||S|/t}$, i.e., $R' = S' = \sqrt{|R||S|/t}$. In the second case, i.e., when one relation is much smaller than the other one, the matrix is partitioned into rectangles of $|S| \times |R|/t$, i.e., $S' = S$ and $R' = |R|/t$ (here, without loss of generality, it is assumed that S is the smaller relation). In the third case, first, as many square partitions of size $\sqrt{|R||S|/t}$ as possible are

created. Then, to cover all the JM cells, smaller rectangle partitions are created. This case is treated differently than in [70] and involves a trade-off between aggravating the memory requirements and increasing the number of iterations. In the proposed solution, memory requirements are traded at the expense of more iterations.

Device Role

In each iteration, the device becomes responsible for a rectangle region of the JM, corresponding to a $R' \bowtie_\theta S'$ join. Since JM are two-dimensional structures, threads are grouped in blocks of two dimensions, which in turn, are organized in a grid of two dimensions, exactly as shown in Figure 2.2. As such, the problem of efficient processing theta-joins on a GPU is essentially reduced to the lower-level problems of deciding on (i) the exact role of each thread and (ii) the thread organization.

A subtle detail has to do with the memory allocation on the host to store the intermediate results produced by the device in each iteration. If unpinned memory is allocated, memory paging can take place which may increase data transfer times. In contrast, pinned memory allocated through the CUDA API, allows data access directly through the physical address and as a result decreases data transfer times significantly. Across iterations, and in cases where the host memory M_h is not sufficiently large to store the complete final output, access to the secondary storage is needed. Therefore, the existence of a function, which transfers partial output results from the primary to the secondary host memory, is assumed to be in place. Nevertheless, the time overhead of this function is fully hidden by the processing on the device side. Finally, it is assumed that the host memory is not smaller than the device memory, i.e., $M_h \geq M_d$ always holds.

3.2.4 Types of Queries Investigated

The proposed solution investigates two types of theta-join queries. The first one performs an aggregation on top of a theta-join. This type is characterized as input-bound, since the size of the output is very small. Consequently, its processing can typically be performed in a single iteration, i.e., $R' = R$ and $S' = S$ unless the input sizes are very large, which is rather unlikely in theta-joins. Larger inputs are not a big problem because the partitioning described previously can also be applied. Dealing with theta-joins combined with an aggregation, apart from being interesting and useful in its own right, enables the focus on implementation details that are orthogonal to the size of the output.

The second type of query corresponds to the generic theta-joins, where the focus shifts to writing the result, which usually does not fit into the device memory. That is, the generic theta-joins are clearly output-bound.

In the next two sections, the approaches followed for the effective implementation of these two types of queries are analyzed respectively. In some parts, to facilitate understanding, exact numbers for the implementation decisions are given according to the characteristics of the GM204-200 or simply GTX 970 GPU (compute capability *5.2*)[2]. The design decisions are largely

[2] Compute capability identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU [69].

driven by the technical specifications of GPUs. However, they are hardware and framework-independent, in the sense that they can apply to any GPU after a straightforward parameter adjustment.

3.3 Input-Bound Theta-Joins

First the implementation of a theta-join followed by an aggregation is investigated. Through this simplification, the focus falls on efficient thread specification and indexing without worrying whether the size of the produced results is larger than the device memory.

Without loss of generality, it is assumed that the relations to be joined consist of three simple integer attributes: a tuple identifier *id*, and two additional fields, *a*, *x*, which are used in the theta-join and aggregation conditions, respectively.

```
typedef struct {
   int id;
   int a;
   int x;
} Tuple;
```

Listing 3.1: Tuple representation (AoS)

```
typedef struct {
   int id[N];
   int a[N];
   int x[N];
} Relation;
```

Listing 3.2: SoA data layout for a relation

The simplest format to store tuples in memory is referred in literature as Array of Structures (AoS). In AoS, the relations are stored as arrays, and the content of each array cell is a tuple as shown in Listing 3.1. However, this data layout is avoided and GPU data management solutions prefer an alternative, which is called Structures of Arrays (SoA) (shown in Listing 3.2). In SoA, there is a different array for each attribute, which is shared among all the tuples in the relation. For completeness, both data layouts are examined, starting from the AoS one.

The aggregate function used in the query calculates a *sum* on the $S.x$ field provided that the theta-join condition is satisfied as shown below. Consequently, the device's output is a single number. An SQL representation of the aggregate theta-join query is shown in Listing 3.3.

```
SELECT SUM(S.x)
FROM R, S
WHERE R.a > S.a;
```

Listing 3.3: Aggregate theta-join query

3.3.1 Partitioning the Join Matrix in Threads

Initially, the workload allocated to each thread concerns the evaluation of the join condition on a single pair of tuples. As such, the size of the flattened grid is the size of the JM. Also, due to the 1-to-1 correspondence between the grid and the JM, tuple mapping is indirectly performed through the thread indexing and vice versa.

Then, the main issue is how to partition the grid in thread blocks. Since there are no memory constraints, one could be tempted to divide the JM in as many thread blocks as the number of SMs. However, this is practically impossible due to limit on the number of threads per block in modern GPUs. For example, in the GTX 970 graphics card, there are 13 SMs and the maximum

Algorithm 3.1 Work per thread in the naive implementation.

1: load R tuple
2: load S tuple
3: **if** θ condition is met **then**
4: update sum in the global memory using `atomicAdd`
5: **end if**

number of threads per block is 1024. A very small theta-join over two relations of 2K tuples each, would require $\frac{2000 \times 2000}{13} > 307K$ threads per SM, which is two orders of magnitude higher than the current limit. So, driven by the GPU specifications, larger grid dimensions need to be considered for the kernel launch not to fail. The resulting grid has size of $\lceil \frac{|R'|}{B.x} \rceil \times \lceil \frac{|S'|}{B.y} \rceil$, where the product of the two thread block dimensions $B.x$ and $B.y$ needs to be at most 1024.

3.3.2 Naive Implementation

A naive implementation is as follows (see also Algorithm 3.1). First, the input relations are copied to the device memory. As a preprocessing step on the host, unnecessary fields are filtered out before the copy. Extra memory is also allocated for a single *sum* variable to hold the result of the aggregation. Every thread in which the theta condition is satisfied needs to increment the *sum* value by the corresponding $S.x$ value. Since threads run in parallel, in principle, multiple threads may try to update the *sum* variable simultaneously. Consequently, a simple arithmetic operation leads to information loss. To deal with such issues, CUDA provides a set of atomic functions. An atomic function guarantees that the corresponding operation will be performed by each thread without interference from other threads[3].

The drawback of using atomic functions lies in the suspension of parallel execution as threads are scheduled and executed in a sequential manner. This happens because no other thread is allowed to access the variable's address until the operation is completed by the thread that is currently manipulating the variable. Apparently, the performance degradation depends on the number of theta conditions satisfied and the warp execution scheduling. In the worst case, the number of instructions forced to be executed sequentially is the maximum number of active threads supported by the GPU.

3.3.3 Improvements

Although the naive implementation is quite simple, is susceptible to some improvements. The main motivation behind them has been to exploit on-chip memories, and increase data reuse and achieved occupancy. The theoretical occupancy is calculated prior to execution by the launch parameters and the register count per thread. The achieved occupancy is the coverage ratio of the theoretical occupancy. As mentioned before, GPU hides its latency by massively executing threads. Occupancy is a metric that indicates the extent to which the GPU is kept busy. Higher

[3] In the implementation `atomicAdd` is used.

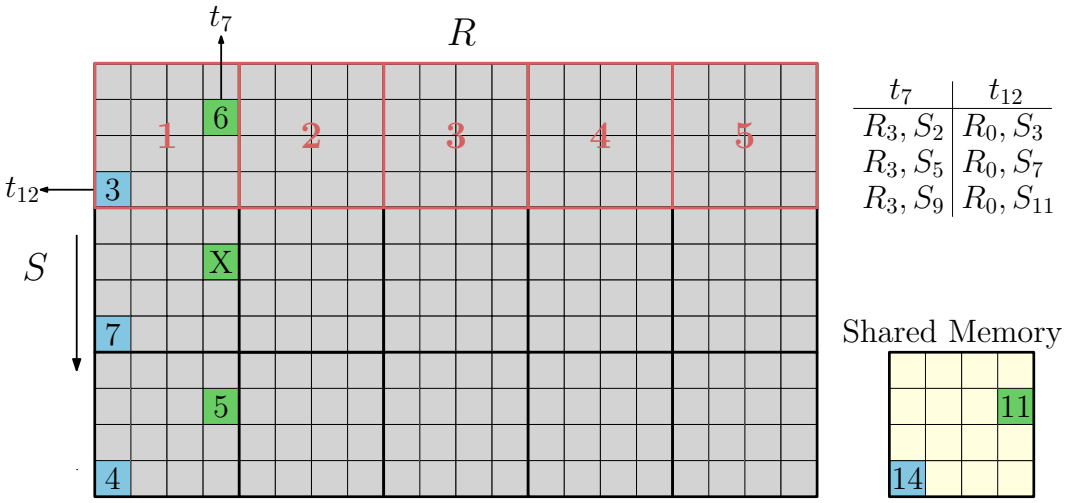


Figure 3.2: Illustration of theta-join processing according to the shared memory approach. Each thread reads a single record from the larger relation R and iterates over several records from S.

occupancy does not guarantee better performance. However its maximization is a good heuristic approach.

Employing Partial Sums

The first improvement concerns the traffic reduction observed to access a single memory address by multiple threads. The goal is to divide the sum calculation into smaller parts, i.e. to employ partial sums. This requires a post-processing step to produce the final sum.

More specifically, a partial sum is calculated for each thread block of the grid. The atomic add function is still used, but unlike the naive implementation, the kernel outputs a number of partial results. Hence, a secondary procedure is necessary to produce the final result. There are two options, either pass the partial sums as input to a second kernel, which performs a parallel reduction [49], or copy the results back to host memory and calculate the final result with a simple iterative process on the CPU. The first choice is preferable since the intermediate results are already stored in the device memory and it takes negligible time.

Exploiting the Shared Memory

The second type of improvements concerns the usage of the shared memory. This memory allows threads to access data loaded from global memory by other threads within the same block, thus enabling data reuse.

Initially, the goal is to minimize the number of global memory transactions. If a 2D grid of 2D thread blocks is launched as before, and each block writes its partial sum in shared memory, then the use of an atomic function can be avoided to a large extent. An outline of this approach is as follows. Every thread in which the theta condition is satisfied loads the corresponding $S.x$ value to the shared memory and then, after a synchronization between the block's threads to avoid race conditions and ensure correct results, each block stores its partial sum in global memory. This

approach improves the simple extension of the naive implementation as it avoids atomic function use, but incurs as many global memory write transactions as the number of thread blocks, which can still be very large. Further, it does not essentially exploit data reuse.

An alternative and more efficient approach, which addresses the issues mentioned above, is to assign more work to each thread. In other words, to allocate more pair comparisons to a single thread, and as such, this significantly departs from the initial naive approach. By launching less threads and assigning more work to each thread, data reuse is exploited through shared memory and further minimize global memory write transactions. Specifically, a 1D grid of size equal to $\lceil \frac{|R|}{B.x} \rceil$ is launched, where R corresponds to the largest input relation.

A visual representation of how the shared memory approach manipulates the JM is illustrated in Figure 3.2. More specifically, it is assumed that the block dimensions are $B.x = B.y = 4$ and the size of R is $|R| = 16$, which yields a 4×1 grid. In each thread block, $B.x$ records from R are read once. The red boxes in the figure show the comparisons for which each of the 16 threads in each of the 4 blocks is responsible at the initial stage. In general, a thread block is responsible for all S rows rather than only the first $B.y$ ones. When the initial comparisons finish, the red boxes slide by $B.y$, and this is repeated until all S rows are processed. Furthermore, the figure shows the example executions of the threads t_7 and t_{12} in the first block. The colored JM cells depict the cells examined by these two threads and the shown number is the $S.x$ value when the join condition is met. On the upper right part of the figure, all the combinations examined by these two threads are depicted; in the example, it is assumed that the pair of R_3, S_5 does not satisfy the condition and the corresponding cell in the JM is marked by X.

The choice of using the size of the largest relation for the grid size is important. In this way, more threads are executed and thus indirectly help the device to hide its latency. Also, each thread makes fewer comparisons, which also leads to fewer non-coalesced accesses to the global memory.

In addition, as shown in the lower right part of Figure 3.2, the shared memory holds the partial results for each thread in a single block. So, when all possible combinations have completed their evaluation, every block has a set of partial sums in its shared memory. To produce the corresponding partial sum for the complete block, the parallel reduction technique mentioned previously is used. After this stage, the intermediate result is stored in global memory. Since a 1D grid with fewer blocks is launched, the number of global memory write transactions is decreased significantly compared to the previous approach, due to the fact that data reuse takes place.

An additional optimization concerns the use of the fastest memory on the device, namely the registers. More specifically, the participating fields of the R relation (i.e., $R.a$ in the example query) can be stored in the registers, as they are reused during the execution of each thread. Hence, the number of global memory read transactions are further decreased. In general, registers must be used with caution because of their limited size. If their capacity is exceeded, register spilling occurs, i.e., register contents are spilled to local memory, which is off-chip. Algorithm 3.2 presents the work allocated to a single thread, which becomes responsible for a single R tuple.

Data Layout

Besides the improvements regarding exploiting unused memory, the data layout used must also be investigated. In the theta-join problem, a relatively high performance penalty is due to global

Algorithm 3.2 Work per thread when exploiting the shared memory.

1: load $R.a$ value to a register
2: load S tuple with id equal to the thread $B.y$ coordinate
3: **repeat**
4: **if** θ condition is met **then**
5: update corresponding shared memory cell
6: **end if**
7: load S tuple with id larger by the $B.y$ size of the block
8: **until** no more S tuples

memory accesses, while the computations are less expensive. According to the AoS data layout, the values of the same field in different records are not stored in consecutive memory addresses. To optimize memory access, consecutive threads should access consecutive memory addresses. This limitation is addressed by adopting the SoA data layout, which bears similarities to the column-oriented storage model in databases.

The AoS data layout can be analyzed a little further by taking into consideration the hardware specifications and the way data flows through the memory hierarchy. The example tuple size is 12 bytes. The GTX 970 GPU uses a memory bus width of 256 bits or 32 bytes. Given that the size of a L2 cache line is also 32 bytes, each memory access can be supported by L2 in full. Overall, in a L2 cache line, two tuples and the first two fields of a third tuple fit. If the missing field from the third tuple is requested, there is a cache miss. Thus a new memory access must be made. To avoid this issue, data alignment is necessary. To this end, by adding 4 more bytes to the tuple as a dummy padding field, as shown in Listing 3.4, the cache miss issue can be addressed. As a result, the new tuple size is 16 bytes and exactly two tuples fit in a single cache line at the expense of some extra preprocessing to add the new field. By contrast, the use of the SoA data layout does not require any preprocessing.

```
typedef struct {
    int id;
    int a;
    int x;
    char padding[4];
} Tuple;
```

Listing 3.4: Tuple representation with padding (AoS)

3.3.4 Evaluation

For the evaluation of the implementations, several sizes for the input relations with different selectivities are used. Selectivity is formally defined as $sel = \frac{|R \bowtie_\theta S|}{|R||S|}$. The selectivities considered are (i) low with $sel = 0.1$, (ii) medium with $sel = 0.5$ and (iii) high with $sel = 0.9$.

In Table 3.2, kernel execution times for datasets with medium selectivity are presented. Each measurement is the average of ten executions. The first two columns refer to the kernels that implement the respective approach, where a 2D grid of 2D blocks is launched. The next three

Size	Naive	Partial Sums	Shared Memory			CPU		PostgreSQL
			AoS	AoS+padding	SoA	6 threads	12 threads	
500×50	0.02	0.021	0.012	0.012	0.011	**less than 0.001**		3
500×250	0.071	0.052	0.016	0.015	0.015	**less than 0.001**		25
500×450	0.121	0.087	0.018	0.017	0.018	**less than 0.001**		22
5K×500	1.271	1.056	0.086	0.084	**0.081(18.1X)**	1.467	1.847	260
5K×2.5K	6.303	5.1	0.293	**0.274(38.2X)**	0.277	12.804	10.472	1140
5K×4.5K	11.348	9.196	0.501	**0.465(46.6X)**	0.474	32.853	21.662	1950
50K×5K	108.815	94.018	4.797	**4.468(43.9X)**	4.504	301.681	196.268	20480
50K×25K	533.027	451.813	23.118	**21.318(49.3X)**	21.702	1804.519	1051.98	102360
50K×45K	957.757	809.986	41.433	**38.177(49.4X)**	38.838	2998.599	1884.701	183340
500K×50K	10610.993	9015	393.217	**363.63(57.6X)**	368.983	33445.87	20938.164	too high
500K×250K	53038.207	45101.3	2395.655	2563.121	**2340.031(45.9X)**	162775.844	107377.409	too high
500K×450K	95456.57	81126.835	4310.646	4576.681	**4341.435(43.8X)**	290216.809	190328.897	too high

Table 3.2: Execution times for the query in Listing 3.3 with medium selectivity (in ms). The winning approach is in bold and the speedup compared to the best performing CPU time is inside the parentheses.

Size	Low selectivity		Medium selectivity		High selectivity	
	PS	SM-SoA	PS	SM-SoA	PS	SM-SoA
50K×5K	25.715	4.477	94.018	4.504	161.001	4.54
50K×25K	113.606	21.524	451.813	21.702	792.342	21.819
50K×45K	199.718	38.582	809.986	38.838	1423.813	39.092
500K×50K	2284.643	366.457	9015	368.983	15828.143	373.08
500K×250K	11430.76	2098.067	45101.3	2340.031	79131.898	2693.466
500K×450K	20566.394	4019.814	81126.835	4341.435	142412.156	5042.282

Table 3.3: Testing with different selectivities (PS: Partial Sums, SM-SoA: Shared Memory(SoA)).

Algorithm 3.3 Stand-alone CPU implementation using p threads.

```
1: split R in p partitions
2: Work of each thread:
3: for all tuples in the corresponding R partition do
4:     for all tuples in S do
5:         evaluate join condition
6:     end for
7: end for
```

columns refer to the kernels that implement the shared memory approach, where a 1D grid of 2D blocks is launched. In all the cases, the blocks are of size 32×32 in order to maximize the achieved occupancy. The rightmost set of three columns provides the host time regarding (i) a stand-alone C implementation using an Intel i7 5820k 3.3GHz CPU with, 16 GB DDR4 RAM at 2400MHz and 6 and 12 threads, and (ii) the corresponding times in PostgreSQL in line with the evaluation rationale in [48], which also uses the performance of PostgreSQL as a baseline. The specific type of CPU has 6 cores supporting 12 threads due to hyper-threading; as such, the penultimate column corresponds to full CPU utilization. The stand-alone implementation is

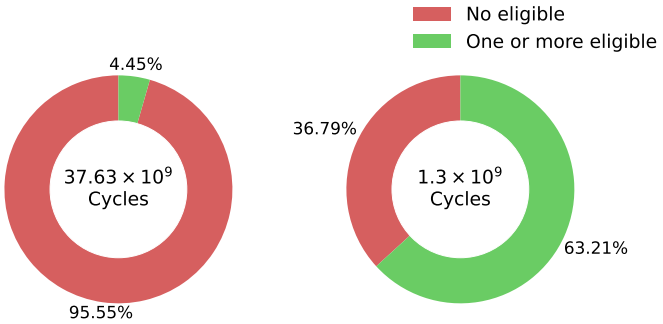


Figure 3.3: Warp issue efficiency: naive (left) vs SM-SoA (right).

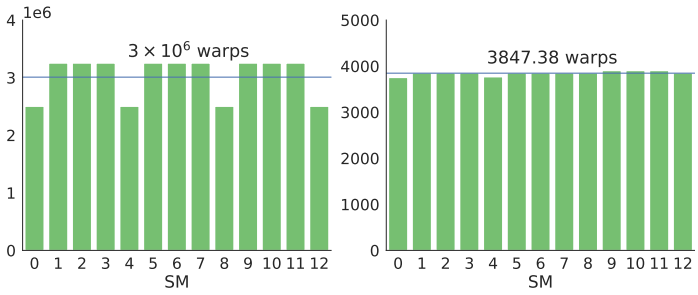


Figure 3.4: Warps launched: naive (left) vs SM-SoA (right).

free of any overheads that are involved through running a complete query processing engine; this advantage comes at the expense of not benefiting from all the sophisticated caching and indexing mechanisms that PostreSQL provides. It adheres to the rationale in Algorithm 3.3, where all data is in main memory, R is split into as many partitions as the number of threads, and each thread iterates over the complete S relation for each R tuple.

In Table 3.2, the winning approaches are in bold, and their speedup compared to the best performing CPU time is inside the parentheses. The execution times in the table show that exploiting the shared memory yields improvements of an order of magnitude (up to more than 20 times faster execution) over the naive implementations. By contrast, simply mitigating the impact of atomic operations yields improvements of only 10-20% lower times. Also, the comparison against the CPU times proves that allocating theta-join processing on GPUs is beneficial yielding lower times by two orders of magnitude when compared to the C stand-alone implementation unless the datasets are very small. The improvement is even more significant when compared against a state-of-the-art database management system, such as PostgreSQL, due to the inherent overheads from running a complete execution engine. Finally, there is no clear winner between the SoA and AoS+padding data layouts; however, the former is more efficient for larger datasets, and for the datasets that is not optimal, it is very close to the optimal.

To investigate the reason for the differences in the GPU execution times, the naive implementation and the shared memory one using the SoA data layout for the dataset of size 50k $\times$ 25k with medium selectivity are selected for profiling. Figures 3.5-3.6 show lower-level metrics that explain why the shared memory approach is more efficient. On the left side of each figure, the profiling of

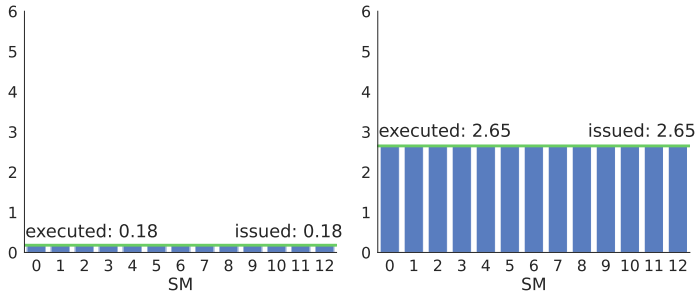


Figure 3.5: Instructions per cycle: naive (left) vs SM-SoA (right).

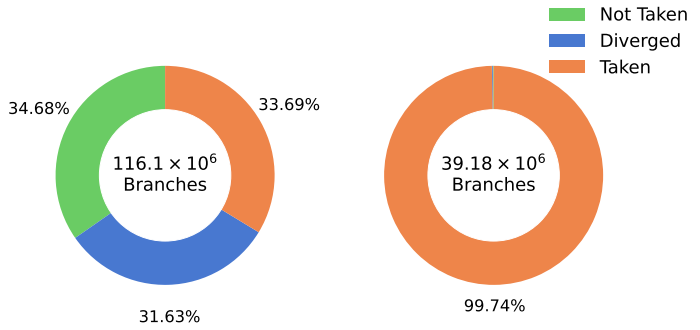


Figure 3.6: Branch condition: naive (left) vs SM-SoA (right).

the naive implementation demonstrated and on the right that of the shared one. Both implementations have 100% theoretical occupancy. However, the naive one achieves only 67.8%, while the shared-memory based one achieves 99.76%. The main reason for this difference lies in the warp execution efficiency as shown in Figure 3.3. Despite the fact that in the shared-memory approach launch 800× less warps are launched, as shown in Figure 3.4, a much higher proportion of them are eligible for execution per cycle and better load balance between the SMs is achieved. This leads to more instructions per cycle, as shown in Figure 3.5. Branching is another issue that is tackled more efficiently in the shared-memory approach, where most threads have the same execution flow thus reducing divergence, as shown in Figure 3.6.

Another aspect of the evaluation deals with the approaches' efficiency on datasets of the same size but with different selectivities. As shown in Table 3.3, the kernels which implement the naive approach are more sensitive to the selectivity value. On the contrary, the execution times of the optimized kernels that leverage the shared memory are less affected by changes in the join selectivity.

Finally, as shown in Figure 3.7 the problem is memory-bounded and supports the emphasis on efficient memory management. Through the shared memory approach, there is a better exploitation of the L1 and L2 caches, and thus a minimization of global device memory accesses. However, the global read/write efficiency reported by profiling remains relatively low. Employing other types of memory, such as texture memory, is not expected to yield tangible benefits. In general, further investigation is required to increase global memory efficiency.

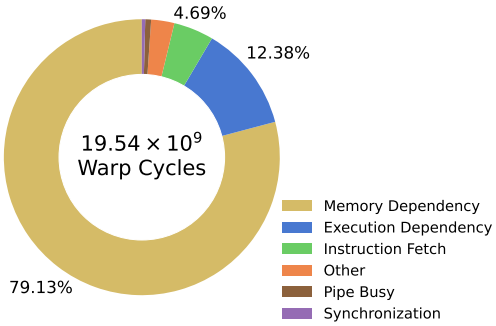


Figure 3.7: Memory dependency.

3.4 Output-Bound Theta-Joins

Having established a general view about efficient thread indexing and JM partitioning, the output-bound query is investigated next. The added complexity concerns the need to efficiently store output data. The query used includes the same theta condition, while output tuples consist of three fields, namely $R.a$, $S.a$, $S.x$, as shown in Listing 3.5.

```
SELECT R.a,
       S.a,
       S.x
FROM R, S
WHERE R.a > S.a;
```

Listing 3.5: General theta-join query

```
typedef struct {
        int Ra;
        int Sa;
        int Sx;
} Result;
```

Listing 3.6: Result tuple representation (AoS)

3.4.1 Naive Implementation

Similarly to the input-bound query, first a naive implementation is provided, where as many threads as the JM cells are allocated. Remember that, in each round, the host feeds the maximum amount of data so that their cartesian product can fit into the device memory. To store the result tuples, initially the AoS data layout is used as shown in Listing 3.6.

To reduce running times, it is important to manage to store output tuples in consecutive memory locations. For this to be achievable, and due to the fact that thread divergence existence is the norm, the usage of a global index counter is required. Every thread, in which the theta condition is satisfied, must store its corresponding output tuple to the address pointed by the index counter and increment the counter by one. To ensure correctness, the increment operation must be done using an atomic function. However, this leads to sequential thread execution, as already discussed for the atomic sum.

Note that in general, every input tuple is read multiple times, even from threads in the same block. The same S tuple is processed by all threads corresponding to the same row in the JM matrix. Similarly, R tuples are also reused. As such, storing the input records on the chip is beneficial. However, employing the shared memory to temporarily store input tuples plays a minor (if not negligible role) in this case. This is because 32×32 thread blocks are employed, which are

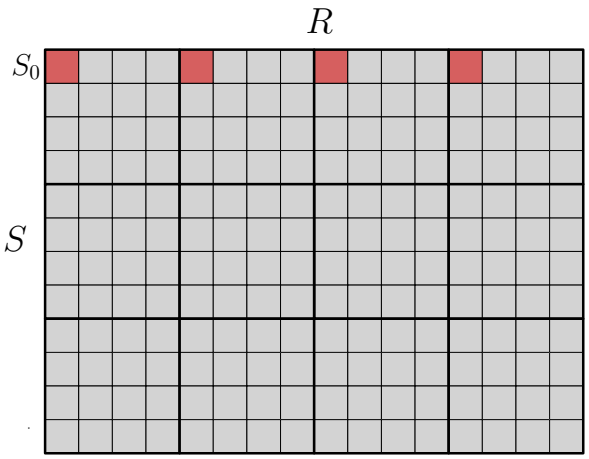


Figure 3.8: Illustration of same reads: tuple S_0 is read $|R|/B.x$ times in the worst case.

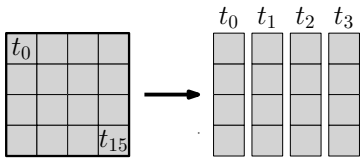


Figure 3.9: Assigning more work in each thread so that it processes a whole JM column (right) instead of a JM cell as in Figure 3.2 (left).

decomposed in 32 warps. In each warp, the same S tuple is processed by multiple threads but all of them belong to the same warp. Consequently, each S tuple is actually read once from global memory and then cached in L1 to be reused by all threads in the same block and in L2, which is shared among SMs. In the worst case, an S tuple is not in L2 when subsequent blocks request it, and it is read $\frac{|R|}{B.x}$ times. An example of such a case is shown in Figure 3.8, where the first tuple from S can be read up to four times from the global memory.

3.4.2 A more efficient approach

As shown in the input-bound query, the best implementation was observed when a 1D grid was launched. In the naive approach, the total number of threads is $T = \lceil \frac{|R'|}{B.x} \rceil \times \lceil \frac{|S'|}{B.y} \rceil \times B$, and each thread evaluates a single theta condition. In optimized approach, a 1D grid is launched, and the number of threads is $T = \lceil \frac{|R'|}{B.x} \rceil \times B$. The main difference from the technique in the previous section, which aimed at the computation of a partial aggregation, is that the thread block size is one dimensional as well. That is, $B = B.x$, which implies that each thread becomes responsible for evaluating $|S'|$ theta conditions (see Figure 3.9) and as such, the code in Algorithm 3.2 is modified to iterate over all S tuples, as shown in Algorithm 3.4.

At first sight, the execution of a much lower number of threads comes at the expense of decreased achieved occupancy. However, as shown in [94] and other works, it is possible to achieve

Algorithm 3.4 Work per thread when exploiting the shared memory for the output-bound case.

1: load $R.a$ value to a register
2: load first S tuple in shared memory
3: **repeat**
4: **if** θ condition is met **then**
5: output result (main option: use shared memory, see Section 5.3)
6: **end if**
7: load next S tuple in shared memory
8: **until** no more S tuples (overall p tuples are processed)

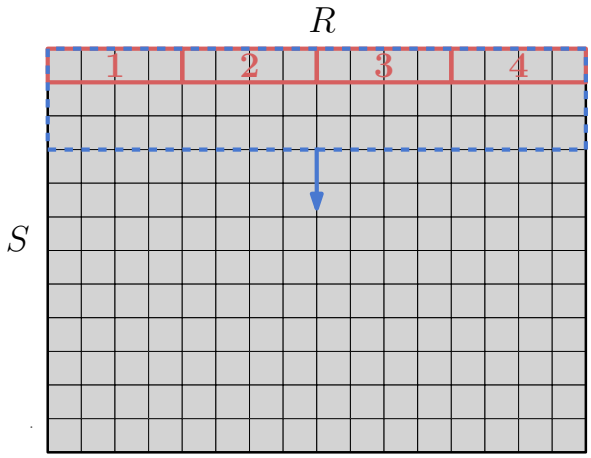


Figure 3.10: The output-bound approach to process the JM on a GPU.

better performance at lower occupancy. As already mentioned, a common misconception about CUDA programs is that, through only the execution of hundreds of thousands threads, the GPU can manage to hide its compute or memory access latency. However, exploitation of on-chip memories, efficient memory access and data reuse and sharing among threads are equally important.

Since each thread processes exactly one tuple from the R relation, it can hold it in its registers throughout the thread lifetime. Furthermore, shared memory can also be leveraged for holding the contents from S. In the aggregation query, the thread block dimensions were fixed to 32×32 and, as such, the S records processed by the same threads in a block were cached in L1 and L2. By rendering the block size 1D, it is very likely to have different warps requiring the same S tuples, and there is no guarantee that these are cached. Hence, the S records are explicitly transferred in the shared memory.

Due to the limited shared memory size, relation S is partially iterated over in chunks of size p; p is selected so that the chunks can fit in the shared memory. Further, since each thread retrieves at most one S tuple from global memory and stores it to the shared memory, the chunk size is directly linked with the block size B, and more formally $p \leq B$. Finally, to avoid further divergence as will shown below, the chunk size p needs to be a divisor of $|S|'$, i.e., $|S|' \; mod \; p = 0$.

To summarize, a 1D grid of $\lceil \frac{|R'|}{B} \rceil$ blocks is launched. As shown in Figure 3.10, the red rectangles represent the blocks launched while the blue rectangle the portion of S relation read. Upon the kernel launch, $R.a$ values are copied to the registers of the corresponding threads. Then, an iterative process begins (the blue rectangle in the figure). In each iteration, p tuples from S are read, the participating fields of which, i.e., $S.a$ and $S.x$, are stored in the shared memory, so that they are reused by all threads in the same block. Next, through a second nested iterative process, each thread evaluates a sequence of p theta conditions and stores the corresponding output tuples. The output storage is discussed in the next section. The last operation is to increment the counter, which indicates the position for the next chunk. These three operations are repeated until the last portion of S is accessed. In between iterations, threads are synchronized at the block level with a view to ensuring correctness (and indirectly maintain the degree of parallelism). This technique can be also regarded as equivalent to applying a sliding window on the JM.

A few additional implementation details are as follows. If B is a divisor of $|R'|$ and multiple of 32, which is the warp size, it is ensured that only $|R'|$ threads will be executed. This is the ideal case. For example, in a different case where $B = 32$ and $|R'| = 15000$, 14976 threads in 468 warps will be executed, where all threads are active. To cover the remaining R tuples, an additional warp, where only 24 threads are active, can be scheduled. Another issue that requires some care is not to allocate the reading of an S tuple to an inactive thread. Although this is required only in the warp processing the rightmost portion of the R relation, it involves the usage of an extra condition, which increases the requirements of registers per thread. This in turn may lead to further restrictions on the amount of threads that can launched in a single thread block and calls for careful design of the kernel to avoid register spilling.

3.4.3 Output writing issues

In the presented implementations, it is assumed that the result of the join is an array of size $|R'| \times |S'|$. With the use of atomic functions, it is ensured that if X tuples are in the output, these will be stored in the first X addresses of the result array. This storage method is necessary because, when the execution is finished, it is more likely that less memory is used than the amount allocated. Therefore, if the result is not written in consecutive memory addresses, then either a post-processing phase is required or more data will be transferred to the host memory through the slow PCI-E link. However, if results are stored consecutively, then there is no need for post-processing.

Another option is to use the shared memory for temporarily storing the results before writing them into the global memory. For this to be applicable, the corresponding amount of shared memory must be free, and this depends on the size of output tuples. In the naive approach, where a 2D grid of 2D blocks is launched, the space needed is defined by the block dimensions. For example, if the block size is 32×32 and the output tuple size is equal to 12 bytes, then the space needed would be 1024×12=12288 bytes. On the other hand, in the second approach, B and p define the space needed per iteration. If B=256 and p=25 then 256×25×12=76800 bytes of shared memory would needed , which is already partially occupied by p tuples from S. In general, through the usage of shared memory, the storage problem is alleviated which in turn, contributes to accelerating the whole execution.

However, the issue of writing tuples consecutively from the shared memory of SMs to the global memory still exists. To solve it, there are two options, either (i) use a shared atomic function or (ii) resort to other techniques similar to prefix sum. Since a block's execution is independent from the execution of other blocks, grid level synchronization is required to ensure that no output tuple overwriting occurs. In the experimental evaluation, option (i) behaved better. Although the second option sounds reasonable, in the experiments, it has not yielded any benefits. More specifically, for option (ii), a 2-pass approach was investigated, where in the first pass, it is computed where each thread should write and then, in the second pass, the actual result writing takes place[4]. However, this resulted in higher execution times by several factors up to an order of magnitude. The performance degradation is partially attributed to the fact that the pointers for each thread are stored in the global memory and the amortized cost of accessing the global memory is higher than an atomic operation.

Finally, compression was investigated, i.e., instead of forcing writes to successive memory addresses, results should be written to predefined locations. Since a result is not expected from every pair of tuples, compression can be used to reduce the amount transferred to the host [5]. As shown in Section 3.4.4, no improvements could be achieved. Another implication of such an approach is that the device memory must be split into two parts: one for the initial results and one for the compressed results.

In summary, efficient output writing remains an open issue. Promising directions include a closer collaboration between CPUs and GPUs in theta-join processing. An early idea is the result of each tuple pair evaluation on the GPU side to be a bit indicating whether the theta predicate evaluates to true or false, and the actual result tuple construction to take place on the CPU. In such an approach, the challenge is not to transfer the bottleneck to the CPU. However, in the proposed solution, focus falls on techniques where the complete processing takes place on the GPU; co-operation between the two types of processors in producing the final results is left for future work. Finally, advances, such as NVLink demonstrated [60], may alleviate any current bottlenecks regarding transfer times, and render GPU-side join processing even more attractive.

3.4.4 Evaluation

For the evaluation of the output bound query, each kernel time is the average over ten independent executions. Two kernels are examined, one for each approach in Sections 3.4.1 and 3.4.2, respectively, with datasets of medium and high selectivity. In Table 3.4, the times for the naive approach, distinguished between execution and transfer times, are presented. Table 3.5 presents the CPU times using the stand-alone implementation (also presenting single-thread executions). In Table 3.6, only the execution times for the shared memory-based approach are presented, for a range of p and B values. The transfer times through the PCI-E bus are in the same range as in Table 3.4 and are omitted. The presented shared memory-based approach employs the AoS data layout with tuple padding. The AoS data layout is preferred to store output tuples, so that threads write in consecutive memory addresses. The experiments with the other data layout modifications

[4] To implement this approach the `exclusiveScan` operation from the CUB library (`https://nvlabs.github.io/cub/`) is employed to access the results of the 1st pass and avoided any `atomic add` operations

[5] The `thrust` library is used in the implementation.

Size	Medium selectivity			High selectivity		
	Execution	Transfer	Output	Execution	Transfer	Output
50K×5K	30.238	645.909	1.4GB	47.487	935.876	2.5GB
30K×30K	139.914	2042.585	5.03GB	199.796	3653.337	9.05GB
50K×25K	343.761	2736.725	6.98GB	441.718	5835.067	12.57GB

Table 3.4: Naive approach times (in ms).

Size	Medium selectivity			High selectivity		
	1 thread	6 threads	12 threads	1 thread	6 threads	12 threads
50K×5K	3151	611.81	490.096	3515	698.174	1009.288
30K×30K	12847	2304.92	1876.202	15048	3177.28	3315.225
50K×25K	17718	3335.919	2811.114	20922	4659.629	5386.257

Table 3.5: CPU times (in ms).

p	50K×5K			30K×30K			50K×25K		
	B=256	B=512	B=1024	B=256	B=512	B=1024	B=256	B=512	B=1024
10	**20.791(23.6X)**	21.207	22.733	86.955	97.871	122.501	131.778	148.627	186.026
25	21.095	21.404	22.628	84.423	97.431	122.061	**126.014(22.3X)**	147.693	183.195
100	21.597	21.264	22.803	84.532	96.398	120.676	128.073	145.531	183.680
250	21.298	21.559	22.619	**84.289(22.3X)**	95.627	121.526	128.072	144.350	183.740
500	-	21.388	22.567	-	97.499	121.899	-	144.407	183.055
625	-	-	22.488	-	-	121.931	-	-	185.052

Table 3.6: Shared memory-based (AoS plus padding) execution times with medium selectivity (in ms). The best performing configuration is in bold and the speedup compared to the best performing CPU time without considering transfer times is inside the parentheses.

that were presented do not exhibit significant differences in execution times, as also shown in the evaluation of the input-bound query in Section 3.3.4, and thus are omitted.

As previously, to maximize the achieved occupancy, in the naive approach a 2D grid of 2D blocks of size 32×32 is launched. In the shared approach, a 1D grid of 1D blocks of size $B = B.x$ is launched, with the corresponding p to iterate the S relation. To avoid further divergence while iterating the last portion, p should be a divisor of $|S'|$. However, in real case scenarios and depending on the JM partitioning, this may not be feasible. The presented implementations are not tailored to a specific data input size with a cost of a small amount of divergence, which is negligible thanks to automated compiler optimizations.

From the results in Tables 3.4, 3.5 and 3.6, three main conclusions can be drawn. First, the improvements on the naive approach are significant but of lower magnitude than in the previous query. For example, the most efficient shared-memory approach runs up to 2.7 times faster than the naive implementation and 23.6 times faster than a 12-thread CPU implementation, whereas for the input-bound query higher speedups have been observed. This is due to two factors: (i) there are too frequent global memory transactions, and (ii) the use of atomic functions to store results consecutively has a negative impact on performance. Second, the performance is higher for relatively low values of p and B, e.g., 25 and 256, respectively. Third, the transfer time seems to

Size	Medium selectivity			High selectivity		
	Execution	Compression	Transfer	Execution	Compression	Transfer
50K×5K	30.321	76.313	585.583	45.733	96.401	1013.376
30K×30K	216.083	274.934	2117.466	271.012	347.681	3754.746
50K×25K	422.425	376.478	2979.389	496.492	479.365	5302.361

Table 3.7: Times when enabling compression (in ms).

dominate. However, as reported in [66] and demonstrated in [60], in the close future, the transfer times are expected to drop by an order of magnitude due to the NVLink technology, which implies that they will become similar to, if not lower than the execution times. In the experiments, the transfer overhead is outweighed by the benefits from the parallel execution on the GPU for certain configurations, such as 6-threads on a CPU and medium selectivity.

The time required for writing the output on the output on the GPU without performing any meaningful processing is also measured. For example, for the 50K×5K times dataset with $sel = 0.5$, writing 125 million tuples takes 20.04 ms, which means that the shared-memory approach manages to drop the execution times close to their minimum possible.

Finally, compressing the output is also investigated. This is done by repeating the experiments of the naive approach as shown in Table 3.4 with compression enabled. The compression-based times are presented in Table 3.7. By comparing the two tables, it is observed that compression performs worse, i.e., the execution time is higher, there is a compression overhead and no benefits from reduced transfer time. This is due to the fact that the compression-based approach suffers from increased conflicts due to concurrent accesses to main memory and does not benefit from shared memory. On the other hand, the naive approach already produces the results in a compacted form, in the sense that results are written in a coalesced manner.

3.5 Related Work

As already mentioned before, existing GPU proposals for relational query operators focus on sorts, selects, aggregates and hash- and sort-based equi-joins or the efficient handling of data transfer between the host and the device.

Many research works study the efficient memory coalescing and splitting of the workload into chunks. In [35], a set of primitives for join processing implemented on a GPU are presented. Their behavior on modern GPUs is also discussed in [82]. The work in [35] contains a chunk-based implementation of nested loops that, in principle, can process theta-joins; however, no implementation details, e.g., shared memory usage, are provided in order their proposal to be comparable against the presented one. In other words, this thesis proposed solution shares the same high-level approach and its contribution can be deemed as a detailed investigation of the corresponding implementation issues.

In [4], the relations are split into chunks with the use of a tailored data structure that allows for efficient data transfer and usage on both sides. In [88], data rearrangement through clone creation is proposed to reduce overall memory contention and race conditions between executed threads.

The proposed solution addresses the same problems, but tailored to the specific and challenging case of theta-joins. Using the shared memory in equi-joins on GPUs is also discussed in [74].

Also, due to the limited size of the GPU memory, there is a need to exploit certain technologies, which allow the device to use host memory. The study presented in [47] investigates the use of the Unified Virtual Addressing (UVA). UVA provides a single virtual memory address space for all memory types in the system, thus enabling pointers to be accessed from GPU no matter where in the system they reside.

Another problem reported in the literature relates to thread divergence. In contrast to a typical CPU, a GPU devotes a larger amount of transistors to data processing compared to data caching and flow control. This means that any kind of divergence during parallel execution incurs a performance penalty. Thread divergence is common in join queries. In [89], there is a study of conjunctive selection queries, but the proposed technique cannot generalize to theta-joins. Additional works on GPU queries include topics, such as concurrent queries [96] and portable development [112]. These issues are orthogonal to the presented solution. Also, specific forms of joins other than equi-joins, e.g., continuous intersection joins of moving objects [102], have been investigated.

With a view to exploiting both CPUs and GPUs, the study in [75] presents a relational query processing strategy, which first calculates quickly and approximately the query results based on compressed data within the device memory and produces the final results on the host side. Another example of closer collaboration between CPU and GPU regarding equi-joins has appeared in [36]. Investigation of techniques where the CPU and GPU cooperate more closely to process a theta-join is an interesting direction for future work, in line with the recent trend in GPGPU [66].

Regarding theta-joins in MapReduce, apart from the work in [70], [27] provides an adaptive solution, whereas [48] explores techniques for more efficient predicate evaluation when there are two predicates. [50] deals with the issue of preprocessing the JM, when selectivity information is known a-priori. [93] improves on [70] for a specific form of JMs, termed as monotonic. In [8, 17, 113], the authors investigate the case where multiple relations are joined in a single step. None of these proposals contains techniques that can be transferred to the GPGPU setting. Finally, theta-joins are also related to similarity joins, e.g. [85]. However, the main effort in similarity joins in parallel settings is to prune non-relevant pairs as soon as possible, which corresponds to eliminating JM candidate cells rather than devising techniques to efficiently process all JM cells as in generic theta-joins. Similarity joins in the GPGPU setting are investigated in Chapters 4 and 5.

3.6 Remarks

The proposed solution deals with theta-joins on GPUs, an issue that has not been explored in depth to date. It considers two cases. In the first one, the theta-join is followed by an aggregate, while in the second, the complete output, which may be very large, needs to be stored. Moreover, this work emphasizes on issues such as exploitation of on-chip memories, data reuse, reduced accesses to the slow global memory, high achieved occupancy, and efficient data layout. It thoroughly discusses the implementation issues involved and proposes specific optimizations, which

yield performance improvements up to an order of magnitude over naive implementations in the first case, and up to 2.7 times in the second case.

A main lesson learned through this activity is that paying attention to the details is of high importance and several factors are significant in an efficient implementation; these factors do not merely relate to a high number of threads to hide latencies, but include all the types of issues mentioned above. Moreover, the relative importance of each factor changes from one case study to another, which calls for specialized solutions for each case. Finally, the most challenging remaining issues is how to efficiently write huge amounts of data to the device memory as a result of data processing and transfer them to the host, and how CPUs and GPUs can co-operate more closely.

4 Accelerating Set Similarity Join

Given two collections of sets and a threshold, set similarity join is the operation of computing all pairs the overlap of which exceeds the given threshold. Because of its generality, set similarity join is used in a wide range of application domains including data mining [90], data cleaning [15] and entity resolution [3].

In very large datasets, finding similar sets is not trivial. Due to the inherent quadratic complexity, a set similarity join between even medium sized datasets can take hours to complete on a single machine[1]. In addition, challenges like high dimensionality, sparsity, unknown data distribution and expensive evaluation arise. To tackle scalability challenges, two main and complementary approaches have been followed. Firstly, to devise sophisticated techniques, which safely prune pairs that cannot meet the threshold as early as possible, typically through simple computations related to the prefix and the suffix of the ordered sets, e.g. [45, 62]. Secondly, to benefit from massive parallelism offered by the MapReduce framework, e.g., [5, 64, 85, 92] or GPUs.

The problem of exact set similarity on GPUs has been investigated in [77, 80, 81, 84]. However, these solutions delegate the complete workload in the GPU, leaving the CPU idle in the join process. In addition, two of them are unable to scale (as detailed in Chapter 5 in which a complete evaluation between existing GPGPU techniques is presented). Therefore, there is a gap in detailed investigation of exact similarity joins in respect to the GPGPU paradigm; utilizing both the CPU and the GPU to solve the problem. The main contribution of the proposed GPGPU technique is to fill this gap and propose efficient solutions after thoroughly investigating several design alternatives.

The proposed technique incorporates a co-process scheme between CPU and GPU in order to efficiently compute set similarity join. In the presented scheme, CPU remains responsible for index building and initial pruning of candidate pairs, whereas the GPU computes the overlap of all remaining pairs. This however leads to limited maximum speedups because of the Amdahl's law. Moreover, the GPU part comes with several challenges regarding the data serialization and layout, the thread management and the techniques to compare sets of set elements.

The presented technique addresses all the challenges and manages to achieve speedups up to 2.6X; moreover, it is shown that the proposed co-processing scheme has reached its maximum potential in the sense that it annihilates the impact of GPU tasks on the running time. In summary, the technical contributions are three-fold. First, a detailed implementation analysis is provided along with three alternatives that differ in the workload allocated to each GPU thread. Second, an extensive performance analysis is conducted on seven real world datasets. The findings are compared to the state-of-art CPU implementations and point out a number of optimizations to further increase the performance. Finally, evidence is provided that in settings where the candi-

[1] E.g., a similarity join over the DBLP dataset, which consists of 6.1 millions distinct sets, using a similarity threshold of 0.85 takes 8.5 hours approximately

Description	Definition
Equivalent overlap (τ)	$\lceil \frac{\tau_n}{1+t_n}(\lvert r\rvert + \lvert s\rvert) \rceil$
Size lower bound (lb_r)	$\tau_n \lvert r\rvert$
Size upper bound (ub_r)	$\frac{\lvert r\rvert}{\tau_n}$

Table 4.1: Normalized threshold τ_n translation size bounds for the $Jaccard(r,s) = \frac{|r \cap s|}{|r \cup s|}$ function (adapted from [62]).

r_1	$\{e_{13}, e_{15}\}$
r_2	$\{e_4, e_7, e_{13}, e_{14}, e_{16}\}$
r_3	$\{e_6, e_9, e_{12}, e_{15}, e_{16}\}$
r_4	$\{e_9, e_{11}, e_{14}, e_{15}, e_{16}\}$
r_5	$\{e_{10}, e_{13}, e_{14}, e_{15}, e_{16}\}$
r_6	$\{e_8, e_{10}, e_{11}, e_{12}, e_{14}, e_{15}, e_{16}\}$
r_7	$\{e_5, e_8, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_{16}\}$
r_8	$\{e_1, e_2, e_7, e_8, e_{11}, e_{13}, e_{14}, e_{15}, e_{16}\}$
r_9	$\{e_3, e_5, e_9, e_{10}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}\}$
r_{10}	$\{e_6, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{14}, e_{15}, e_{16}\}$

Figure 4.1: An example collection of records R.

date pairs are tens of billions or more, the proposed solutions reach their maximum potential, since they manage to fully hide the impact of GPU tasks on the running time due to overlapped execution with the CPU.

4.1 Background

In this section, a formal definition for the set similarity join problem is given. In addition, the filter-verification framework used by state-of-the-art main memory set similarity join algorithms is presented, in line with the comparison work conducted by Mann et al [62].

4.1.1 Problem Definition

Given collections R, S and a normalized threshold value $\tau_n \in [0,1]$, set similarity join is the operation of computing:

$$R \bowtie S = \{(r,s) \in R \times S | sim(r,s) \geq \tau_n\}$$

where $sim(\cdot,\cdot)$ corresponds to a function used to calculate the similarity degree between each (r,s) set pair. Sets consist of elements from a finite universe $E = \{e_1, e_2, \ldots, e_m\}$ with m the number of distinct elements.

Similarity Functions. To measure the similarity between sets, normalized similarity functions such as Jaccard, Cosine and Dice are typically used. The given normalized threshold τ_n is translated to an equivalent overlap τ and thus, a pair (r,s) is considered similar only if $|r \cap s| \geq \tau$. In

addition, τ_n can be further used to denote the set size range for all possible candidates of a set r. Table 4.1 shows the τ_n translation and size bounds for the Jaccard similarity function.

Self Join. Most commonly, the set similarity join is investigated as a self-join using only a single collection of sets ($R = S$). However, non self-joins can be transformed to self-joins as discussed in [107].

Data Layout. Set elements are sorted by their frequency in increasing order, so that infrequent elements appear first in a set. The sets of a collection are sorted first by their size and then lexicographically within each block of sets of equal size. An example collection R consisting of ten sets is illustrated in Figure 4.1. In this figure, the sets are sorted by their size in ascending order. For instance, consider the set r_8, the first element of which is the most infrequent one in the complete dataset, i.e. e_1 and in contrast, the most frequent element, i.e. e_{16}, is at the end. This data layout is preferred in order to enable effective filtering, as explained next.

4.1.2 Set Similarity Joins in CPU

The state-of-the-art main memory set-similarity algorithms conform to a filter-verification framework, as explained in [62], in which seven key representatives[2] are compared using real world datasets. The common idea behind all these algorithms is (i) to avoid comparing all possible set pairs by applying filtering techniques on preprocessed data to prune as much candidate pairs as possible; and (ii) then to proceed to the actual verification of the remaining candidates. The majority of existing solutions focus on how the filtering cost could be decreased. As a result, several filters have been proposed.

Filters

There are two basic filters that rely on the existence of an index-like data structure, namely, the prefix and the partition filter. Both have the highest filtering potential among all filters found in literature. Based on which filter they use, existing algorithms can be categorized into (i) prefix-based, and (ii) partition-based.

Apart from basic filters, there are other simpler filters that exploit the given threshold value and set size. Nevertheless, more sophisticated filters exist, such as the one presented in [84]. A concise description for each of the examined filters is given below.

Prefix filter. The first applied filter, called prefix-filter, examines only two subsets called prefixes, one from each sorted set in the candidate pair, and discards the pair if there is no overlap between the prefixes. More specifically, a prefix of a set r_i, denoted as $pre_{\pi_i}(r_i)$ is formed by the $\pi_i = |r_i| - \tau + l$ first tokens of the set, where l is the required overlap. Thus (r_i, r_j) is considered a candidate pair if

$$|pre_{\pi_i}(r_i) \cap pre_{\pi_j}(r_j)| \geq l$$

In Figure 4.2(a), for $\tau_n = 0.8 \rightarrow \tau = 5$ and $l = 1$, there is no overlap between the respective set prefixes, thus, even if there is an overlap on the remaining tokens, any overlap threshold set to 5 or higher cannot be reached, and in such cases, the candidate pair can be safely pruned. Since sets in R are processed in increasing size order, no candidate set r_j is longer than the current probing

[2] AllPairs [7], PPJoin and PPJoin+ [108], MPJoin [79], MPJoin-PEL [61], AdaptJoin [95] and GroupJoin [12].

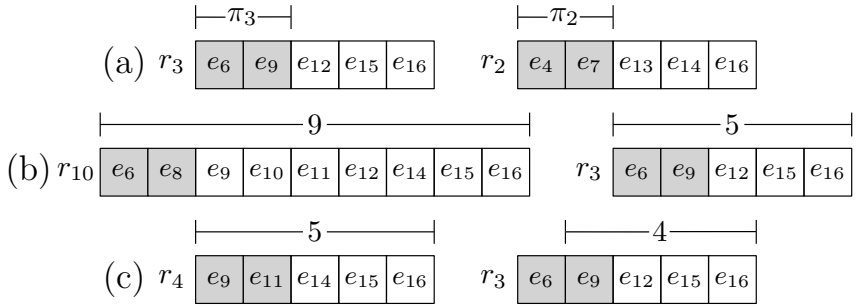


Figure 4.2: Example of the most established filters: (a) prefix, (b) length, (c) positional.

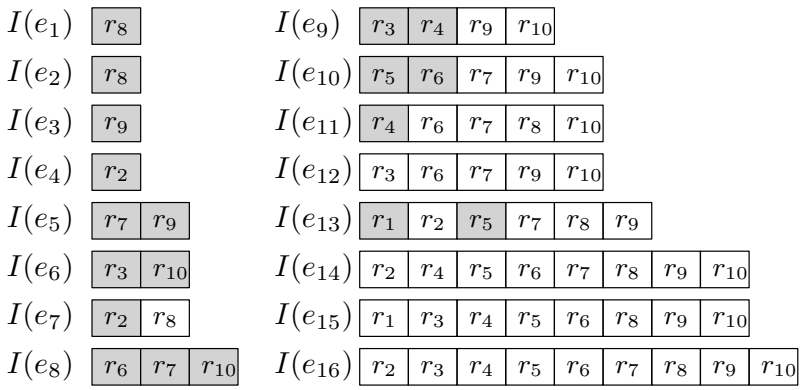


Figure 4.3: The complete inverted index for collection R. The materialized index is highlighted in gray cells ($\tau_n = 0.8, l = 1$).

set r_i. This enables the use of shorter prefixes of size $\pi_i = |r_i| - \lceil lb_{r_i} \rceil + l$ for indexing which consequently results in a smaller inverted index. An example of such inverted index can be seen in Figure 4.3.

Length filter. Another filter, known as length filter, takes advantage of the normalized similarity functions dependency on set size. Hence, a set r_j is considered a candidate pair of r_i if

$$lb_{r_i} \leq |r_j| \leq ub_{r_i}$$

In Figure 4.2(b), if $\tau_n = 0.8$, the shown candidate pair (r_{10}, r_3) is pruned by length filter despite the prefix overlap because set r_{10} requires a candidate set r_j of size $8 \leq |r_j| \leq 11$.

Positional filter. Given the first match position for an element e, denoted as pos_e, positional filter evaluates if a candidate pair can ultimately reach the required overlap. Thus (r_i, r_j) is a candidate pair if

$$|r_j| - pos_e(r_j) + 1 \geq \tau$$

As an example, in Figure 4.2(c), the first match position is $pos_{e_9}(r_3) = 2$, and thus for $\tau_n = 0.8 \rightarrow \tau = 5$, the pair (r_4, r_3) is pruned since the remaining tokens from set r_3 are not enough to reach the required overlap.

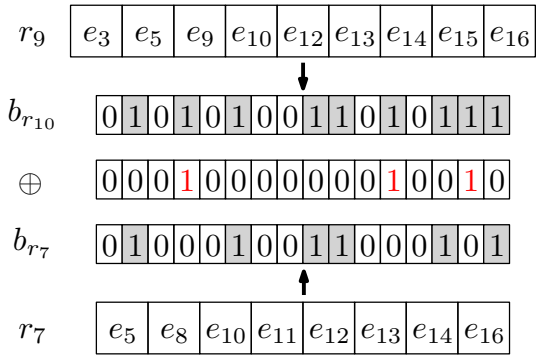


Figure 4.4: Candidate pair (r_9, r_7) can be safely pruned for $\tau_n = 0.8 \rightarrow \tau = 8$ since its expected overlap upper bound is 7 (Adapted from [84]).

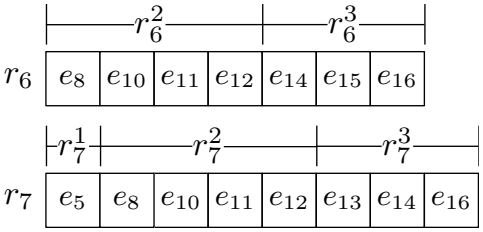


Figure 4.5: An partition filter example on candidate pair (r_7, r_6).

Bitmap filter. The authors of [84] propose a new low overhead filtering technique called bitmap filter. Essentially, the bitmap filter uses hash functions to create signature bitmaps of size b for the input collection sets. Thus, the initial collection elements are mapped in a fixed bitmap space. Without compromising the exactness of set similarity join, bitmap filter can deduce an overlap upper bound for a candidate pair. If the upper bound is less than the minimum required overlap the candidate can be safely pruned. More formally, (r_i, r_j) is considered a candidate pair if

$$\lfloor \frac{|r_i| + |r_j| - popcount(b_{r_i} \oplus b_{r_j})}{2} \rfloor \geq \tau$$

with b_{r_i},b_{r_j} the bitmap signatures of r_i,r_j respectively, and $popcount$ the operation of counting the hamming distance of the bitmap signatures. Figure 4.4 illustrates the application of bitmap filter on the (r_9, r_7) candidate pair.

Partition filter. In [22], the authors introduce a different type of index-based filter, noted as partition filter. Sets are partitioned into disjoint subsets based on the element universe E partitioning. Two sets are considered similar only if they share a common subset. In the first step, each set r_i is partitioned into $p_i = \lfloor \frac{1-\tau_n}{\tau_n} |r_i| \rfloor + 1$ subsets which are stored into an inverted index. In the second step, every set r_j is partitioned into p_i subsets, for any $lb_{r_i} \leq |r_j| \leq |r_i|$. Thus, by probing all the subsets of r_i in the inverted index it is easy to retrieve every (r_i, r_j) candidate pair. In Figure 4.5, for $\tau_n = 0.75$, probe set r_7 is partitioned into 3 subsets, $r_7^1 = \{e_5\}$, $r_7^2 = \{e_8, e_{10}, e_{11}, e_{12}\}$ and $r_7^3 = \{e_{13}, e_{14}, e_{16}\}$. By applying the same partitioning scheme to

candidate set r_6 it can be seen that $r_7^2 = r_6^2$. Therefore (r_7, r_6) is considered a candidate pair and undergoes a full verification.

Algorithm Outline

Algorithm 4.1 Filter-Verification Framework.

Input: A collection of sets R, a threshold τ_n
Output: All similar $pairs$ with $sim(r_i, r_j) \geq \tau_n$

```
1:  I ⟵ index(R)
2:  for each set r_i ∈ R do
3:      C ⟵ {}
4:      for each element e ∈ pre_{π_i}(r_i) do
5:          if (r_i, r_j) ∉ C and lb_{r_i} ≤ |r_j| ≤ ub_{r_i} and |r_j| − pos_e(r_j) + 1 ≥ τ then
6:              C ⟵ C ∪ {(r_i, r_j)}
7:          end if
8:      end for
9:      for each pair (r_i, r_j) ∈ C do
10:         if |r_i ∩ r_j| ≥ τ then
11:             output(r_i, r_j)
12:         end if
13:     end for
14: end for
```

A high level overview of the filter-verification framework is provided in Algorithm 4.1. Essentially, after the initial indexing of an input collection of sets R (Algorithm 4.1, line 1), a nested loop join consisting of two steps is executed. In the first step, noted as *filtering* phase, an index lookup and filters application per set of pairs are conducted (Algorithm 4.1, lines 4-8). The pairs that pass all filters form the candidate set C. In the second and final step, also noted as *verification* in the literature, the similarity score for each of the remaining candidate pairs is computed and if it exceeds the threshold, the pair is added to the output result (Algorithm 4.1, lines 9-13). Existing solutions for the set similarity join in literature mostly differ in the filtering phase, i.e., in which filters they use. In Algorithm 4.1, the length and positional filters are applied (line 5).

Leveraging Set Relations

The majority of the techniques proposed for set similarity join examine each set independently. This results in an accumulated computational cost and possible work overlap. Motivated by this, Wang et al. [100] introduce two skipping techniques, on top of prefix-based methods, which leverage relations among sets and achieve a computational cost decrease through shared computations.

Moreover, in the first skipping technique, noted as *index-level skipping*, the inverted index is rearranged so that set elements indexed in the same inverted list are partitioned into different blocks. Each block consists of sets of the same size and its entries are sorted in non-decreasing order of the set element position on the corresponding sets. Thus, for a probe set, whenever an entry fails the

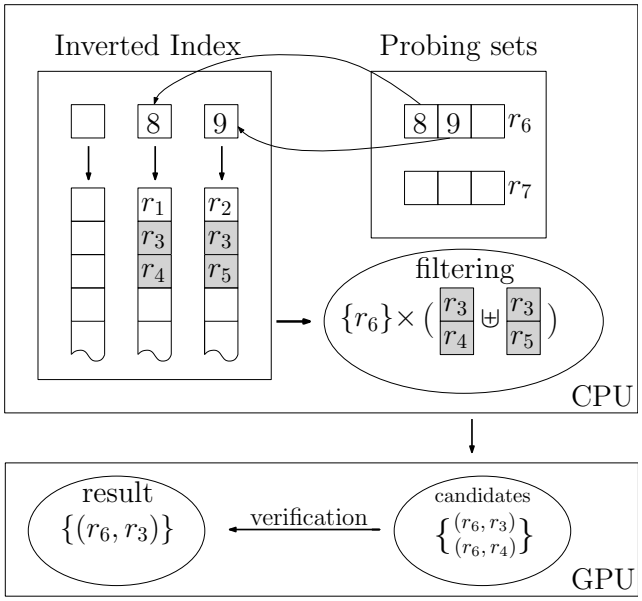


Figure 4.6: Splitting the workload between host (candidate generation) and device (candidate verification). The workload representation is adapted from [62].

position filter all the remaining unprobed entries in the same block are skipped. This also applies for the next sets to be evaluated. As a result, by exploiting index relations among sets, there is a significant reduction on the amount of redundant index probes.

The second skipping technique, noted as *answer-level skipping*, takes advantage of the possibility that two similar sets may also have similar answer-sets. An answer-set $A(r_i)$ is defined as the collection of similar sets for set r_i. For each similar set record r_j where $(r_i, r_j) \in A(r_i)$, using a cost estimation, it is determined if $A(r_j)$ will be computed from scratch or derived from $A(r_i)$. The latter leads to a complete skip of the r_j evaluation.

4.2 A Co-processing CPU-GPU Technique

Since the algorithms solving the set similarity problem efficiently conform to the filter-verification framework, splitting the workload between CPU and GPU involves specifying which component each phase is assigned to. In principle, GPUs are used to accelerate compute-intensive applications by processing data in a predefined memory space. In addition, based on the way that the CPU invokes the GPU, it is considered an anti-pattern if a GPU operates autonomously, e.g., self-exiting based on some condition, which may hurt the overall performance.

The proposed solution to efficiently compute set similarity join involves a co-process scheme between the CPU and the GPU. Due to the fact that for a probe set, an arbitrary number of candidate pairs may be generated, the filtering phase remains a CPU task. Thus, the CPU remains responsible for index building and initial pruning of candidate pairs. On the other hand, the verification phase is more suitable for parallelization, as it involves a merge-like loop, where the overlap of candidate pairs is computed. As soon as the overlap threshold is met or cannot be reached, the

operation terminates. True positives must be verified and the necessary overlap is still computed, while the rejection of pairs in this stage leads to less set element comparisons without sacrificing accuracy. Therefore, the verification phase is delegated to the GPU. A visual representation on how the workload is split on the proposed co-processing technique, is depicted in Figure 4.6.

Based on the empirical evaluation of the state-of-the-art CPU algorithms presented in [62], there are three key observations. First, all the evaluated algorithms have small performance differences except those which involve sophisticated filtering. Second, efficient verification yields significant performance speedups and renders complex filters inefficient. Finally, candidate generation is indicated as the main bottleneck especially for the techniques that employ the prefix filter. Based on these observations, the proposed co-processing technique encapsulates for the filtering phase the least complex algorithms which are the fastest, namely, AllPairs, PPJoin and GroupJoin. The key characteristics of the algorithms are summarized below.

AllPairs (ALL). It is the first and most naive main memory algorithm to exploit the given threshold. During the inverted index lookup, it applies the prefix and length filters to prune candidate pairs [7].

PPJoin (PPJ). It extends ALL by applying the positional filter to generated candidate pairs[108]; therefore its verification phase is less loaded at the expense of a higher overhead during filtering.

GroupJoin (GRP). It is an extension to PPJ. Sets with identical prefix are *grouped* together. Each group is handled as a single set. Thus GRP has faster filtering, as it discards candidate pairs in batches. During the verification phase, the candidate pairs are expanded [12].

Even though the average verification runtime is reported as constant for most datasets in [62], employing the GPU for this part with a view to improving the overall performance is the main goal of the proposed co-processing technique. A key observation of [62] is that filtering contributes to the total running time significantly. Since filtering remains a CPU task, due to the Amdahl's law, employing the GPU for the verification in an ideal setting is expected to yield improvements of several times, but lower of an order of magnitude.

In the filter-verification framework, the whole join process is conducted incrementally, i.e. for a probing set, first its candidates are generated and verified, and then the algorithm proceeds to the next set. Naively allocating and copying small chunks of data on the GPU through a different kernel invocation per probing set, would incur an enormous overhead penalty. A more efficient alternative is to copy a large chunk of data stored in linear memory space to the GPU, process it there and copy back the results. Adopting this approach also improves overall runtime as the CPU builds candidate pair collections in waves and feeds them in a non-blocking manner to the GPU, which conducts the verification. Thus, time overlapping between the CPU and GPU tasks can be achieved.

In the proposed co-processing technique, it is assumed that the host (resp. device) is equipped with M_h (resp. M_d) memory capacity. The host runs 3 threads (H_0, H_1, H_2), while the device executes T threads in blocks of size B. The input collection is transferred in the device main memory in a linearized form, denoted as R_{el}, and it is accompanied by its offset array R_{off}. Table 4.2 summarizes the notation. In the next sections, both the host and the device tasks are analyzed thoroughly.

Parameter	Description
R	Collection of sets
C	Set of candidate pairs
OUT	Device output
R_{el}	Set elements array
$R_{off}, S_{off}, C_{off}$	Offset arrays
$\|\|R_{el}\||, \|\|C\||, \|\|OUT\||$ $\|\|R_{off}\||, \|\|C_{off}\||$	Size of arrays in bytes
H_i	The i^{th} host thread, $i \in 0, 1, 2$
T	Number of device threads
B	Thread block size
M_h	Host memory
M_d	Device memory
$M_c < M_d$	Device memory for candidate pairs

Table 4.2: Notation for the proposed co-processing CPU-GPU technique.

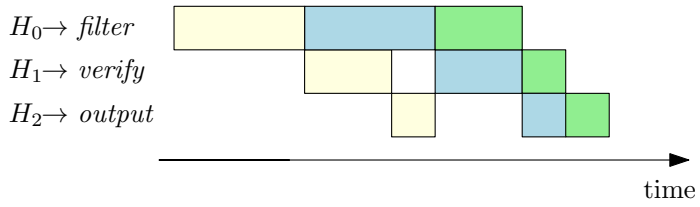


Figure 4.7: Execution overlap between host (H_0, H_2) and device (H_1).

4.2.1 Host Tasks

The host side is responsible for the filtering phase and works as the coordinator. Specifically, the host runs three threads. The first thread, H_0, conducts any filtering and builds chunks of candidates. When each chunk is built, the second thread, H_1, noted as *device handler*, enacts the verification phase by copying the chunk to device memory and launching the kernel code. Meanwhile, H_0 continues to build the next chunk of candidates. As soon as the device output is copied back to host memory, the third thread H_2 post-process it to form the final pairs result. Note that H_2 may not be invoked if an aggregation is performed on top of the join, i.e., if only the count of pairs is needed instead of the actual pairs. In such a case, the device counts the number of pairs and returns the result to H_1. Since the limited device memory is the most dominant constraining factor, the workload is divided into multiple chunks and the device is invoked several times. Hence, the host iteratively transfers as many candidates as the device memory can handle. This results in a non-blocking filtering phase, which naturally lends itself to an execution overlap. A visual representation of the execution overlap is depicted in Figure 4.7, where each color corresponds to a different data chunk.

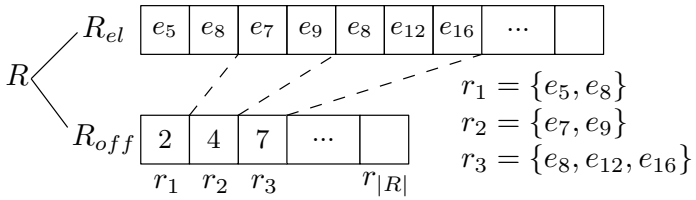


Figure 4.8: Example layout of a collection of threes sets in the device memory.

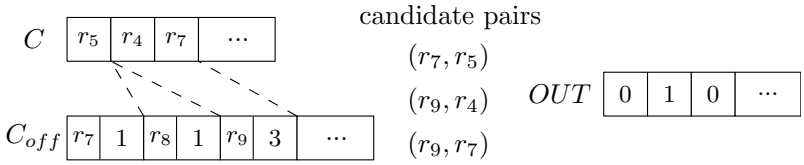


Figure 4.9: Candidates and output layout in device memory.

4.2.2 Device Tasks

The device side is responsible for the verification phase. It is invoked when the host prepares a chunk of candidate pairs. Two fundamental design choices of the proposed co-processing technique concern the data layout used and how the device is invoked. For the data layout, a detailed overview is given in the next section. Regarding the device invocation, there are two levels of concurrency in CUDA, *grid* and *kernel*. Grid level concurrency concerns mostly the overlap between computation and data transfers, while kernel level concurrency refers to how a single task is executed in parallel by many threads [16]. On the grid level, each input chunk of candidates is further divided into smaller chunks, each assigned to a different block. Thus, the overlapping between device computation and host-to-device data transfer is enhanced. On the kernel level, the encapsulated approaches are summarized as high-level verification alternatives (described in detail below).

Data Layout

By default, data is passed to the device as arrays stored in consecutive memory space. This is preferred because parallel execution benefits from coalesced global memory accesses. However, due to the nature of the problem, divergence in global memory access patterns is unavoidable, therefore the exploitation of on-chip memories is required to alleviate performance bottlenecks.

According to the linear memory layout, a collection R is physically implemented as the composition of two arrays: set elements R_{el} and offsets R_{off}. The former holds every set element of every set in the collection in a sequence, while the latter is used to delimit each set boundaries. Figure 4.8 depicts how a collection of sets R is stored in the device memory. The collection of sets R is transferred to the device once in the beginning of the process.

When the device is invoked to perform the verification phase, the host transfers an array of set IDs, noted as C, alongside with an array of offsets (C_{off}) which indicates the candidate pairs to be evaluated. In addition, an array of equal length to C, noted as OUT, is allocated on the device and it is used to store the output result. Essentially, OUT is an array of boolean flags where

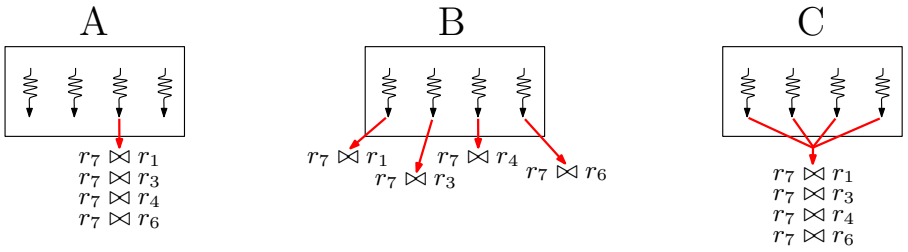


Figure 4.10: Thread workload per alternative.

true indicates that the corresponding candidate pair similarity is equal or greater than the given threshold. Figure 4.9 shows the mapping of probing sets to their candidate pairs and the output result array (e.g. only (r_9, s_4) pair is similar). Every even index position in C_{off} corresponds to a probe set id and every odd one to a candidates offset. In the figure, since $C_{off}[0] = r_7$ and $C_{off}[1] = 1$, r_7 should be compared against only the first entry of C. Since $C_{off}[3] - C_{off}[1] = 0$ and $C_{off}[2] = r_8$, r_8 does not participate in any candidate pair. Finally, since $C_{off}[5] - C_3[1] = 2$ and $C_{off}[2] = r_9$, r_9 should be verified against the next two sets in C.

Memory restrictions. In the presented data layout, most memory space is required to store the array of candidate set IDs C, which is of quadratic space complexity. Depending on the dataset and threshold value, the overall number of candidate pairs could reach billions. Considering that a set id is a 4-byte integer, $||C||$ space could be of several gigabytes. In addition, $||O|| = \frac{||C||}{4}$ space is also required to store the output result (assuming that 0 and 1 take a byte rather than a bit). Due to the limited device memory, the host iteratively transfers as many candidates as the device memory can handle (so that both candidate set IDs and output fit into the global memory). This favors the overlap between both ends, as the host builds the next chunk of candidate pairs and the device conducts the verification in parallel.

Verification alternatives

There are three main kernel-level concurrency layers or dimensions of in the problem as follows: *grid layout* (**GL**), which corresponds to thread execution; *memory hierarchy* (**MH**), which corresponds to efficient exploitation of the fastest on-chip memories; and last *output writing* (**OW**), which deals with result output. The latter is also distinguished into two cases depending on the type of querying being performed: *output count* (**OC**) for an aggregate query and *output select* (**OS**) for a full select of the similar pairs query.

These layers are tightly coupled and often intertwined, which means that certain options on a layer can rule out available options on the next ones. To investigate the impact of each available layer option, three alternative scenarios are presented below, which differ in the workload assigned to a single thread, as shown in Figure 4.10.

Alternative A. In the first alternative, the workload assigned to each thread is a probing set and the evaluation of all its corresponding candidate pairs, i.e., a single thread becomes responsible for the verification of all candidate pairs involving a specific probing set.

GL: A 1D grid of 1D blocks is launched, with the overall number of threads executed across all blocks (T) being equal to the input set collection size (R). Each thread is responsible for a probing set r_i and conducts all the joins with the corresponding candidates r_j.

MH: In this alternative, the shared memory is not utilized. An option could be each thread to load the corresponding probing set r_i to the shared memory, and then to access every candidate set r_j from global memory. Thus, a thread does not access global memory for r_i during the verification of each candidate pair. Since there is no fixed set size, blocks which handle sets of thousand set elements require larger amount of shared memory. For example, a thread block of 32 threads and average probing set size equal to 1000, would require 32×1000×4=128KB of shared memory which exceeds the capacity that modern GPUs have. In such cases, an adaptive approach must be followed where the thread block size is limited to allow for proper execution. In summary, the option to load r_i in shared memory implicitly defines the block size and gives rise to further challenges; thus is avoided in the presented technique.

OC: As every thread verifies its own candidate pairs independently, it can also count the amount of pairs satisfying the threshold using a register. After finishing the verification, each counter can be stored in shared memory in order for a fast reduction on block level to be performed. The result of each block is then stored in global memory for a grid level reduction to output the global count. The amount of shared memory required depends on the block size, but it is small (e.g., for 32 threads per block 128 bytes are needed).

OS: Having allocated the memory required for output array OUT, a device thread updates specific cells of the array. Incorporating the shared memory in this output is not straightforward because each thread does not know beforehand the length of its output pairs. Hence, allocating shared memory for the worst case scenario per thread is required. However this is practically impossible with the current GL. For example, if a block has 100000 candidate pairs to verify, 100KB of shared memory must be allocated, which exceeds the maximum allowed space. As a result, the shared memory cannot be employed to speed-up the output generation.

Alternative B. This alternative allocates less work to each thread by shifting the workload of a single probing set from a single thread to a single thread block. By assigning the comparisons referring to a probing set to a thread block, threads evaluate only a portion of the candidate pairs in parallel. The main benefit of this alternative is that the workload of threads within a block is more evenly distributed.

GL: A 1D grid of 1D blocks is launched, with the number of blocks being equal to the input set collection size. Each thread block is responsible for a probing set and each thread is assigned with a portion of candidate pairs to verify.

MH: First, the block threads load the corresponding probing set r_i to shared memory, then each thread verifies a portion of candidate pairs by accessing the corresponding candidate sets s_j from global memory. By using shared memory for one probing set per block, unlike alternative A, the maximum supported probing set size also increases.

OC/OS: Same as Alternative A.

Alternative C. Alternative B was designed with the goal to improve performance on the warp level. Alternative C further extends the rationale of alternative B. More specifically, in alternative C, each block is assigned with a probing set but with the difference that all the block threads cooperate to evaluate a candidate pair using the intersect path algorithm proposed in [33]. This

further mitigates the problem of balancing, since the threads do not only become responsible for an equal number of candidates, but also perform a roughly equal number of operations.
GL: A1D grid of 1D blocks is launched, with a number of thread blocks equal to the input set collection size. Each thread block is responsible for a probing set and multiple threads contribute to each candidate pair verification. A control thread outputs the result to global memory.
MH: Extending alternative B, due to thread cooperation, candidate sets are loaded to shared memory by default. If there is not enough space to hold all candidate sets, data is loaded in chunks, verification is conducted, and then the process continues with the next chunk.
OC: Since all threads within a block cooperate to verify a candidate pair, only a single thread is assigned with the task of incrementing the block's counter. Thus there is no need for a thread block counter reduction. However, grid level reduction is still required.
OS: The same applies for updating the output array OUT. If a candidate pair meets the threshold, only a single thread updates the corresponding array cell.

4.3 Implementation Issues

4.3.1 Host Details

The proposed co-processing technique leverages the work of Mann [62]. The main difference is that in the presented technique, the verification phase is delegated from the host to the device side via a multi-threaded implementation, which gives rise to the issues discussed in this section.

Candidate Serialization

The need to transfer candidate pairs to device memory highlights the necessity of efficient serialization methods. The goal is for the device to avoid complex global memory accesses. Therefore, the host is responsible for storing the candidates of a probe set in successive memory addresses. The options for serializing candidates C are listed as follows:

1. Use a sequence container such as *std::vector* and push back every new candidate. The main drawback is the extra memory checks on insertion to determine if a reallocation is required.
2. Prior reserve memory space for *std::vector* to avoid memory checks.
3. Use primitive arrays and handle memory operations manually.
4. Use a map structure where a key is an integer, i.e. the probe set ID, and its value is a *std::vector* containing the corresponding candidates IDs.

As a complement to C, a separate array C_{off} to delimit candidate pairs is required. Moreover, data that is insert into C_{off} are pairs consisting of a probe set ID and its corresponding offset on C. Omitting the probe set ID and inserting the candidate offset by itself, thus reducing the C_{off} size, implies that the probe set ID should be capable to be *extracted* from the index of C_{off}. For that to be possible, continuous probe sets IDs in ascending order must be processed, which might not always be the case.

The use of map is an intermediate stage to group together in memory probe set candidates. Before invoking the device, the map must be iterated in order to serialize every candidates list,

which is prerequisite to construct the final C. In addition, updating C_{off} per iteration is also required.

As will be shown in Section 4.4, primitive arrays, i.e., the third option listed above, perform better than *std::vector*, i.e., the first two options, and are adequate for ALL and PPJoin that produce the full candidate set for a single probing set in a single phase. For GRP, which employs two phases during candidate generation, a map structure is necessary if the full verification phase is delegated to the GPU.

GroupJoin Work Split

The three best-performing algorithms examined can be divided into two categories: those which generate every candidate pair in one phase, i.e. ALL and PPJ, and the one, GRP, which requires an extra phase (group expanding) to output all candidate pairs.

For each probe set in ALL and PPJ, candidates are guaranteed to be stored in successive memory addresses. Thus, candidates are serialized using primitive arrays. In contrast, GRP generates candidates that are intertwined due to the expanding phase. Therefore storing candidates in a map structure is required. However, the experimental evaluation of this approach indicates that serializing the map adds extra overhead, rendering this option unfeasible.

To incorporate GRP in the co-processing scheme, a part of the verification is assigned to the GPU as well. Moreover, the verification of every candidate pair generated in the first phase is assigned to the device. Hence, candidates from this phase are serialized in primitive arrays and transfer them to the device. Every candidate pair that emerges from the second phase, i.e. group expanding, is left to be verified in the host side by H_0.

By splitting the work, overall performance is alleviated as shown in Section 4.4. In spite of this gain, transferring the whole GRP verification workload to device remains a challenge and would further improve the performance.

4.3.2 Device Details

Block size

In every verification alternative, the kernel grid launched is composed of 1D blocks. The block size B must be a power of 2 for reduction on the shared memory to work properly. $B = 32$ is preferred since a warp can be considered as the CPU thread equivalent. However, as shown in Section 4.4, there is a correlation between block size and set size; thus, increasing the block size should be considered when the third verification alternative is the best performing one.

Merge and Intersect Path

Computing a list intersection can be derived from a list merge operation. An efficient parallel merging algorithm for GPUs is Merge Path [32]. Given two sorted lists A and B, Merge Path considers the order in which elements are merged, which is equivalent to the traversal of a grid, noted as *Merge Matrix*, of size $|A| \times |B|$. Beginning from the top left corner of the grid, the path can only move to the right if $A[i] \geq B[j]$ or downwards otherwise, until it eventually reaches the bottom right corner.

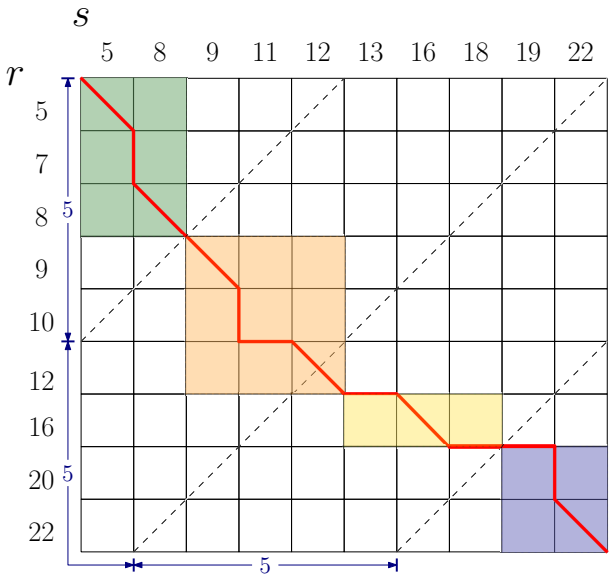


Figure 4.11: Intersect path example.

There are two partitioning stages, one on kernel grid level and the other on block level. On grid level, equidistant cross diagonals are placed on the Merge Matrix. Using binary search, the point of intersection for a cross diagonal and the path is found. As a result, each SM is assigned to merge non-overlapping portions of the input. On block level, threads cooperate in loading the required list portions on shared memory and then merge them in global memory.

By modifying Merge Path in [33], the authors propose a fast list intersection algorithm, called Intersect Path. They introduce a new diagonal path move, if $A[i] = B[j]$. The same partitioning stages still hold. Each SM outputs a portion of the intersection to global memory. Because in alternative C, thread blocks verify whole candidate pairs, Intersect Path must be modified accordingly. Thus, both partitioning stages are performed on block level.

Set Intersection Count

In order to verify a candidate pair, threads must calculate the intersection of two sets. Each thread in alternatives A and B, independently performs a merge-like loop and counts the number of intersects. On the other hand, in alternative C, threads collaborate to output the intersection. Since the problem can be reduced to finding the intersection count of two sets, a modified Intersect Path algorithm is used to divide the workload between block threads, as mentioned above.

Cross diagonals are equally placed apart at $\lceil \frac{|r_i|+|s_j|}{B} \rceil$ [33]. Each thread is assigned with a partition with starting point the intersection of the path and the corresponding diagonal. An example of Intersect Path using $B = 4$ threads is shown in Figure 4.11 (each color corresponds to a different thread). Starting from the top left corner, cross diagonals are placed apart 5-hops away, where hops are along the axes. Each thread calculates independently its partition intersection count and stores it in registers memory. When finished, intersection counts are copied in shared memory for a fast reduction to output the overall intersection.

Count Reduction

The primary focus is to minimize the global memory transactions. Whenever possible, the registers are used to store counts per thread. Similarly, every thread count is stored to shared memory and with the use of warp shuffle functions fast reduction is enabled. Hence, when performing an aggregation on top of the join, only a single write to global memory is required per block. To output the final count, the `reduce` function from the `Thrust` library is used on the global memory-resident intermediate counts.

4.4 Evaluation

The goals of the experiments are threefold: (i) to show the extent to which the verification phase delegated to GPU is hidden (overlapped) by the index and filtering phases running on the CPU; (ii) to give concrete evidence about the speed-ups achieved in practice, and (iii) to provide explanations about the observed behavior.

4.4.1 Experiment setting

The experiments were conducted on a machine with an Intel i7 5820k clocked at 3.3GHz, 32 GB RAM at 2400MHz and an NVIDIA Titan XP. This GPU has 3840 cores, 12 GB of global memory and a 384-bit memory bus width.

The overall runtime, noted as *join* time, is the composition of candidate generation and serialization performed by H_0 host thread, and the verification conducted by the device. The former is referred as *filtering* time and the latter as *verification* time. The data preprocessing time, which is performed exactly as in [62], is not included. The experiments are conducted for all datasets using ten thresholds in the range [0.5, 0.95]. Moreover, the experiments focus on self-joins using the Jaccard similarity. Additionally, an aggregation on top of the set similarity join is performed. The reported time for each experiment is an average over 3 independent runs (no significant deviation was observed). The *filtering* and total *join* time are measured with the `std::chrono` library. For the *verification* time the CUDA event API is used. The times for allocating device memory and transferring chunks of candidates to the device were negligible for all the experiments and furthermore were completely hidden due to the execution overlap.

For the experiments seven real world datasets that were also employed in [62] are used. Table 4.3 shows an overview of each dataset characteristics. A summary of each dataset (adapted from [62]) is as follows:

AOL: query log data from the AOL search engine. Each set represents a query string and its set elements are search terms.

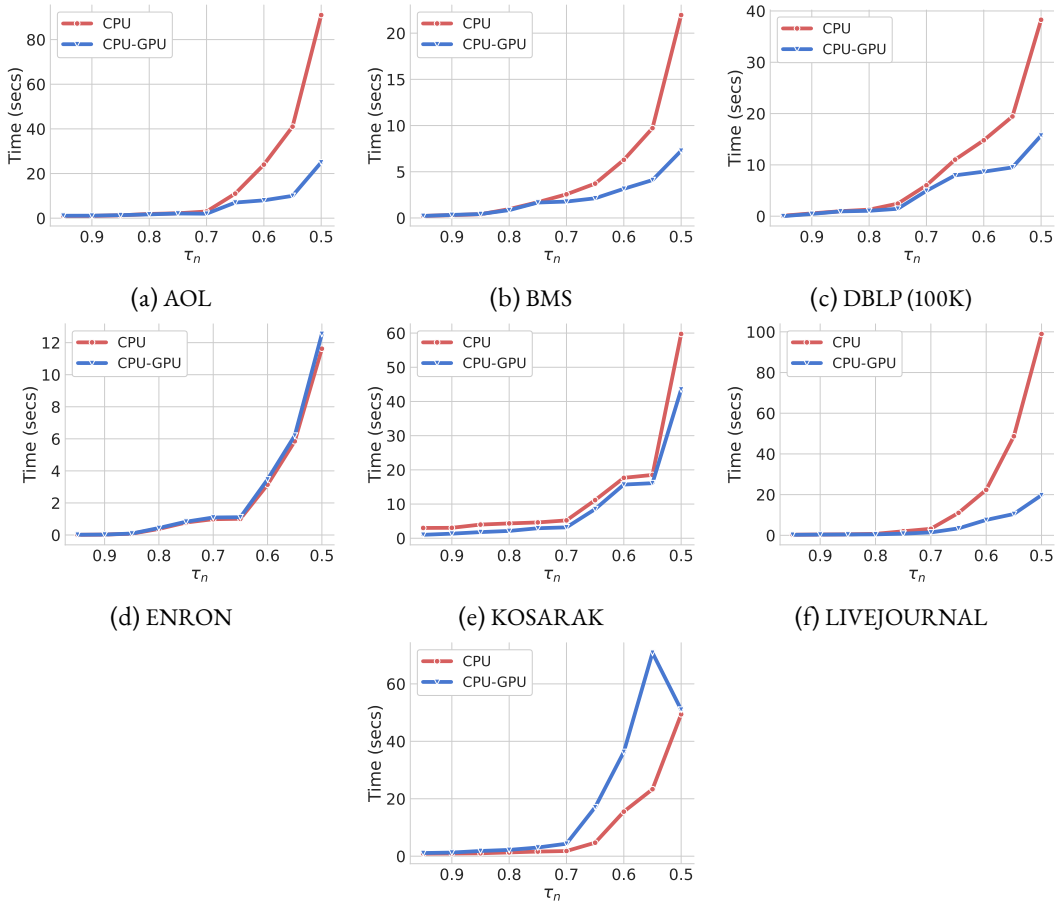
BMS-POL: purchase data from an e-shop. Each set represents a purchase and its set elements are product categories in that purchase.

DBLP: article data from DBLP bibliography. Each set represents a publication and its set elements are character 2-grams of the respective concatenated title and author strings.

ENRON: real e-mail data. Each set represents an e-mail and its set elements are words from either the subject or the body field.

Dataset	Cardinality	Average set size	# diff set elements
AOL	$1.0 \cdot 10^7$	3	$3.9 \cdot 10^6$
BMS-POS	$3.2 \cdot 10^5$	6.5	1657
DBLP*	$1.0 \cdot 10^5$	88	7205
ENRON	$2.5 \cdot 10^5$	135	$1.1 \cdot 10^6$
KOSARAK	$6.1 \cdot 10^5$	8	$4.1 \cdot 10^4$
LIVEJOURNAL	$3.1 \cdot 10^6$	36.5	$7.5 \cdot 10^6$
ORKUT	$2.7 \cdot 10^6$	120	$8.7 \cdot 10^6$

Table 4.3: Datasets characteristics (*DBLP is further increased later).



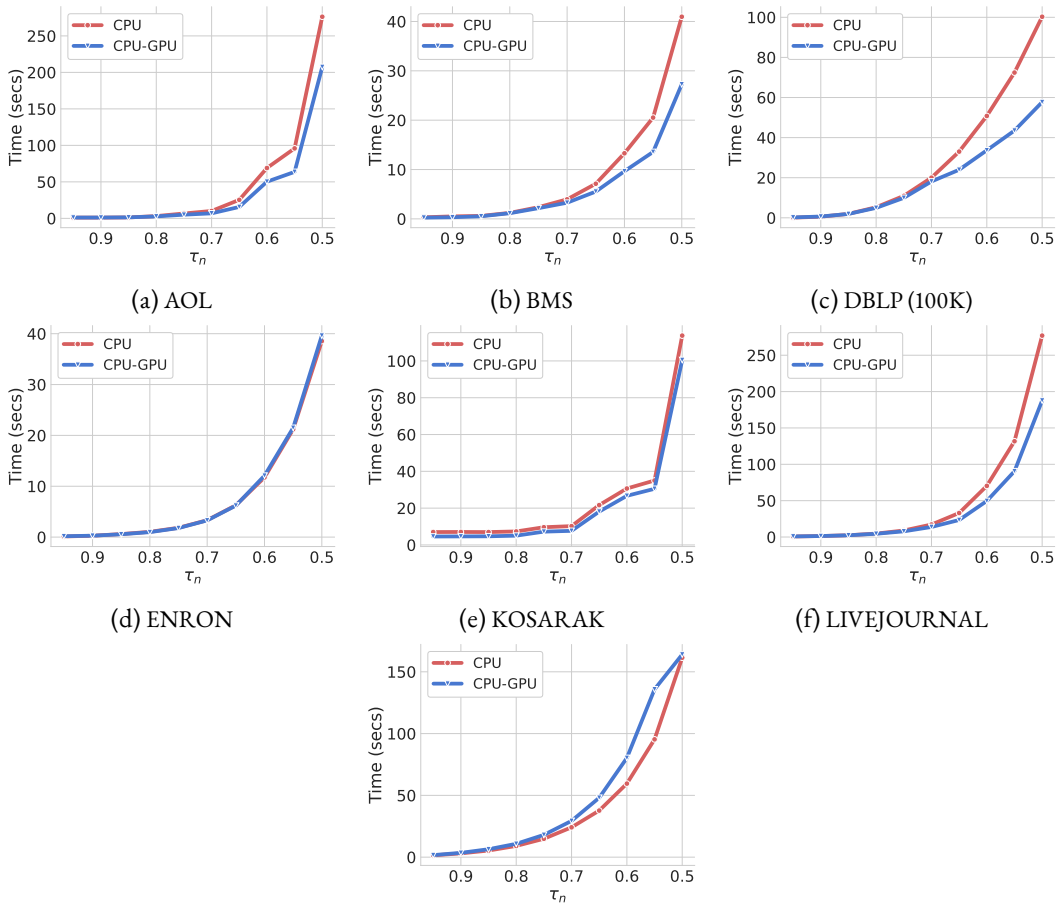
(a) AOL (b) BMS (c) DBLP (100K)

(d) ENRON (e) KOSARAK (f) LIVEJOURNAL

(g) ORKUT

Figure 4.12: Comparison between the best verification times of the CPU and an unoptimized GPU execution ($B = 32$ and $M_c = 4GB$) for different thresholds.

KOSARAK: click-stream data from a Hungarian on-line news portal. Each set represents a user behavior and its set elements are links clicked by that user.



(a) AOL (b) BMS (c) DBLP (100K)

(d) ENRON (e) KOSARAK (f) LIVEJOURNAL

(g) ORKUT

Figure 4.13: Comparison between the best join times of the CPU and an unoptimized GPU execution ($B = 32$ and $M_c = 4GB$) for different thresholds.

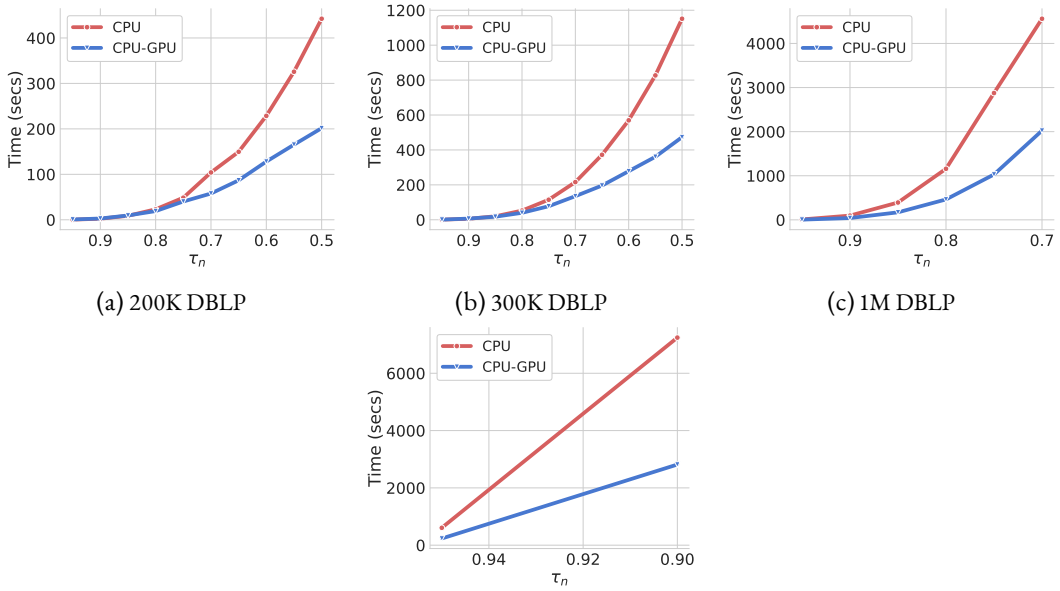
LIVEJOURNAL: social media data from LiveJournal. Each set represents a user and its set elements are interests of that user.

ORKUT: social media data form ORKUT network. Each set represents a user and its set elements are group memberships of that user.

4.4.2 Main Experiments

The baseline comparison of the presented co-processing technique, noted as CPU-GPU, is to the CPU standalone implementation of the work presented in [62]. The speedups that can be achieved during the verification phase can be up to more than 5X as shown in Figure 4.12. Nevertheless, in practice, it is more important to investigate the total response time, which includes both the filtering and the verification phase.

The best join times measured for both techniques are presented in Figure 4.13. Each time reported for the CPU is the overall best of the three algorithms execution and therefore the best that



(a) 200K DBLP

(b) 300K DBLP

(c) 1M DBLP

(d) Complete DBLP (6.1M)

Figure 4.14: Comparison between the best times of the CPU and an unoptimized GPU execution for larger datasets for different thresholds.

τ_n	join	CPU		GPU	||C||
		filtering	serialization	verification	
0.95	233	134	96	92	72.7GB
0.9	2815	1892	921	698	0.56TB
0.85	11367	8935	2430	2311	1.8TB

Table 4.4: CPU-GPU join time decomposition for processing the complete DBLP dataset (in secs).

can be achieve in the experimental setup. Respectively for the CPU-GPU, each time reported is the best among unoptimized executions ($B = 32$, $M_c = 4$ GB) of the three CPU filtering algorithms and the three GPU verification alternatives. Thus, performance can be further improved, especially on datasets such as ENRON (Figure 4.13d) and ORKUT (Figure 4.13g) where the CPU-GPU performs similar or worse than the CPU.

As shown in Figure 4.13, for every dataset on large thresholds, i.e. the threshold range is in [0.7, 0.95], the GPU does not yield any performance speedup. Given the fact that in large thresholds the number of candidate pairs, hence the memory required to store them, is quite smaller than the device memory M_c, the GPU remains idle during the candidate generation phase, only to be invoked once before the process finishes. On the other hand, billions of candidate pairs are generated when using smaller thresholds ([0.5, 0.65]). As a result, the GPU is invoked several times leading to an execution overlap and therefore to faster join times.

This conclusion is further supported by running the evaluated techniques over larger DBLP datasets as show in Figure 4.14. As it can be seen, speedups are much more evident, e.g., 2.6X

	0.95	0.90	0.85	0.80	0.75	0.7	0.65	0.6	0.55	0.5
ALL	1	1	0	1	2	3	4	4	5	3
PPJ	2	2	2	4	4	3	2	2	1	3
GRP	4	4	5	2	1	1	1	1	1	1

Table 4.5: Number of datasets in which each algorithm is the best per threshold.

in Figure 4.14d, and are tangible even for larger thresholds, where the candidate pairs are tens of billions. In Figure 4.14c, the candidates are 3.5B, 17B and 77B for thresholds 0.9, 0.8, and 0.7, respectively, and there are clear benefits for the last two settings. In addition to Figure 4.14d, the CPU-GPU join time is decomposed, as shown in Table 4.4 (note that the standalone CPU technique takes more than 8.5 hours to complete for $\tau_n = 0.85$ for the complete DBLP dataset, therefore the respective experiments are committed in Figure 4.14d). In Table 4.4, it is shown that the join time is solely attributed to the filtering time; the join time is roughly equal to the sum of filtering, index building and serialization. This means that the co-processing scheme that assigns verification only to the GPU has reached its maximum potential.

Key result: The proposed co-processing technique manages to hide the impact of verification phase on the running time of the similarity join, when (i) the candidates are in the order of tens of billions at least and (ii) the filtering and verification phases are intertwined.

4.4.3 Algorithm Performance

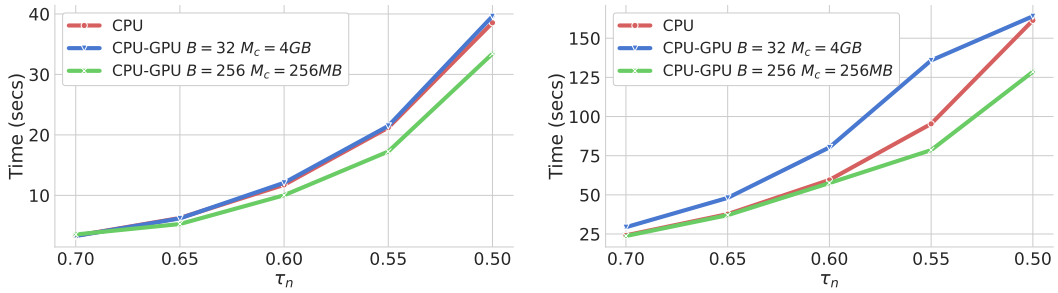
Table 4.5 shows which algorithm exhibited the best performance in Figure 4.13 on the co-operative CPU-GPU technique. It can be observed, that as in the CPU comparison in [62], there is no algorithm that dominates the others. However, ALL favors low thresholds, PPJ mid-range and GRP the high ones, especially for the datasets with small average size.

Next, the algorithm behavior issues are discussed in relation to the three datasets, where CPU-GPU does not show tangible benefits in Figure 4.13 even for relatively low thresholds, namely ENRON, KOSARAK and ORKUT.

Execution overlap and the role of M_c

The most significant gain due to the device stems from the execution overlap between indexing and verification. ALL invokes the device earlier and more frequently because of its fast candidate generation on all datasets. PPJ and GRP have higher index times, generate less candidates and as a result invoke the device less frequently.

To exploit the execution overlap, and therefore keep the device busy most of the time, defining the appropriate M_c is necessary. The ENRON and ORKUT datasets are examined. For these two datasets, all three algorithms have similar index times. The overall best algorithm for the standalone CPU execution is PPJ since it generates less candidates than ALL in the same indexing time frame and does not include any candidate expanding during the verification phase as GRP does. Figure 4.15a and Figure 4.15b show the join time for the ENRON and ORKUT dataset respectively, using PPJ as the index algorithm. By decreasing M_c, and by fine-tuning B as will be



(a) ENRON (b) ORKUT

Figure 4.15: Impact of execution overlap on the GPU performance.

shown in the device performance section, overlapped execution between the CPU and the GPU is increased and thus, verification time is completely hidden in the execution overlap.

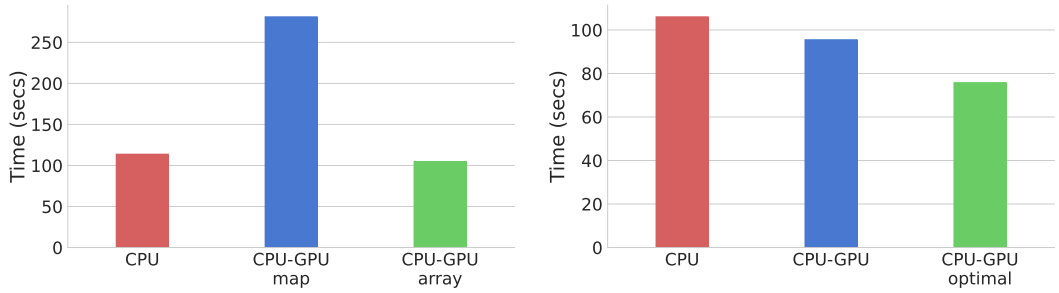
GroupJoin Issues

Based on the experimental evaluation of the filtering algorithms, the KOSARAK dataset is more efficiently processed by GRP regardless of the threshold. In Figure 4.16, the join times of the CPU and two CPU-GPU executions for the KOSARAK dataset are presented. In the first CPU-GPU execution, a map structure is used to delegate the whole verification phase to the device. In the second, raw arrays are used to delegate only the candidates generated in the filtering phase to the device, the verification workload is split between host and device. As it can be seen, the overhead imposed by using a map, which entails numerous memory checks, is not outweighed by faster verification. On the other hand, the drawback of assigning part of the verification to the CPU is that in datasets, such as KOSARAK, where the group expanding yields a larger number of candidates than the first phase, the host is assigned with significantly more verification workload than the device. Therefore, it is expected that the proposed co-processing technique not to yield any benefits for such cases.

Another issue during the expanding phase is that the host iterates and skips candidate pairs that are to be verified on the device. This adds extra overhead. On every dataset, except AOL, BMS and KOSARAK, the group expanding phase generates less candidates than the filtering phase. Nonetheless, the candidate skipping overhead cannot be avoided. Furthermore, in Figure 4.16, its impact on join time is illustrated. For the DBLP dataset with $\tau_n = 0.5$, the group expanding does not generate any candidate, thus, by removing the expanding phase, a performance gain is achieved.

4.4.4 Device Performance

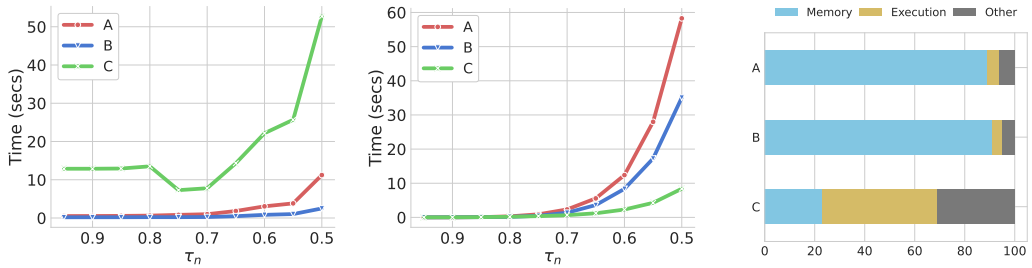
There are two kernel parameters, which require fine tuning depending on the dataset characteristics: (i) the verification alternative to be followed and (ii) the thread block size. Therefore, it is of a great importance to understand what values should be used for each parameter in order to optimize the performance of the co-processing technique.



(a) KOSARAK, alternative B.

(b) DBLP, alternative C.

Figure 4.16: Comparison of GRP flavors ($t = 0.5$, $M_c = 4$GB).



(a) Kosarak kernel execution.

(b) Livejournal kernel execution.

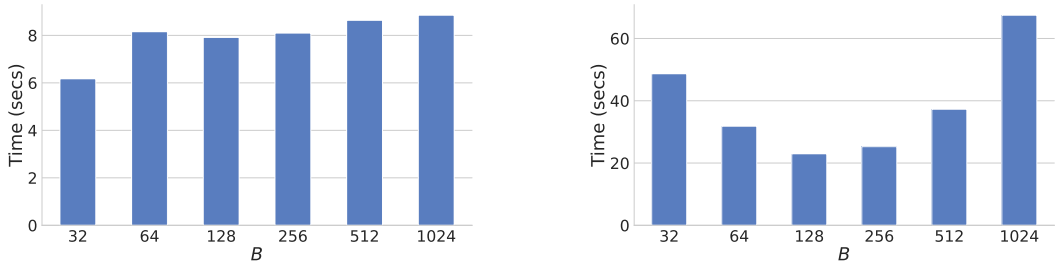
(c) Alternatives boundaries.

Figure 4.17: Alternatives comparison.

Comparison of the workload alternatives

The two main differences between the verification alternatives is how threads access global memory and how they calculate the intersection of two sets. Each thread in alternative A accesses its respective candidate pair sets and calculates each intersection. Because of that, intra-warp divergence is maximized since a thread has its own execution path. Alternative B alleviates the performance because the threads of a block collaborate to load the corresponding probing set to shared memory and then, less candidate pairs per thread are verified. However, since each thread independently loads a candidate set and calculates the intersection, intra-warp divergence is still present. In alternative C, each block's threads collaborate first to load the probing and each candidate set to shared memory, and second, to perform the intersections. Therefore intra-warp divergence is low.

For datasets with small average set size ($\leq$ 10) such as AOL, BMS, KOSARAK, the global memory access footprint is also small, which renders alternatives A and B competitive. As shown in Figure 4.17, both A and B have similar performance for large thresholds, but for small ones alternative B performs better since thousands more thread blocks are launched and thus device occupancy is increased. On the contrary, alternative C seems infeasible for small set sizes, since the overhead to store them in shared memory dominates the verification time. The advantage of alternative C becomes apparent in datasets characterized by larger average set size such as DBLP, ENRON, LIVEJOURNAL and ORKUT. Figure 4.17b shows the performance of the three al-



(a) AOL, Alternative B.

(b) ORKUT, Alternative C.

Figure 4.18: Block size impact on verification time, $t = 0.5$, $M_c = 4GB$.

ternatives for the LIVEJOURNAL dataset. As the number of candidate pairs increases for small thresholds, alternative C performs better since it achieves a higher warp execution efficiency (200% and 33% increase from A and B respectively). In addition, it has lower memory dependency as shown in Figure 4.17c. By minimizing the global memory access footprint, alternative C becomes mainly execution dependent. Other dependencies such as instruction fetch, instruction issue and synchronization arise, but they are less expensive than constantly accessing the global memory in a non-optimal way.

Key result: It is more beneficial to employ alternative B for sets of small size, and alternative C otherwise.

Block size

For datasets with small average set size, where alternative B is preferred, $B = 32$ has the best performance. Increasing B in such datasets leads to a higher proportion of inactive warps and hence increases the verification time as shown in Figure 4.18a. As the number of average set size in candidate pairs increases, alternative C is preferred to minimize the global memory access bottleneck. Alongside, fine tuning of B is also required to achieve best device performance. For example, as shown in Figure 4.18b for the ORKUT dataset, if $B = 32$, each thread receives more workload than optimal. By increasing B up to 128, the workload is more evenly distributed since more threads contribute to the join. If B is further increased, this leads to a high number of inactive warps, and therefore to low warp execution efficiency.

Key result: Judiciously increasing B when alternative C is employed leads to higher performance. When combined with lower M_c, it manages to fully overlap verification and filtering phases.

4.5 Related Work

Exact Set Similarity Join

There is a substantial body of literature for exact set similarity join. In [62], Mann et al. provide a comprehensive survey on set similarity join for prefix filter based techniques. Recently, Wang et al. [100] propose the SKJ algorithm which is built on top of the prefix filter and encapsulates

two skipping techniques to further improve set similarity join. In [23], Deng et al. introduce the SizeAware algorithm which divides the input collection to small and large sets, and process them separately. For the distributed setting, Fier et al. [28] conduct an experimental survey for set similarity joins on the MapReduce framework and highlight the inability of the evaluated algorithms to scale. More recently, Yang et al. [109] devise a length-based distribution framework for set similarity join on top of the Apache Storm platform. As a result, they avoid pitfalls of previous prefix-based distributed techniques and show promising improvement on runtimes. In [29], the authors introduce a multi-threaded CPU framework, which encapsulates the best performing standalone CPU algorithms.

In the GPGPU setting, the first work proposed for the exact similarity join problem on GPUs has appeared in [81]. The basic idea is to invoke the GPU for each probe set and calculate its similarity score with every other set in parallel. However, this entails an enormous launch overhead penalty. In [77], the authors propose a multi-pass scheme to overcome the scalability constraints of [81]. In [80], the authors adopt the same multi-pass scheme as in [77] and compute the set similarity scores by using a static inverted index without any kind of filtering. Thus, their solution remains more robust against varying threshold values. In [84], the authors introduce the bitmap filter for a more lightweight filtering phase. As a proof of concept, they demonstrate that employing the GPU is beneficial because of the fast parallel bitwise operations. However, their GPU solution cannot scale. The proposed co-processing differs from existing GPGPU techniques since it utilizes both ends to address the exact set similarity join problem.

Approximate Set Similarity Join

Most of the techniques proposed for approximate set similarity joins resort to data reduction to speedup the join process. In [18], the authors employ the parallel-friendly MinHash algorithm to estimate the Jaccard similarity of two sets. Their solution is space-efficient since they only store set signatures instead of whole sets to perform the similarity join. However, due to the MinHash nature (i.e. data partitioning in bins), fine-tuning is required to achieve balance between accuracy (to avoid false positives) and execution time. In [55], Li et al. reduces the preprocessing time from the original MinHash by using one permutation hashing. Furthermore, Ji et al. [44] introduce MinMax hash and reduce the hashing time by half. Recently, Wang et al. [99] propose MaxLogHash to accurately estimate similarities in streaming sets. The main limitations of the above techniques is that they are inherently limited to Jaccard similarity only.

Other problems, which are close to set similarity join and have been studied on the GPGPU paradigm, are similarity join and similarity (nearest neighbor) search. For set similarity join, an early proposal has appeared in [56], according to which Lieberman et al. cast the similarity join operation as a GPU sort-and-search problem. First, they create a set of space-filling curves using bitonic sort on one of the input relations; then, they process each record from the other relation in parallel by executing searches in the space filling curves, using the Minkowski metric for similarity. Similarity joins are also discussed in [10], where two nested loop join (NLJ) algorithms are presented: a naive NLJ and a faster index-supported NLJ. The index is created on the CPU side during the preprocessing phase. Both algorithms use the Euclidean distance for similarity and thus they are not suitable for set similarity joins.

For the similarity search the proposals in [72, 115] use a GPU-based parallel Locality Sensitive Hashing (LSH) algorithm to perform approximate k-nearest neighbor (kNN) search. In [101], a hybrid CPU-GPU framework, which uses LSH combined with other techniques, such as reservoir sampling is presented, where, the CPU constructs hash tables and the GPU process them and performs a count-based top-k selection. In [103], the authors propose a two-level tree and a re-ranking method for fast approximate nearest neighbor search. In [46], Johnson et al. present a framework to compute a k-NN graph. All these proposals can be deemed as devising key elements for building additional approximate set similarity join techniques.

4.6 Remarks

The presented co-processing technique utilizes both processors in contrast to existing techniques where the complete workload is delegated either to the CPU or GPU. In addition, solutions are provided to the issues involved, such as data layout on the device memory, serialization, and thread workload. Using real datasets, is shown that fully overlap between the GPU tasks with the CPU ones is achieved, when the candidate pairs are at the order of tens of billions of sets, which in turn leads to speedups of up to 2.6X. Motivated by the significant speedups reported, co-processing techniques are further investigated next. More specifically, in the next chapter, after presenting a complete empirical evaluation between the state-of-the-art set similarity join techniques, a hybrid framework that employs both the CPU and GPU for the problem of exact set similarity join is introduced.

5 Towards a Hybrid Framework for Set Similarity Join

In the previous chapter, an introduction to the preliminaries of the set similarity join problem was given. In addition, the proposed co-processing GPGPU technique was presented, alongside with its extensive evaluation against single core standalone CPU techniques. However, other GPGPU addressing the set similarity join have been proposed [77, 80, 81, 84]. Therefore, to better understand the status quo in GPU-enabled set similarity joins further investigation is required. To this end, the contributions of this chapter are twofold.

The first contribution is to provide a comprehensive presentation and comparative evaluation of the state-of-the-art GPU-accelerated set similarity join techniques. By conducting an extensive performance analysis using eight real world datasets, the conditions under which each technique becomes the dominant one are identified. The findings demonstrate that there does not exist a clear winner among the evaluated techniques; in other words, each alternative has its own sweet spot depending on the data and query characteristics. Thus, a set of guidelines as to when to use each technique is derived.

Motivated by the fact that in a single GPU-endowed machine setting, no globally dominant technique exists, the second contribution is the development of a hybrid framework for the exact set similarity join operation which encapsulates the state-of-the-art both CPU and GPU techniques and utilizes both processors concurrently. Through the extensive experimental evaluation on both real world and synthetic datasets, it is shown that the proposed hybrid framework achieves speedup of up to 3.25X over the best single-threaded CPU-standalone and GPU-enabled techniques. Additionally, the benefits over multi-threaded CPU solutions are also presented, where the hybrid framework's speedups are higher.

The rest of this chapter is organized as follows. Section 5.1 gives an overview of the state-of-the-art GPGPU techniques. In Section, 5.2 the comparative evaluation of the GPGPU techniques is presented. The proposed hybrid framework for set similarity join is presented in Section 5.4 , with its evaluation presented Section 5.5. Finally, the chapter concludes with Section 5.6 where the findings are further discussed.

5.1 Overview of GPGPU Techniques

The techniques that use the GPU fall into two categories: (i) those that split the workload between the CPU and the GPU, and (ii) those that move the whole workload to the GPU. For the former, the only technique to date is thoroughly described in Section 4.2. Regarding the latter, more details on the existing techniques found in the literature, are given below.

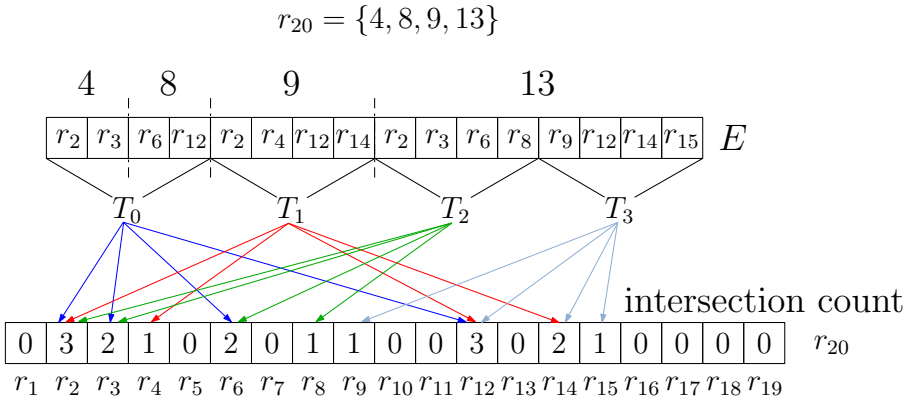


Figure 5.1: gSSJoin workload allocation example for four GPU threads.

5.1.1 Standalone GPU techniques

Ribeiro et al. with their successive works in [77, 80, 81] perform the complete exact set similarity join on the GPU. An overview of the three main techniques proposed is provided below along with a brief discussion.

GPU-based Set Similarity Join (gSSJoin) [81]. *gSSJoin* first constructs a static inverted index over all set elements; then, for each probe set, it performs a set similarity operation consisting of three steps: (i) First, an intersection count between the input set and every other set is calculated. By using the inverted index, the workload is evenly distributed among the GPU cores. Figure 5.1 depicts how *gSSJoin* evenly distributes the workload among four GPU threads. In the example in the figure, first, the four inverted lists corresponding to the set elements of probe set r_{20} with ids 4,8,9, and 13 are concatenated in a logical vector E. Next, given the number of GPU threads, $|T| = 4$, each thread is assigned with $|E|/|T| = 16/4 = 4$ elements. Each element contributes to the computation of the intersection count of a specific candidate pair containing r_{20}. Each thread processes independently its corresponding partition; this implies that a count may be incremented by multiple threads and thus the atomic add operation is used to ensure correctness. (ii) Then, the Jaccard similarity of every pair is calculated and stored in global memory. (iii) Finally, the pairs that have similarity higher or equal to the threshold value are selected and transferred to the main memory. To reduce the transfer size, a stream compaction technique is employed.

Discussion. *gSSJoin* can be considered as an efficient brute force approach, since, for every pair, the corresponding similarity score is calculated. By completely avoiding any kind of filtering, it remains more robust to variations of threshold values and set element frequency distribution; this is its strongest point. By not using any kind of filtering, proven to be quite effective especially in large threshold values, *gSSJoin* conducts numerous unnecessary intersection counts and may add extra overhead to the overall runtime in contrast to filter-based approaches. However, the main drawback of *gSSJoin* is the inherent launch overhead, which starts dominating in medium

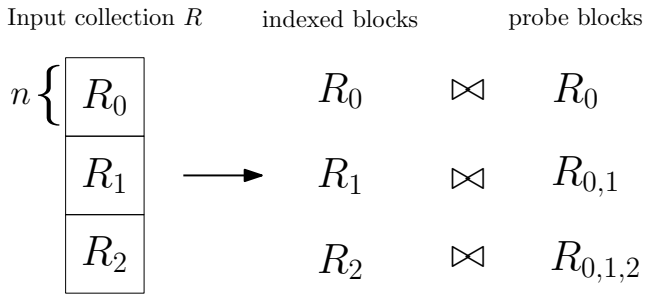


Figure 5.2: Block partitioning scheme used in fgSSJoin and sf-gSSJoin.

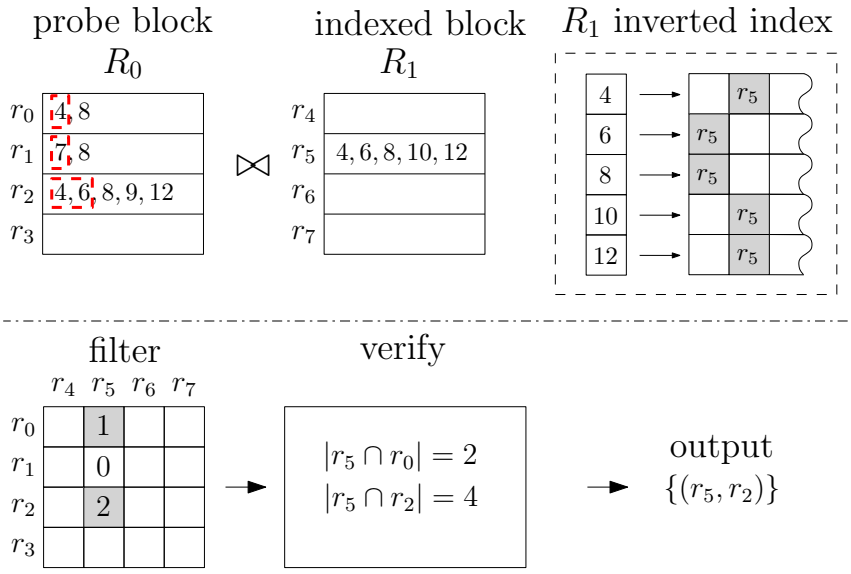


Figure 5.3: fgSSJoin probe block example (block size $n = 4$, threshold $\tau_n = 0.8$).

and large datasets ($> 1M$): as the dataset size increases, the launch overhead increases as well, since each probe set corresponds to several independent kernel calls.[1]

Filter-based gSSJoin (fgSSJoin) [77]. *fgSSJoin* extends *gSSJoin* by encapsulating a filtering phase. In addition, a block partitioning scheme is adopted in order to process collections of arbitrary size. In summary, the input collection is divided into blocks of size n, with n being an input parameter chosen accordingly for the result to fit in global memory. Through an iterative process, a block is indexed and probed against itself and all its predecessors. Figure 5.2 illustrates the block partitioning and probing. First, block R_0 is indexed and probed against itself. After that, block R_1 is indexed and probed against itself and its predecessor R_0. Finally, the same procedure applies to block R_2. The sets are ordered by their size; this allows a length filter to be applied and whole block-to-block comparisons to be skipped.

[1] In the evaluation setting, a launch overhead is approximately 0.01ms, which is several times higher than the execution overhead of a single kernel. Given the fact that for each probe set, gSSJoin launches many kernels to find its similar pairs, as the dataset size increases, the aggregated overhead becomes the main performance bottleneck.

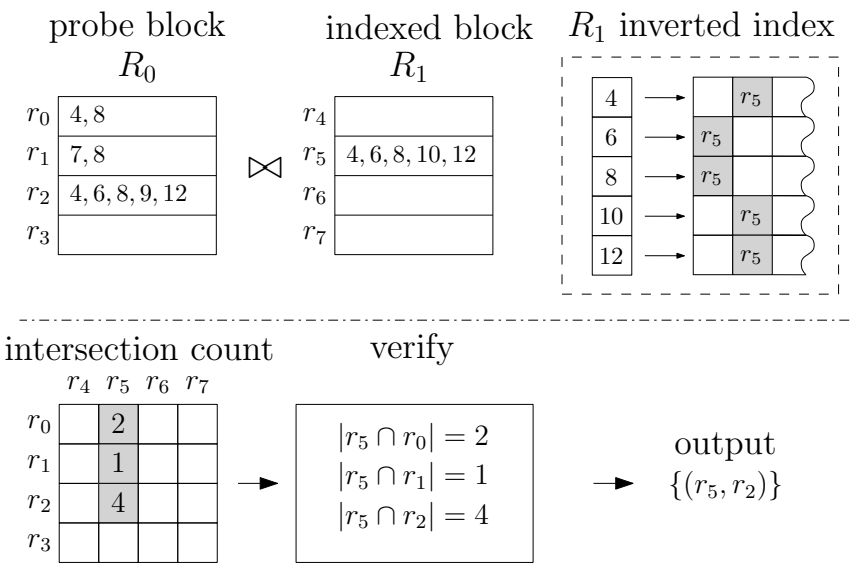


Figure 5.4: sf-gSSJoin probe block example (block size $n = 4$, threshold $\tau_n = 0.8$).

During the filtering phase, partial intersection counts between indexed and probed sets prefixes are calculated and stored in global memory, given that on-chip memories are too small to hold the results. This incurs an $O(n^2)$ space complexity to cover the worst case scenario, where all set pairs are candidates. Subsequently, every pair that has a non-zero partial intersection count undergoes full verification, whereas the rest is pruned. The workload is distributed similarly to *gSSJoin*. Figure 5.3 depicts the filter-verification steps of *fgSSJoin*. Initially, by using the prefix filter, *fgSSJoin* computes the partial intersection counts between the probe sets r_0, r_1, r_2 and the indexed set r_5. Before moving to the verification phase, the pair (r_5, r_1) can be safely pruned since its partial intersection count is zero. Finally, the non-zero partial intersection count pairs undergo full verification. As a result, only the pair (r_5, r_2) has the required overlap and thus it is added in the output.

Discussion. By integrating a filtering stage, *fgSSJoin* reduces the candidate search space, especially in large threshold values. Its most significant contribution is the block partitioning scheme, which overcomes the launch overhead limitation of *gSSJoin*. In addition, thanks to a length filter, whole blocks can be pruned. A scalability issue arises when processing the quadratic memory space required to store the intermediate intersection counts. More specifically, after each probe block call, this space must be reset to zero to ensure correctness. As a result, the accumulated overhead may become a significant issue, as the dataset size increases.

Size-filtered gSSJoin (sf-gSSJoin) [80]. *sf-gSSJoin* adopts the same block partitioning scheme as *fgSSJoin*. It performs the set similarity join into two phases per probe. First, it calculates the complete intersection counts between probed and indexed sets without any prefix filtering. Then, for each non-zero intersection count, it calculates the corresponding pair's Jaccard similarity. Finally, the output stored in linear global memory is transferred back to main memory. Figure 5.4 gives an overview of *sf-gSSJoin*'s probing. In contrast with *fgSSJoin*, *sf-gSSJoin* directly calculates the complete intersection counts between probe sets r_0, r_1, r_2 and the indexed set r_5 in its first

pass. This results to more global memory accesses, compared to *fgSSJoin* when prefix filtering is effective.

Discussion. As stated before, as the threshold value decreases, filter effectiveness degrades leading to an increased number of candidates and subsequently larger execution times. *sf-gssJoin* combines the robustness of *gSSJoin* against varying threshold values and set element frequencies with the scalability of the block partitioning scheme used in *fgSSJoin*; thus, as reported in [80], it outperforms its two predecessors when the filtering does not make much difference, e.g., for threshold values lower than 0.5. Contrary to *fgSSJoin*, no prefix filter is employed; only the length one is used.

Bitmap filter

In their work, Mann et al. [62] point out the prefix filter as the main bottleneck of the filter-verification algorithms and underline that future filtering techniques should invest in faster and simpler methods. Driven by this, the authors of [84] propose a new low overhead filtering technique called bitmap filter. An overview of how bitmap filter operates is presented in Section 4.1. Furthermore, the authors in [84] argue that by employing a GPU for bitmap filtering leads to significant speedups.

Discussion. The simplicity and speed of bitmap filter relies on the fast bitwise operations. In order to deduce an overlap upper bound, a population count operation of the signatures' hamming distance is required, which in turn can be directly converted to the corresponding hardware instruction. Such bitwise operations can easily benefit from the massive parallel environment of the GPU. To support their reasoning, the authors of [84] describe a simple GPU implementation, in which the GPU generates candidate pairs using the bitmap filter and the CPU verifies them. As a result, significant speedup is achieved over the sequential algorithms depending on the dataset characteristics. However, their solution lacks scalability, since it involves transferring complete candidate pairs via the PCI-E bus, which becomes the main bottleneck. In the presented evaluation, the application of bitmap filter utilizing the *fgSSJoin* block partitioning scheme is investigated; contrary to the initial proposal in [84], to mitigate scalability issues, the whole join process is kept on the GPU.

5.1.2 Summary view

In Table 5.1, a unified table is provided, where all the GPU-enabled alternatives are shown with regards to their differences from the standard CPU algorithms (as described in Section 5.2). During verification, techniques are distinguished between those where multiple thread workload alternatives are employed and those that simply resort to atomic add operations in a straight-forward manner. As already stated, the evaluated techniques are main-memory ones. Therefore, the dataset must fit into CPU main memory, while the GPU's main memory must be large enough to hold the inverted index in the GPU standalone techniques.

	index	filters	verification
CPU	See Algorithm 4.1		
CPU-GPU	in CPU	as in CPU	multiple workload alternatives
gSSJoin	in GPU	-	using atomic operations
fgSSJoin	in GPU	prefix, length	using atomic operations
sf-gSSJoin	in GPU	length	using atomic operations
bitmap	-	bitmap, length	multiple workload alternatives

Table 5.1: Feature comparison between the evaluated techniques

5.2 A Comparative Evaluation

The goals of the experimental evaluation are threefold: (i) to show the extent of the achieved speedups between CPU standalone and GPU-accelerated implementations while improving on the CPU should not be taken for granted; (ii) to identify the conditions under which each solution becomes the dominant in practice, and (iii) to provide explanations about the observed behavior.

For ease of presentation, the experiments are split into two parts: the main part, which is adequate to reveal the main pros and cons of each technique, and the additional part, where the emphasis is on scalability and on using synthetic data to cover a larger range of data characteristics. The remarks of both sets of experiments are either the same or complementary and are summarized at the end.

5.2.1 Setting of the main experiments

The setting of the experiments remains the same as described in Section 5.2. Furthermore, in the experiments below, eight real world datasets are used; seven of them were also presented and employed in Section 4.1 and the TWITTER dataset found in [31] is also employed. Table 5.2 shows an overview of each dataset characteristics. Some datasets follow a Zipf-like distribution of set sizes, as shown in Figure 5.5, but in general, the distribution types differ. A summary of the TWITTER dataset is as follows:

TWITTER: social data from Twitter. Each set represents a user tweet and its set elements are character 2-grams of the respective tweet text.

In the remainder of the discussion, datasets of 10^5 will be referred to as small, the ones in the order of 10^6 medium, and those in 10^7 as large. As can be observed from Table 5.2, the datasets differ significantly in the average set size and number of distinct set elements. When the average set size is less than 10, it will be referred to as small; otherwise, as large. Also, the number of different set elements will be characterized as small when in the order of 10^4, and large when in the order of 10^6. These properties are essential to derive the sweet spots of each technique, because, along with the threshold values, they affect the amount of candidate pairs that need to be verified. For example, for the same dataset size and threshold values, fewer pairs are filtered when the number of distinct set elements is small. Similarly, any filtering becomes less effective when the average set size is small, especially when combined with low number of distinct set elements.

Dataset	Cardinality	Avg set size	# diff set elements
AOL	$1.0 \cdot 10^7$	3	$3.9 \cdot 10^6$
BMS-POS	$5.1 \cdot 10^5$	6.5	1657
DBLP (100K)	$1.0 \cdot 10^5$	88	7205
DBLP (200K)	$2.0 \cdot 10^5$	88	8817
DBLP (300K)	$3.0 \cdot 10^5$	88	$1.0 \cdot 10^4$
DBLP (1M)	$1.0 \cdot 10^6$	88	$1.5 \cdot 10^4$
DBLP (Complete)	$6.1 \cdot 10^6$	88	$2.7 \cdot 10^4$
ENRON	$2.5 \cdot 10^5$	135	$1.1 \cdot 10^6$
KOSARAK	$1.0 \cdot 10^6$	8	$4.1 \cdot 10^4$
LIVEJOURNAL	$3.1 \cdot 10^6$	36.5	$7.5 \cdot 10^6$
ORKUT	$2.7 \cdot 10^6$	120	$8.7 \cdot 10^6$
TWITTER	$1.6 \cdot 10^6$	75	$3.7 \cdot 10^4$

Table 5.2: Datasets characteristics.

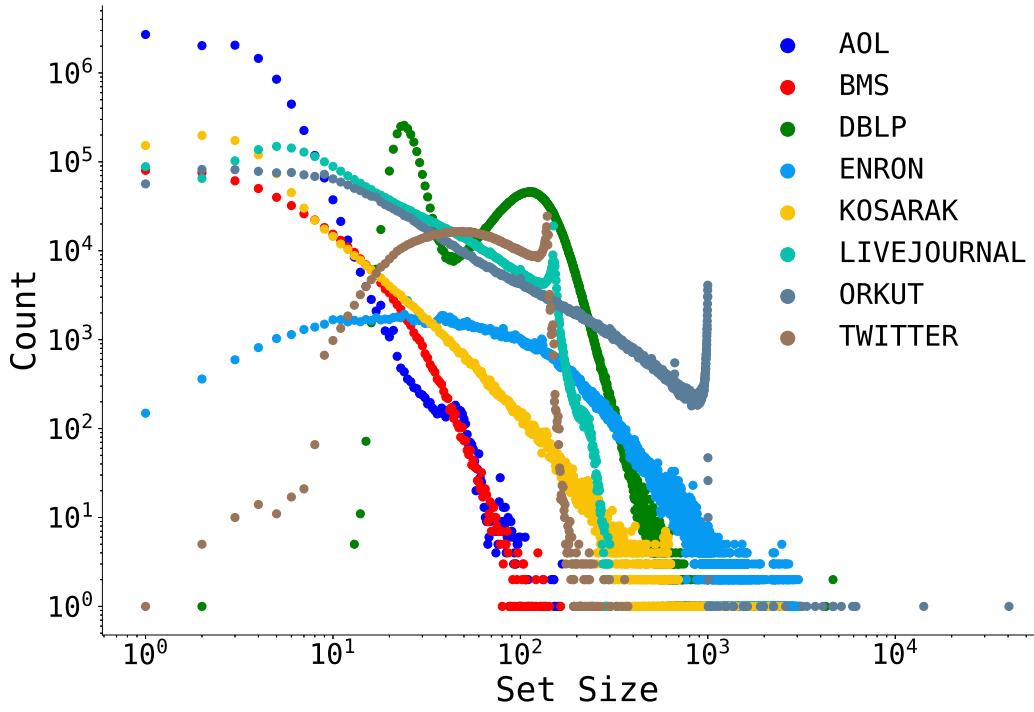


Figure 5.5: Datasets set size distribution.

Launch configuration

In GPGPU computing, selecting the best launch parameters, i.e. number of blocks and threads per block is required to fully exploit the massive parallelism. Although recent compiler optimizations tend to offload the developer from the burden to meticulously pick these parameters, depending on the context and the GPU code, it may be inevitable for these not to be *hand picked*.

For the *CPU-GPU* co-processing technique presented in Section 4.2, with fine tuning[2] and by selecting the appropriate workload allocation between threads, depending on each dataset characteristics, fully overlap between filtering, verification and the final phase, where the CPU constructs the result pairs, is achieved. For *gSSJoin* and its variations, a fixed number of thread blocks (equal to the number of SMs) and threads per block (max supported value) is used, as proposed by their authors. Specifically, a 1D grid consisting of 15 thread blocks and 1024 threads per thread block is launched. In addition, the input collection is chosen to be partitioned into blocks of size $n = 15000$.

5.2.2 Main experiments

The main experiments consist of the comparison between the state-of-the-art CPU standalone implementation of [62] against its GPU-accelerated version presented in Section 4.2, noted as *CPU-GPU* and the GPU standalone solutions described in [77, 80, 81]. In Figure 5.6, the best join times measured for all are presented. Each time reported for the CPU is the overall best among the three algorithms (i.e., ALL, PPJ, GRP) and therefore the best than can achieved in the evaluation setup. Respectively, for the *CPU-GPU* co-processing solution, each time reported is the overall best among of the three CPU algorithms and the best GPU verification techniques described in Section 4.2. Finally, for *gSSJoin* and its variations, the sum of time for all the GPU operations required to perform the set similarity join is reported.

As shown in Figure 5.6, for the majority of datasets and high thresholds, i.e. the threshold range is in [0.8, 0.9], invoking the GPU does not yield any performance speedup. Given the fact that in high thresholds, filtering is quite effective, hence the number of candidates is quite small, employing the GPU seems redundant, especially for small datasets. On the other hand, as the threshold value decreases ([0.5, 0.7]), GPU standalone solutions are quite effective in general. This is due to the fast intersection count conducted in parallel which accelerates the verification phase, but as explained later, there are several other factors that impact on performance.

Nevertheless, there is no clear pattern. Continuing with the GPU standalone solutions, it is observed that are rather inefficient for the biggest dataset (AOL); in contrast, they perform better for the TWITTER dataset. This calls for a deeper analysis. The detailed observations with the key strengths and weaknesses of each technique are deferred to Section 5.2.5 and after the technique behavior is explained in more detail. Also, the *gSSJoin* solution has been omitted, due to its inability to scale, as explained below.

[2] Fine-tuning for this approach is only required for the small datasets where, depending on certain key dataset characteristics, the GPU verification may add an extra overhead. As the dataset size increases, the overall join time is bounded by the CPU filtering time. Hence, fine-tuning the GPU in such cases would not yield any performance speedup.

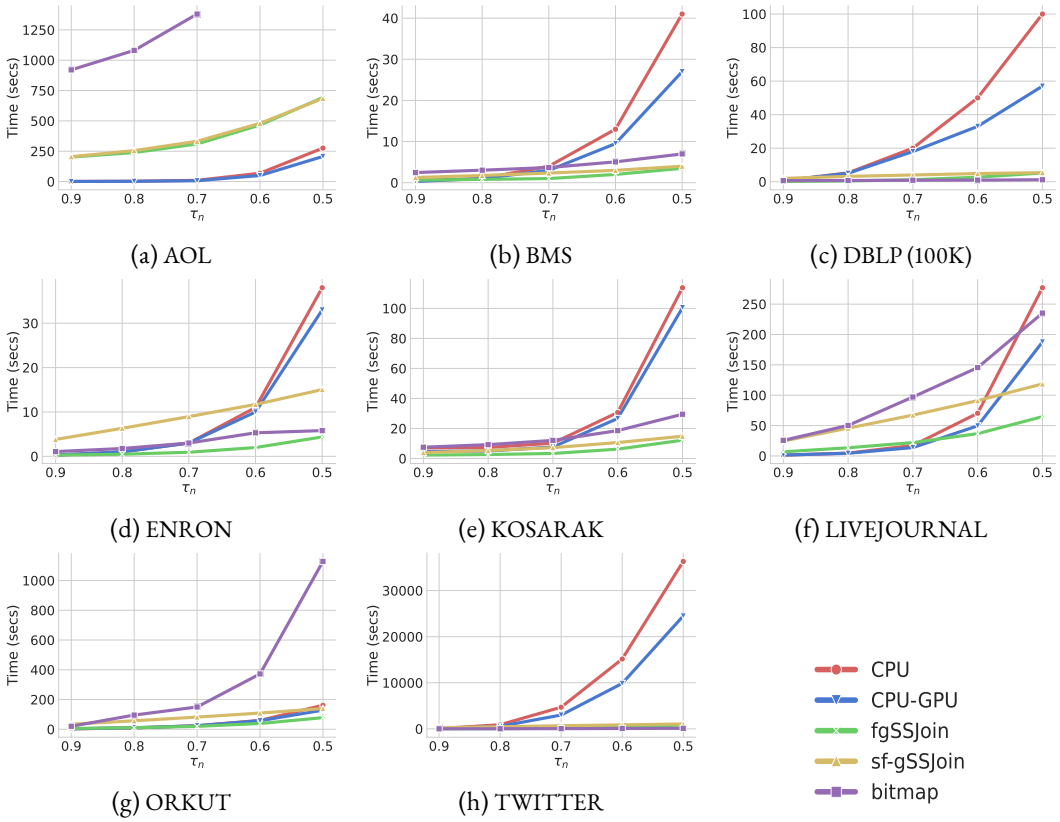


Figure 5.6: Comparison between the best times for different thresholds.

5.2.3 Performance analysis

Motivated by the fact that neither technique is the most dominant one (as also shown in Figure 5.6), first the impact of the fact that the GPU standalone solutions require quadratic space in the global memory is investigated. The effect of dropping the prefix filter in *sf-gSSJoin* is also discussed. Further, the use of bitmap filter, instead of prefix filter used in *fgSSJoin*, is evaluated, and useful insights are provided. In addition, the omission of *gSSJoin* from the main experiments is explained. Another aspect of the analysis, involves the evaluation of the scalability of all solutions by running them over artificially increased datasets. Finally, experimentation on synthetic datasets is also investigated to further highlight the impact of specific dataset characteristics. The two latter aspects, due to their significance, are examined separately.

Quadratic space overhead of *fgSSJoin* and *sf-gSSJoin*

For datasets, the output of which does not fit in the GPU memory, *fgSSJoin* and *sf-gSSJoin* use a block division scheme to process input data gradually. To conduct a set similarity join between two blocks, an $O(n^2)$ memory space is required to store the intersection counts. After each block probe, this memory space must be cleared to ensure correctness for the next probe calls.

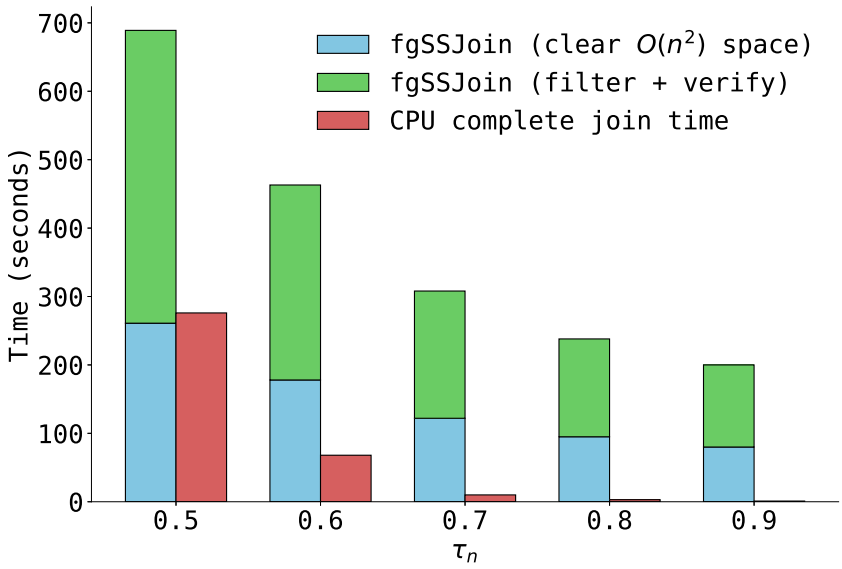


Figure 5.7: Quadratic space overhead.

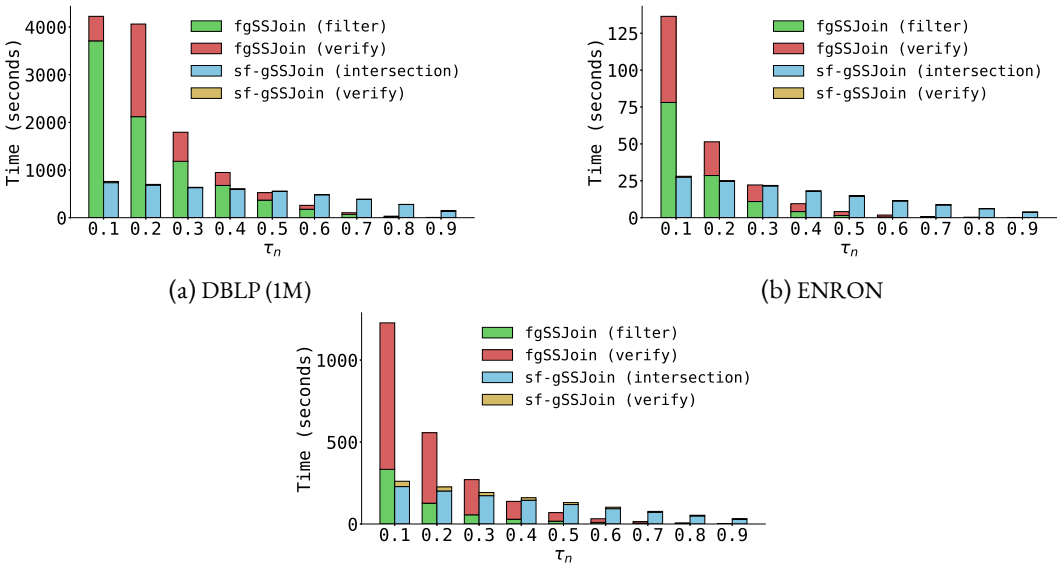
τ_n	# probes	accumulated $O(n^2)$ space	time (secs)
0.5	138841	125 TB	261
0.6	94747	85 TB	178
0.7	64928	58 TB	122
0.8	50431	45 TB	95
0.9	42628	38 TB	80

Table 5.3: Quadratic space overhead for AOL.

Considering that an intersection count is a 4-byte integer, the required memory space to store all counts for $n = 15000$ is 900MB. In the experiments, the `cudaMemset` function is used to clear the intersection count space. This entails a 2 ms overhead per probe call.

For datasets with a high proportion of similar set sizes, such as AOL, length filtering on block level is ineffective. This results in a higher number of GPU calls, which in turn increases the clear operation overhead of the quadratic memory space. Concrete numbers are presented in Figure 5.7 and Table 5.3. The magnitude of the impact of quadratic space on runtime is dictated by two factors: input collection size and the number of pruned blocks.

The bottom line is that in large datasets, especially when, due to the dataset size and set size, the length filter cannot prune many pairs, the overhead associated with the quadratic space complexity is not outweighed by any benefits compared to the CPU solution for thresholds above 0.6, and, overall *fgSSJoin* and *sf-gSSJoin* are not the optimal GPU-enabled techniques.



(a) DBLP (1M)

(b) ENRON

(c) ORKUT

Figure 5.8: Comparison between the best times of *fgSSJoin* vs *sf-gSSJoin* for different thresholds.

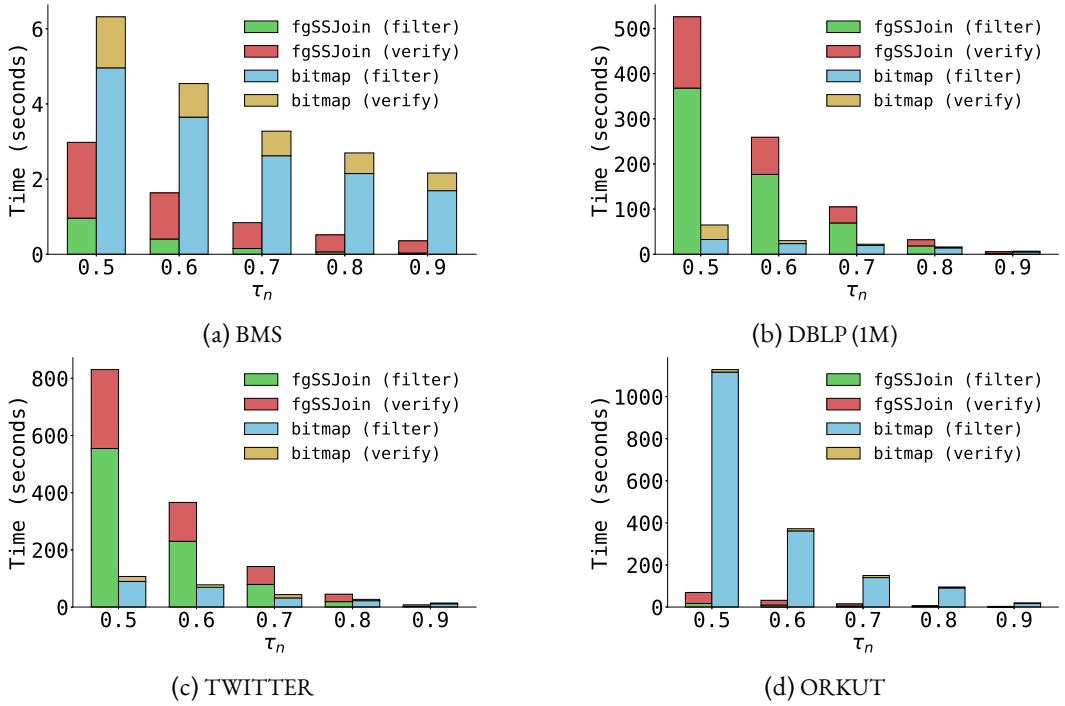
Filtering vs complete intersection (fgSSJoin vs sf-gSSJoin)

Assume that someone would like to run a similarity query that is rather close to a cartesian product by setting the threshold to low values, such as $\tau_n < 0.5$. In this case, adopting and relying on effective filtering is not beneficial. This renders *sf-gSSJoin*, which is closer to brute-force, the best performing technique. In general, *sf-gSSJoin* exhibits robustness with regards to low threshold values. While it may perform worse than *fgSSJoin*, it scales better when the threshold value is gradually lowered. Figure 5.8 shows concrete examples for the DBLP(1M), ENRON and ORKUT datasets, and complements the relevant results from Figure 5.11.

Bitmap performance

Bitmap filtering can be seen as an alternative to a prefix-based approach. However, the technique is highly dependent on the dataset characteristics, since sets may produce similar bitmaps even if they do not share similar set elements; in general, the probability of such undesired collisions is increased whenever the number of set elements is increased and the bitmap size is reduced. Trying to increase the bitmap size along with the number of set elements does not scale either because it leads to more global memory accesses. Nonetheless, the *bitmap*-based solution has its own sweet spot: high average set size combined with low number of different set element, as in the TWITTER and DBLP datasets. Figure 5.6 shows the best timings of the bitmap solution for the main experiments (among the several signature sizes that were used).

For the AOL dataset, bitmap has the worst performance because it inherits the quadratic space overhead of *fgSSJoin* and its filtering is ineffective. For the BMS, ENRON and KOSARAK datasets and high threshold values, bitmap performs worse than every other technique. However, as the threshold value decreases, the performance difference between the best performing



(a) BMS

(b) DBLP (1M)

(c) TWITTER

(d) ORKUT

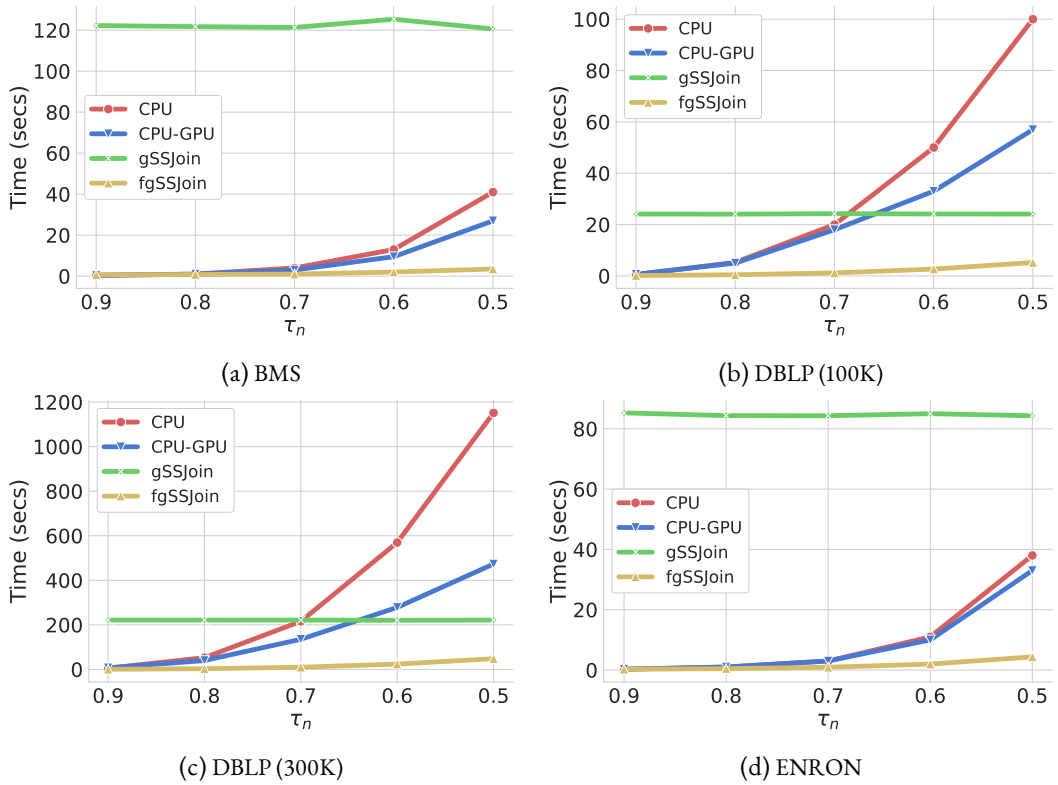
Figure 5.9: Comparison between the best times *fgSSJoin* vs *bitmap* for different thresholds.

techniques and bitmap is smaller. The best bitmap performance is observed for the DBLP(100K) and TWITTTER datasets. Furthermore, in Figure 5.11 to be discussed later in detail, it is shown that bitmap remains efficient for larger portions of the DBLP dataset. On the other hand, for the LIVEJOURNAL and ORKUT datasets, bitmap has the worst performance for the majority of threshold values.

Finally, the impact of the average set size and the number of different set elements on the efficiency of bitmap filter is shown in Figure 5.9. The BMS, DBLP (1M) and the TWITTER datasets have small number of different set elements. However, BMS, contrary to DBLP and TWITTER has small average set size, which leads to a big increase in the filtering time. For ORKUT, even though it has a large average size, due to its large number of different set elements, it renders bitmap filtering ineffective (the same behavior was also observed for the LIVEJOURNAL dataset).

gSSJoin launch overhead

As stated in Section 5.1, *gSSJoin* launches a sequence of separate GPU kernel calls per probe set in order to conduct the set similarity join operation. Given the fact that launching a large number of small kernels is considered a bad practice, executing *gSSJoin* results into an accumulated launch and execution overhead as shown in Figure 5.10. Hence, the overall runtimes are dominated by the overhead required to invoke the GPU and it is threshold value independent. In some cases, such as the DBLP datasets (Figures 5.10b and 5.10c), *gSSJoin* performs better than the CPU at lower thresholds since the GPU invocation overhead is lower than the filtering phase conducted on the



(a) BMS

(b) DBLP (100K)

(c) DBLP (300K)

(d) ENRON

Figure 5.10: gSSJoin runtimes for different thresholds compared to other techniques.

Dataset	Cardinality	Avg set size	# diff set elements
BMS (x25)	$1.3 \cdot 10^7$	6.5	1681
ENRON (x25)	$6.1 \cdot 10^6$	135	$1.1 \cdot 10^6$
LIVEJOURNAL (x5)	$1.5 \cdot 10^7$	36.5	$7.5 \cdot 10^6$

Table 5.4: Larger datasets' characteristics. Within parentheses is the increase factor.

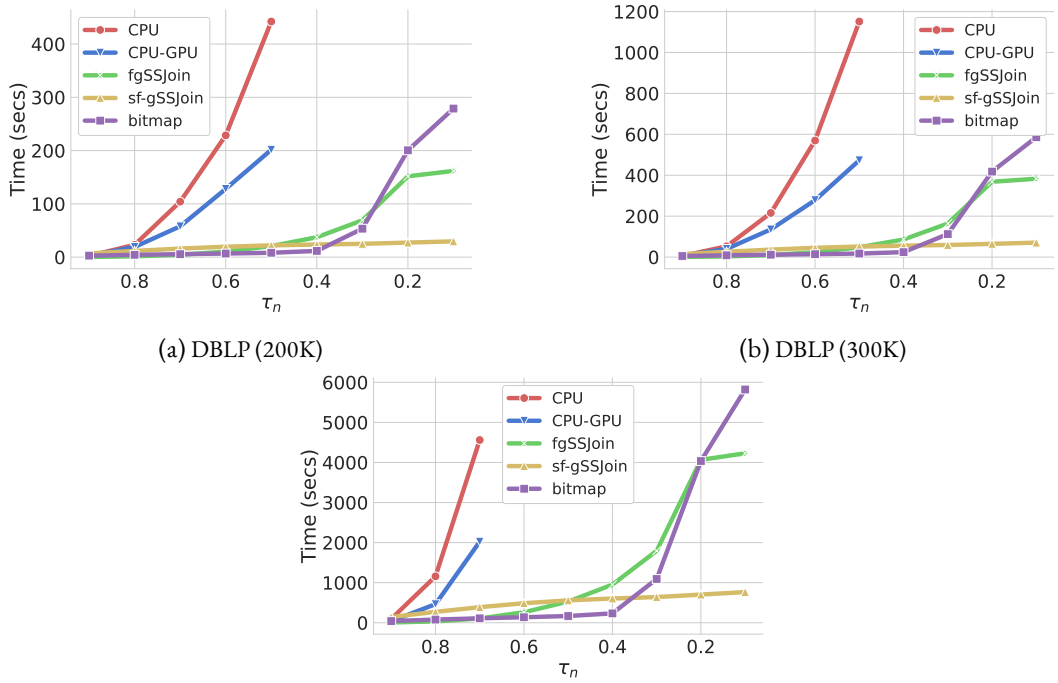
CPU. However, as the input collection size increases the GPU overhead increases dramatically, rendering *gSSJoin* inefficient for real world applications and is totally dominated by *sf-gSSJoin*.

5.2.4 Additional Experiments

Before presenting the main outcomes, additional experiments using larger and synthetic datasets are conducted.

Scalability

The scalability of the evaluated solutions is examined (i) on larger portions of the DBLP dataset, which follow the same set element distribution frequency and, (ii) on artificially increased ver-



(a) DBLP (200K)

(b) DBLP (300K)

(c) DBLP (1M)

Figure 5.11: Comparison between the best times for larger portions of the DBLP dataset.

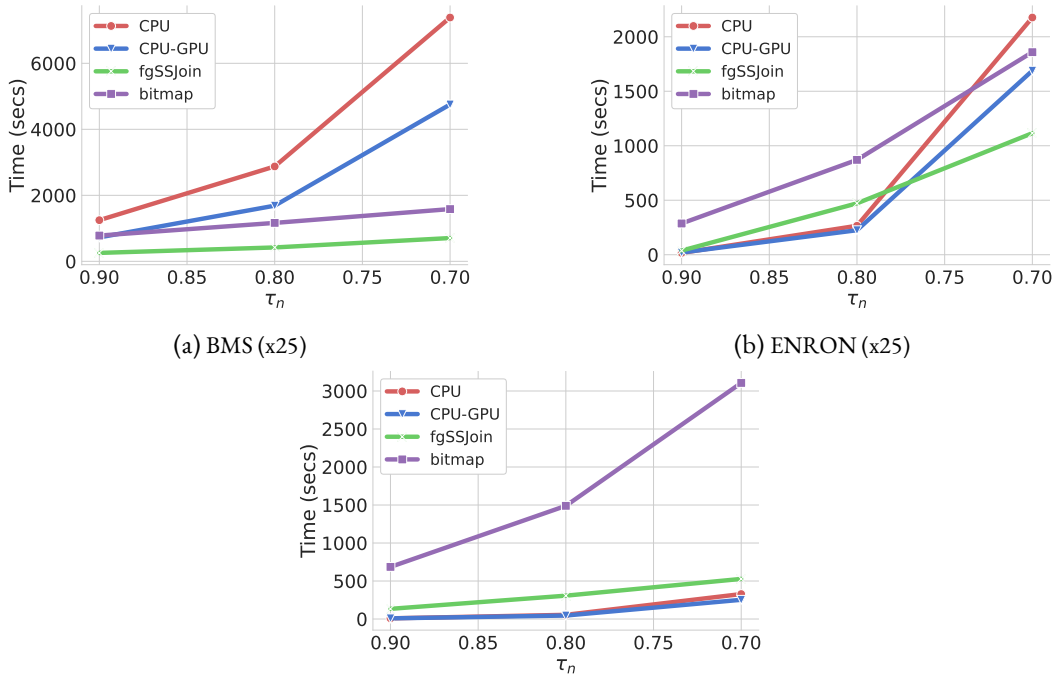
sions of the BMS, ENRON and LIVEJOURNAL datasets. The increased datasets are populated similarly to [92]. All the corresponding dataset characteristics are located in Table 5.4.

From Figure 5.11, it can been seen that the speed-ups of the GPU standalone solutions are much more evident as the collection size increases compared to the CPU standalone for larger portions of the DBLP dataset. This is further verified from partial results regarding the full DBLP dataset, as shown in Table 5.5. The benefits from employing GPUs are tangible even for larger thresholds, given that, for the complete dataset, the candidate pairs are tens of billions and CPU takes up to two hours approximately for DBLP dataset with $\tau_n = 0.9$.

However, the correct interpretation needs to take into account the behavior for the AOL dataset, where the GPU standalone solutions are suboptimal. The key observation is that in Figure 5.11[3], the GPU standalone solutions perform very well not due to their capability to scale for large datasets, but due to the length filter they encapsulate. This type of filter is particularly effective for the DBLP dataset but not for the AOL. The important lesson learned is that dataset characteristics should be always taken into account in all their main dimensions (dataset size, average set size and number of different set elements).

Consequently, the best performing techniques are evaluated on artificially increased datasets. The threshold range is also narrowed to $\tau_n \in [0.7 - 0.9]$ due to the large overall join times. Depending on the index size, it may not be feasible to increase datasets arbitrarily. For example,

[3]When $\tau_n = 0$, the similarity join is essentially a cartesian product, where the only practical solution is *sf-gSSJoin* with running time as for $\tau_n = 0.1$.



(a) BMS (x25)

(b) ENRON (x25)

(c) LIVEJOURNAL (x5)

Figure 5.12: Comparison between the best times for the increased datasets.

the ORKUT dataset could be increased x10 times by following the method used in [92]. However, this leads to a 13GB index size, which cannot fit in the GPU memory. Thus, two small datasets are selectively chosen, i.e. BMS and ENRON, one medium dataset, i.e. LIVEJOURNAL, and each is increased by an appropriate factor in order to comply with the main-memory restriction.

As it can be seen in the Figure 5.12a, increasing the BMS dataset does not yield any performance difference between the techniques compared to the original dataset. This is mainly due to the combination of the small number of different set elements and the small average set size, which results in a high number of frequent set elements. As a result, this benefits *fgSSJoin* to conduct filtering and intersection count faster.

In contrast, the *CPU* and *CPU-GPU* techniques perform better than *fgSSJoin* on the increased ENRON dataset for $\tau_n = 0.8$ as shown in Figure 5.12b. Because of the large number of different set elements coupled with the large average set size, there is a high number of infrequent set elements, which favors prefix filtering. While the prefix filter remains effective, CPU-based filtering techniques also remain competitive. For the same reasons, the performance difference between the CPU-based filtering techniques and *fgSSJoin* becomes more evident in the increased LIVEJOURNAL dataset (Figure 5.12c). However, as the threshold value is decreased for the ENRON dataset, prefix filtering becomes ineffective, leading *fgSSJoin* to perform better (Figure 5.12b, $\tau_n = 0.7$).

τ_n	CPU	CPU-GPU	fgSSJoin	bitmap	sf-gSSJoin
0.8	-	32387	1259	607	10353
0.9	7244	2814	230	303	5227

Table 5.5: Runtimes for the complete DBLP consisting of 6.1M sets (in secs).

Dataset size	5M, 10M, 20M
Number of different set elements	50K, 500K
Average set size	5, 25

Table 5.6: Synthetic datasets' characteristics.

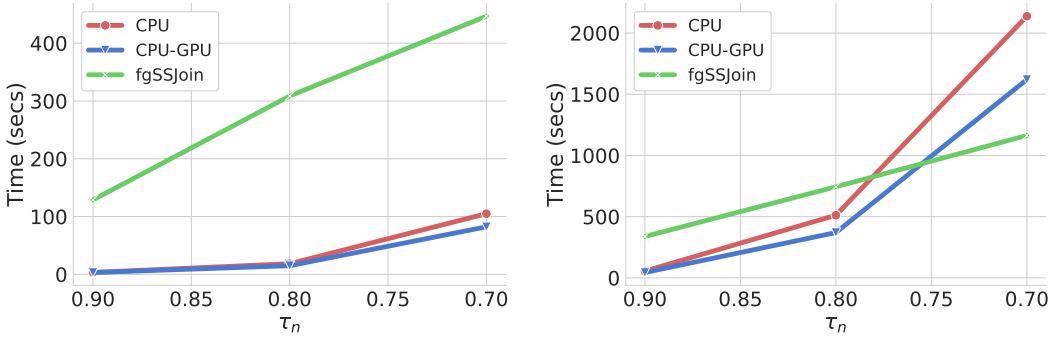
Evaluation on synthetic datasets

Three key dataset characteristics are highlighted: (i) dataset size, (ii) average set size, and (iii) the number of different set elements. In order to evaluate the impact of each of these factors in a more controlled manner, twelve synthetic datasets using the combination of characteristics listed in Table 5.6 are created[4]. Furthermore, the synthetic datasets follow a Zipf-like set element frequency distribution. To distinguish each dataset, the notation *Dataset Size - Number of different set elements - Average set size* is used.

Initially, the dataset size is kept fixed to 10M and the values of the other two characteristics are varied as shown in Figure 5.13. In the same manner, the number of different set elements is kept fixed to 500K (Figure 5.14) and the average set size is kept fixed to 25 (Figure 5.15). In all of the synthetic datasets, two behavioral patterns were observed. As such, the runtimes can be classified into two categories, (i) those where the CPU-based filtering outperforms *fgSSJoin* for every examined threshold value and, (ii) those in which *fgSSJoin* begins to perform better for $\tau_n = 0.7$.

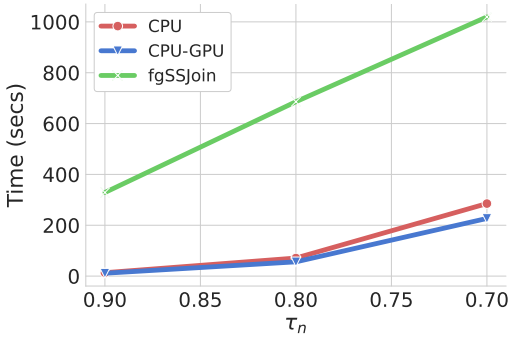
It can be seen that CPU-based solutions are constantly more efficient than *fgSSJoin* apart from the cases where the number of distinct set elements is not large and the average set size is large. More specifically, in cases where the number of different set elements is 500K, there is a high number of infrequent set elements, which benefits CPU-based filtering. For lower number of distinct set elements, prefix filtering gradually becomes ineffective, and thus *fgSSJoin* performs better, as previously stated in the Section 5.2.4.

In conclusion, the combination of the three main dataset characteristics, namely dataset size, number of distinct set elements and average set size, directly impacts on the prefix filter efficiency, and consequently on which technique is the best-performing one. A small number of different set elements alongside a large average set size will result also in more frequent set elements, which does not allow prefix filtering to prune a lot of candidate pairs. On the contrary, for larger datasets with a high number of different set elements, it is more probable to have infrequent set elements.



(a) 10M - 50K - 5

(b) 10M - 50K - 25



(c) 10M - 500K - 25

Figure 5.13: Comparison between the best times for synthetic datasets with fixed dataset size.

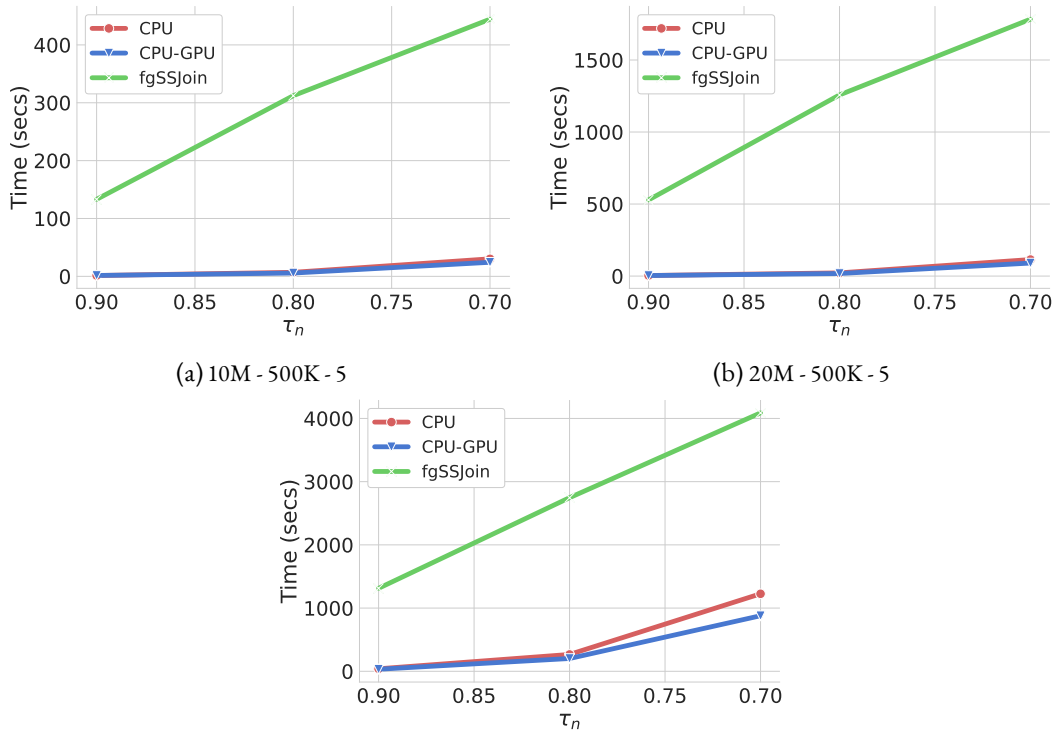
5.2.5 Summary of Remarks

The strong and weak points of each technique are summarized below. In the discussion, threshold values of 0.9 are considered as very high, 0.8 as high, 0.5-0.7 as medium and less than 0.5 as low.

CPU:
Strong points: handling high and very high thresholds where (prefix) filtering is effective, e.g., no small average set size combined with high set element cardinality.
Dominating cases: very high thresholds unless small dataset and small average set size, where moving the dataset to the GPU and perform extremely quick analysis there is more beneficial.
Weak points: handling threshold values lower than 0.8.

CPU-GPU:
Strong points: handling large datasets (but cannot scale with decreasing threshold values).
Dominating cases: medium-high thresholds ($0.5 < \tau_n < 0.8$) and large datasets.
Weak points: handling (i) small thresholds; and (ii) medium thresholds and no large datasets, because in these cases, the filtering phase (not parallelized by CPU-GPU) dominates and/or is sig-

[4] The original scripts provided by the authors of [62] were used.



(a) 10M - 500K - 5

(b) 20M - 500K - 5

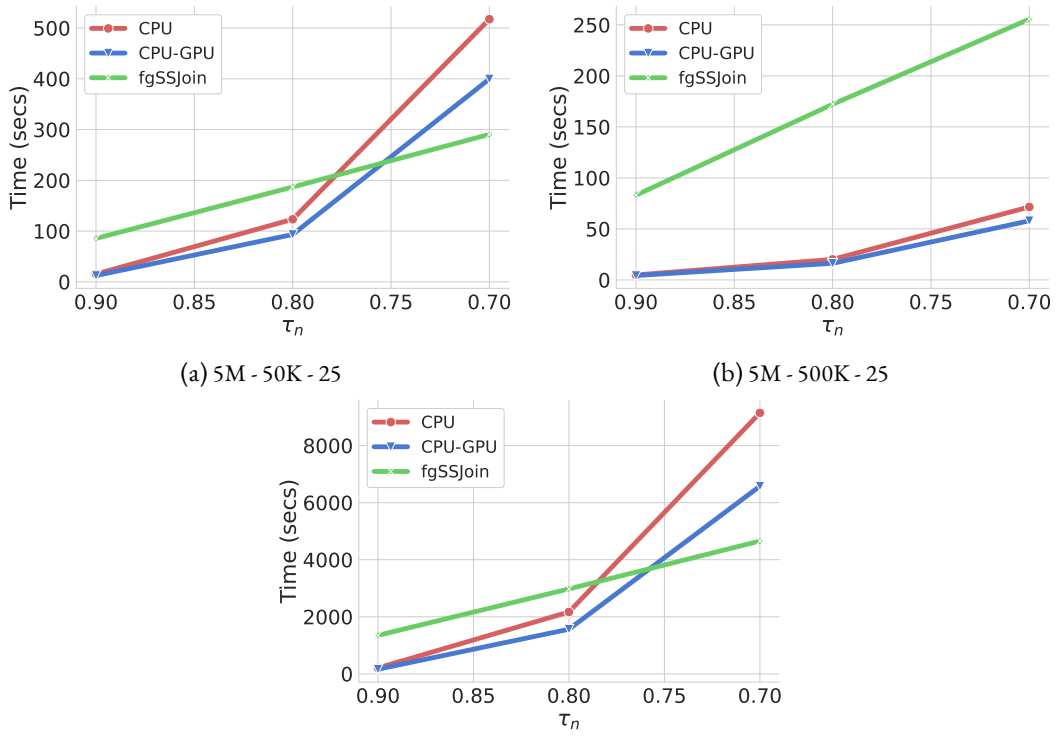
(c) 20M - 500K - 25

Figure 5.14: Comparison between the best times for synthetic datasets with fixed number of different set elements.

nificant.

fgSSJoin:
Strong points:handling medium and high thresholds ($0.5 < \tau_n < 0.8$) but not large datasets, where the initial index-based prefix filtering is ineffective.
Dominating cases: Same as the cases in the strong points.
Weak points: handling large datasets due to the quadratic complexity in the block size and the associated overhead.

sf-gSSJoin:
Strong points:handling not very big datasets combined with low thresholds.
Dominating cases: threshold below 0.5, where sophisticated filtering, e.g., prefix ones, is not effective.
Weak points: handling the cases, where prefix-based filtering can manage to prune a significant portion of candidate pairs, e.g., queries with high thresholds, especially when combined with high average set size.



(a) 5M - 50K - 25

(b) 5M - 500K - 25

(c) 20M - 50K - 25

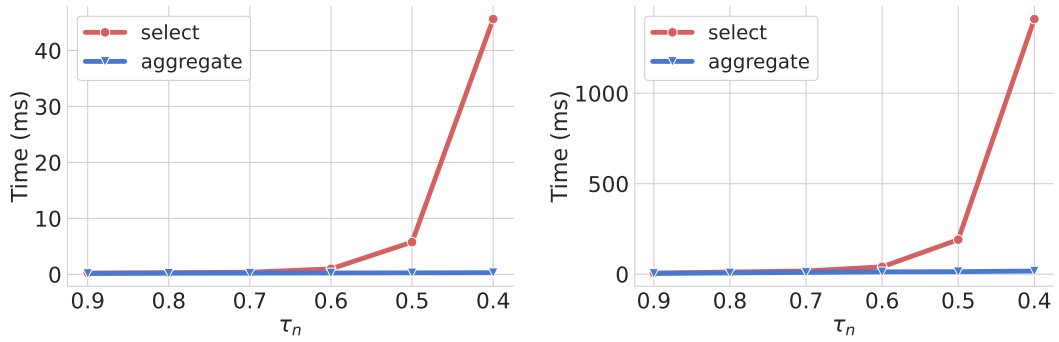
Figure 5.15: Comparison between the best times for synthetic datasets with fixed average set size.

τ_n	Dataset size - Average set size				
	small - small	small - large	medium - large	large - small	large - large
Very high(0.9)	fgSSJoin CPU	CPU	fgSSJoin CPU	CPU	CPU CPU-GPU
High(0.8)	fgSSJoin	fgSSJoin	fgSSJoin CPU-GPU	CPU-GPU	CPU-GPU
Medium(0.5-0.7)	fgSSJoin	fgSSJoin	fgSSJoin bitmap	CPU-GPU	fgSSJoin CPU-GPU
Low(<0.5)	sf-gSSJoin	sf-gSSJoin	sf-gSSJoin	N/A	N/A

Result size	Dataset size - Average set size				
	small - small	small - large	medium - large	large - small	large - large
$< 10^4$	N/A	fgSSJoin CPU	fgSSJoin CPU	CPU	CPU CPU-GPU
$[10^4 - 10^6]$	N/A	fgSSJoin	fgSSJoin CPU-GPU	CPU-GPU	CPU-GPU
$[10^7 - 10^8]$	fgSSJoin CPU	fgSSJoin	fgSSJoin bitmap	CPU-GPU	CPU-GPU
10^9	sf-gSSJoin	sf-gSSJoin	sf-gSSJoin	N/A	fgSSJoin CPU-GPU
$> 10^9$	sf-gSSJoin	sf-gSSJoin	sf-gSSJoin	N/A	N/A

Figure 5.16: Summary of the best techniques per scenario examined.

bitmap:
Strong points:handling cases where bitmap signatures are effective, i.e., high set size and not high number of different set elements.
Dominating cases: medium thresholds and dataset size combined with low number of different set elements and high average set size.
Weak points: scalability in the dataset size.



(a) DBLP (100K)

(b) DBLP (1M)

Figure 5.17: Transfer times via PCI-E for the select (pairs) and aggregate (counts) queries for the *fgSSJoin*,*sfgSSJoin* and *bitmap* techniques.

In Figure 5.16, an overview of the cases, where each technique is the dominant one, is provided based on the experiments. When there are two techniques for the same combination of dataset size and average set size, the distinct set element cardinality makes the difference. The top part of the figure splits the rows according to the τ_n values. The bottom part repeats the table with rows split according to the result size. As it can be seen, the contents are highly correlated, thus the discussion above based on threshold values can be rephrased to be based on result sizes.

5.2.6 Output consideration

In the presented experiments, the set similarity results contained only the count for the GPU-enabled solutions. However, when considering outputting the complete results, there are negligible changes. More specifically, *CPU-GPU* manages to completely hide the final output construction due to the thread overlapping as previously discussed in Section 4.4. The rest of the techniques share the same approach to constructing the final result pairs in the GPU and then counting them. However, the main bottleneck is to transfer the results back to the CPU. In Figure 5.17, the overhead for the DBLP dataset is shown. For instance, in a challenging case where the dataset is 1M, $\tau_n = 0.4$ and the number of output pairs is $1.3 \cdot 10^9$, *fgSSJoin* transfers back to the main memory a tuple consisting of the pair and its similarity degree for each result item. Thus, each tuple has a size of 12 bytes, which amounts to roughly 15.6 GB for the complete output. This can be transferred in less than 1.5 second through the slow PCI-E CPU-GPU connection, which is negligible compared to the join times reported in Figure 5.11.

5.3 Discussion

The main techniques that have been proposed in the last few years have different characteristics, which supports the hypothesis that GPU-enabled set similarity joins is still a technology in evolution. More importantly, there is no clear winner, which leaves the question as to whether a globally dominant solution exists open. In Section 5.2.5, the key strengths and sweet spots of

each technique are summarized. Below, additional generic observations are summarized, lessons learned and technical challenges encountered:

1. The presented evaluation aims to extend comprehensive evaluations, such as these in [62], where a single physical machine is employed, through allowing computations to be also performed on a single GPU. It is assumed that the initial datasets are up to several GBs in size, so that they can fit into the RAM. In such a setting, judiciously employing the GPGPU paradigm can lead to speed-ups up to two orders of magnitude, e.g., 339X for the Twitter dataset when $\tau = 0.5$ (see Figure 5.6). Judiciously relates to the decision about which technique to employ according to the summary table in Figure 5.16, which may suggest not to use GPU at all. However, this is the exception; despite working with datasets fitting into the main memory of a modern machine, GPU outperforms CPU-only solutions, especially when $\tau_n < 0.9$.

2. A main contribution of the presented evaluation is to reveal and describe the strengths and weaknesses of each technique. A prerequisite is to identify the main factors that impact on the relative performance. The dataset size, set element frequency distributions, number of different set elements, average set sizes and τ_n parameter (related to the result size) have been identified as such factors, with the dataset size, average set size and τ_n parameter being the most important ones. At a high-level, these parameter directly impact on the effectiveness of the filter types employed by the different techniques.

3. From the technical perspective, the most challenging issues are the design choices, which concern the kernel grid and balanced splitting of the workload across GPU threads. Filtering and verification using the intersection count technique, share common data, and as such, they are tightly coupled. Therefore, they cannot be regarded as individual operations and must be implemented taking into account their interactions. Based on how data is accessed and the workload of each thread, certain GPU features, such as shared memory, that otherwise could speed up the join process, may not be possible to exploit to its full extent. For example, *fgSSJoin* splits the workload among threads based on the static inverted index, which leaves global memory atomic operations as the only alternative to process filtering and intersection count. As a result, the fast on-chip shared memory is not utilized.

4. Elaborating on the technical issues, brute-force-like solutions, such as *gSSJoin* cannot scale because of the high launch overhead, which is associated with the input dataset size. More sophisticated solutions, such as *fgSSJoin* and *bitmap* adopt a block partitioning scheme that enables them to process thousands of sets per GPU invocation. However, their current limitations, which are generic in the GPGPU programming, concern (i) memory consistency and (ii) kernel invocation.

    - Regarding memory consistency, for *fgSSJoin* to ensure correctness, prefix filtering is conducted via atomic operations on global memory. This may lead to a performance degradation in cases where multiple threads must update a specific global memory address. For *bitmap*, as the signature size increases, the global memory access footprint also increases and eventually dominates the filtering phase.

Parameter	Description
$R = \{r_1, \ldots, r_{\lvert R \rvert}\}$	A collection of records
$E = \{e_1, \ldots, e_{\lvert E \rvert}\}$	Element universe set
$R_i = R_1^i \cup \cdots \cup R_{\lvert R_i \rvert}^i$	A partition of blocks
R_j^i	The j-th block of the i-th partition
τ_n	Normalized threshold
n	Block size
p	Number of partitions

Table 5.7: Notation summary.

- Regarding kernel invocation, for the GPU-standalone techniques, a GPU invocation consists mainly of two kernels, a filtering and a verification kernel. Essentially, the filtering kernel produces candidate pairs for the verification kernel to consume and verify in order to output the final result. Due to the GPGPU programming nature, a kernel cannot consume any portion of data produced by a prior kernel until the latter runs to completion. This has an impact on resource utilization and consequently on performance since a thread block that has finished the filtering phase waits for every block of the grid to finish. Recent advancements on GPGPU programming such as multi-block cooperative groups which enable collective operations, such as verification, to work across all threads in a group, may contribute to alleviate this issue, but this remains an open issue.

5.4 HYSET: A HYBRID SET SIMILARITY JOIN FRAMEWORK

The findings of the empirical evaluation of GPU-enabled techniques presented in Sections 5.2 and 5.3, alongside the inability of other parallel paradigms such as MapReduce [28] to the problem, provide compelling evidence that a hybrid GPU-enabled approach is indeed viable for the set similarity join problem.

In this section, the proposed hybrid framework is presented. With the exception of the cooperative CPU-GPU technique, every other technique works as standalone on either the CPU or the GPU. Consequently, this results to inactivity of the other end. The main goal of the proposed framework is to minimize this inactivity by splitting the join workload in smaller chunks and utilize both ends efficiently.

First, the partitioning scheme is introduced which leverages two existing GPU-based techniques; the one proposed in [77] and the one presented in Section 4.2, and combines them with CPU-based solutions. Next, two workload allocation strategies to utilize both the CPU and GPU are proposed. Finally, a concise overview of the framework's pipeline is given along further details on the implementation.

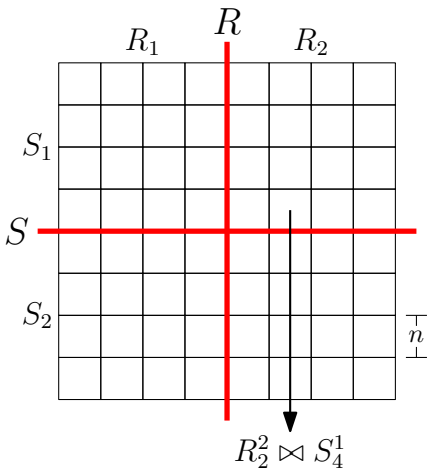


Figure 5.18: Partitioning scheme, each both input collections are divided into partitions ($p = 2$) that are composed of blocks of size n.

5.4.1 Partitioning Scheme

CPU-standalone techniques process the complete join in a loop-style fashion where each iteration outputs similar sets for a specific set. On the other hand, GPU-standalone techniques process the complete join in batches due to the GPU's limited memory space and inefficiency in dynamic memory allocation. Therefore, a two-level partitioning scheme is developed that supports both.

The two-level partitioning scheme is composed of two type of segments: (i) block, and (ii) partition. On the first level, input collections are divided into blocks of size n so that the required $O(n^2)$ memory space fits in the GPU. On the second higher level, input collections are divided into p partitions. Essentially, a partition R_i is a segment consisting of a sequence of blocks $(R_0^i, R_1^i \ldots, R_{|R_i|}^i)$. Figure 5.18 illustrates the partitioning scheme. The frequently used notations are summarized in Table 5.7.

5.4.2 Workload Allocation Strategies

In order to utilize both ends concurrently and efficiently, workload allocation is of paramount importance. Based on the partitioning scheme, the GPU supports only joins between blocks, whereas the CPU supports also joins between partitions. In respect to this, two workload allocation strategies are developed. Each strategy is described below.

Concurrent Queue The most evident approach to keep both the CPU and GPU utilized is by splitting the complete join workload into smaller joins and have each end, as soon as it becomes available, process a portion of them. Hence, the join matrix is split into pairs of blocks which are used to populate a concurrent lock-free queue. Thus, both ends can run concurrently and independently until the queue is empty. An overview of the concurrent queue strategy is depicted in Figure 5.19.

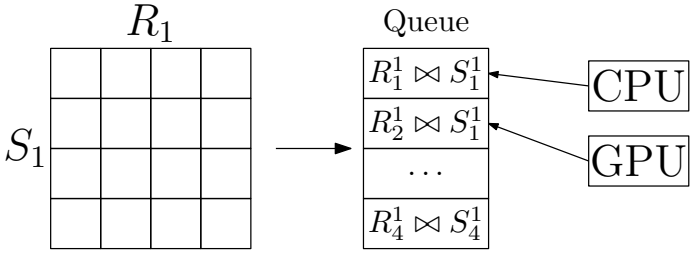


Figure 5.19: Concurrent queue workload allocation strategy.

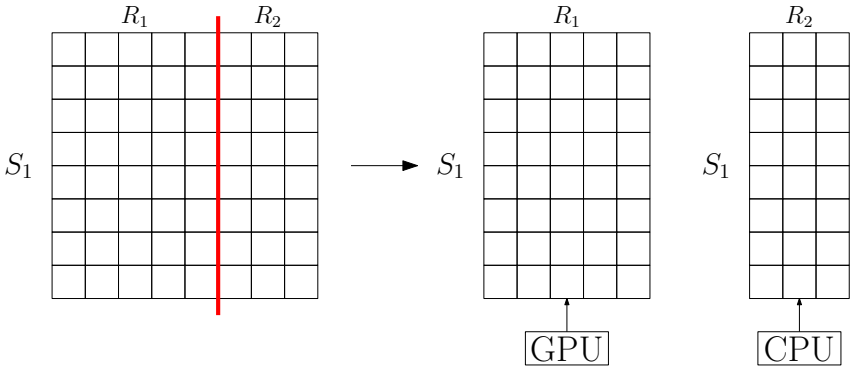


Figure 5.20: Dichotomy workload allocation strategy.

		CPU	GPU	CPU-GPU
main	prefix	✓	✓	✓
	bitmap		✓	
auxiliary	length	✓	✓	✓
	positional	✓	✓	✓

Table 5.8: An overview of the applicable filters.

Dichotomy Another approach is by dichotomizing the complete join workload. To achieve this, the probe collection is split into two partitions ($p = 2$), i.e. two separate joins between partitions, and assign each to either the CPU or the GPU. For the GPU, the partition join is decomposed to joins between blocks. Figure 5.20 illustrates the dichotomy work allocation strategy.

5.4.3 Overview and Technique Selection

Since there is no dominant solution for the set similarity join problem as shown in Section 5.3 and 5.3, the framework must encapsulate the best performing techniques. Three categories of available techniques are enumerated, (i) CPU-standalone and (ii) GPU-standalone, where the complete join is conducted exclusively on either side, and (iii) cooperative CPU-GPU where filtering is conducted on the CPU and verification on the GPU.

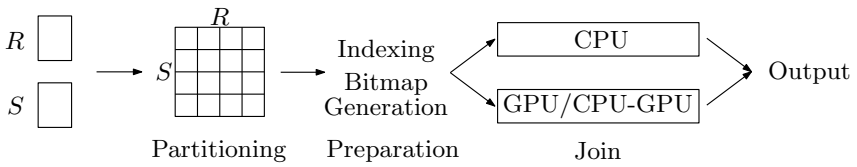


Figure 5.21: Framework pipeline.

The applicable filters are categorized per category of techniques into (i) *main*, i.e. those which require an external structure (either an index or bitmap signatures), and (ii) *auxiliary*, i.e. those which exploit only set size to determine if the required overlap can be met. Table 5.8 summarizes the filters used in the framework. Bitmap can also be used as a main filter for the CPU-enabled techniques alongside prefix filter. However, the speedup gains in this case for the best performing CPU techniques are relatively small, up to 1.35x on average [84]. In addition, length filter can also be used at block level, which enables pruning of whole blocks.

Figure 5.21 depicts the framework's pipeline. Initially input collections are partitioned and then the preparation step begins, where, based on the work allocation strategy, the inverted index is built, and if necessary the bitmap signatures. Then, two separate threads are launched, (i) a CPU-only thread that executes only CPU-standalone techniques, and (ii) a GPU-enabled thread that may run GPU-standalone or cooperative CPU-GPU techniques. Both these threads run in parallel until the whole process is finished.

For the most efficient filters and techniques per execution, the findings presented in Section 5.2.5 are used. Dataset cardinalities in the order of 10^5 are referred to as small, the ones in the order of 10^6 medium, and those in 10^7 as large. When the average set size is less than 10, it is referred to as small; otherwise, as large. Finally, the number of different set elements is characterized as small up to the order of 10^4, and otherwise, as large. Threshold values of 0.9 are considered as very high, 0.8 as high and $0.5-0.7$ as medium. Threshold values lower than 0.5, where filtering is not effective and only the alternative of brute-forcing the output is the best performing approach, are not considered. As it can be seen in Figure 5.16, the GPU-standalone prefix technique handles better non large datasets on medium and high threshold values. As the dataset cardinality increases, cooperative CPU-GPU techniques tend to perform better on medium and high thresholds. Respectively, the CPU-standalone prefix technique remains competent regardless of the dataset cardinality on very high thresholds. Last, the GPU-standalone bitmap technique is quite efficient on medium thresholds for medium dataset cardinalities combined with low number of different elements and high average set size.

By using the findings presented in Section 5.2.5 as a baseline, the transition to a hybrid solution is more convenient. More specifically, based on the provided overview on each technique's strong and weak points, the best performing ones per scenario can be selected and adapted to the framework's work allocation strategies. For example, it is evident that for large datasets and high to very high thresholds, the GPU-standalone techniques do not perform well. Therefore, in these cases, it is preferred to run concurrently CPU-standalone techniques alongside cooperative CPU-GPU techniques. Further details are provided in Section 5.5.

Algorithm 5.1 Hyset Queue.

Input: R, n, τ_n, Bitmap size bmp
Output: All similar $pairs$ with $sim(r_i, r_j) \geq \tau_n$

1: $blocks \longleftarrow divide(R, n)$
2: $I \longleftarrow \{\}$
3: **for each** block $b_i \in blocks$ **do**
4: $\quad I \longleftarrow I \cup \{index(b_i, \tau_n), offsets(b_i)\}$
5: **end for**
6: $bitmaps \longleftarrow \{\}$
7: **if** $bmp > 0$ **then**
8: $\quad bitmaps \longleftarrow constructBitmaps(R, bmp)$
9: **end if**
10: $queue \longleftarrow populate(blocks)$
11: **while** $queue \neq \emptyset$ **do**
12: $\quad cpuJoin(queue.pop(), I, \tau_n)$ } Runs in parallel
13: $\quad gpuJoin(queue.pop(), I, bitmaps, \tau_n)$ }
14: **end while**

Algorithm 5.2 Hyset Dichotomy.

Input: R, n, τ_n, Bitmap size bmp, Dichotomy point x
Output: All similar $pairs$ with $sim(r_i, r_j) \geq \tau_n$

1: $R_0 \longleftarrow R[0 - x)$
2: $R_1 \longleftarrow R[x - |R|)$
3: $I_0 \longleftarrow \{index(R_0, \tau_n), offsets(R_0)\}$
4: $bitmaps \longleftarrow \{\}$
5: **if** $bmp > 0$ **then**
6: $\quad bitmaps \longleftarrow constructBitmaps(R, bmp)$
7: **end if**
8: $cpuJoin(R_0, I_0, \tau_n)$ } Runs in parallel
9: $gpuJoin(R_1, R, bitmaps, n, \tau_n)$ }

5.4.4 Implementation details

The algorithmic overview of the two work allocation strategies is presented in Algorithms 5.1 and 5.2. In order to incorporate the state-of-the-art techniques in a single framework, there are some implementation peculiarities. The first one concerns how index data might be shared between the CPU and GPU, while the second one concerns the effect of the selected work allocation strategy on the indexing step.

Since prefix filter is supported in every category of techniques, a uniform inverted index structure is designed in order to be used by both the CPU and GPU conveniently. Thus, the inverted

index consists of two linear memory arrays, (i) *index*, and (ii) *offsets*. The former holds inverted lists in a sequence, while the latter is used to delimit each inverted list boundaries.

Indexing is delegated to the GPU since it can be regarded as a composition of two primitives, sort and prefix sum, in which the GPU excels the CPU. However, depending on the selected work allocation strategy, there are different memory requirements. For the concurrent queue, since joins are conducted at a block level, the required memory space for the offsets array is increased by a factor of $\frac{|R|}{n}$. On the other hand, that is not the case for the partition join, since the inverted index is constructed at a partition level and the required memory for the offsets array is increased only by a factor of p.

In addition, for the concurrent queue, the complete inverted index is constructed on the GPU and copied back in main memory where it resides throughout the complete process (Algorithm 5.1, lines 3-4). Whenever the GPU is to conduct a block join, the corresponding block's inverted index is transferred to the GPU memory. In case of dichotomy, the inverted index to be used by the CPU is built beforehand (Algorithm 5.2, line 3) and every inverted index required for a block join is constructed on the GPU (Algorithm 5.2, line 8) on the fly .

For the bitmap filter, since it is only available for the GPU part, all bitmap signatures are generated and stored in the GPU memory (Algorithm 5.1, lines 6-7, Algorithm 5.1, lines 5-6). Thus, for both work allocation strategies, the CPU relies on the inverted index whereas the GPU is index-independent.

5.5 Hyset Evaluation

In this section, the proposed hybrid framework is experimentally evaluated with the state-of-the-art techniques found in literature and an in-depth analysis of the results is conducted. The same experimental environment, as described in Section 5.2, is used. In order to incorporate the CPU-standalone techniques and the cooperative CPU-GPU techniques to the hybrid framework, the original implementations, provided by [62] and described in Section 4.2 respectively, are modified. For the GPU-standalone techniques the implementation of [77] was used for the prefix filter and the modified version, as described in Section 5.1, for the bitmap filter.

Experiments are conducted on three types of datasets: (i) original, the eight real-world datasets employed in Section 5.2.2, (ii) increased, the inflated original datasets, same to those employed in Section 5.2.4, and (iii) synthetic, the artificially created datasets, same to those employed in Section 5.2.4.

Next, the main experiments are presented. After, the work allocation strategies are evaluated. Finally, end-to-end runtimes are reported and overall observations are discussed.

5.5.1 Main Experiments

The two hybrid work allocation strategies, denoted as *queue* and *dichotomy*, are compared against the state-of-the-art single threaded CPU-standalone and GPU-enabled techniques for the set similarity join. The best join times measured for all the techniques for the original datasets are presented in Table 5.9 and the speedups in Table 5.10. Respectively, for the increased datasets, the join times are presented in Table 5.11 and the speedups in Table 5.12. Last, for the synthetic datasets, the join times are presented in Table 5.13 and the speedups in Table 5.14. Every unsuccessful test,

		Threshold τ_n				
		0.5	0.6	0.7	0.8	0.9
AOL	CPU	276.16	69.32	10.41	3.23	1.21
	GPU	693.21	466.36	310.14	239.87	202.56
	CPU-GPU	207.12	50.41	7.31	2.52	1.01
	SKJ	**126.74**	**29.71**	7.22	3.67	2.16
	Queue	142.63	65.42	29.13	20.78	16.92
	Dichotomy	163.12	49.31	**6.00**	**2.47**	**0.74**
DBLP-200K	CPU	442.30	228.59	104.11	23.13	2.33
	GPU	8.21	6.73	4.39	**1.47**	**0.41**
	CPU-GPU	201.60	128.08	57.83	19.14	2.81
	SKJ	615.02	212.01	52.11	8.14	0.86
	Queue	4.64	4.16	2.81	1.68	0.54
	Dichotomy	**3.81**	**3.51**	**2.80**	2.09	1.77
DBLP-1M	CPU	N/A	N/A	4560.34	1157.92	96.76
	GPU	72.97	**40.09**	**29.11**	**16.95**	**6.76**
	CPU-GPU	N/A	N/A	2023.29	463.16	41.17
	SKJ	N/A	N/A	N/A	234.06	16.60
	Queue	**63.60**	44.61	35.53	19.5	6.91
	Dichotomy	102.86	41.63	33.44	17.13	6.93
KOSARAK	CPU	113.57	30.66	10.22	7.33	7.05
	GPU	12.2	6.23	3.35	2.63	2.24
	CPU-GPU	100.5	26.69	7.66	5.01	4.63
	SKJ	52.10	13.40	5.27	4.38	4.13
	Queue	**9.37**	**3.74**	**1.74**	**1.16**	**0.95**
	Dichotomy	18.99	6.06	3.22	2.41	2.01
ORKUT	CPU	161.40	59.51	24.13	9.72	3.23
	GPU	78.08	38.73	19.60	10.49	5.32
	CPU-GPU	128.58	57.51	23.73	10.32	3.36
	SKJ	397.64	128.28	46.14	19.58	9.46
	Queue	97.28	42.99	20.90	12.05	8.26
	Dichotomy	**27.41**	**13.87**	**6.87**	**3.18**	**1.61**

		Threshold τ_n				
		0.5	0.6	0.7	0.8	0.9
BMS	CPU	41.12	13.67	4.20	1.06	0.50
	GPU	3.51	2.07	1.09	0.83	0.60
	CPU-GPU	27.36	9.51	3.19	1.14	**0.31**
	SKJ	19.99	6.38	2.22	0.88	0.34
	Queue	**3.08**	**1.62**	**0.89**	**0.59**	0.39
	Dichotomy	4.39	2.49	1.37	1.03	0.79
DBLP-300K	CPU	1151.44	569.53	215.72	52.67	5.58
	GPU	17.20	14.18	9.86	3.21	**0.79**
	CPU-GPU	473.15	278.32	135.24	39.87	6.89
	SKJ	1498.01	519.65	122.58	18.55	1.72
	Queue	**10.11**	6.94	4.85	3.23	0.84
	Dichotomy	12.70	**5.35**	**4.38**	**3.04**	2.07
ENRON	CPU	38.12	11.47	3.64	1.06	0.27
	GPU	4.37	1.98	0.99	0.65	0.27
	CPU-GPU	33.03	10.21	3.47	1.09	0.29
	SKJ	36.57	10.02	3.15	1.41	0.67
	Queue	7.69	3.15	1.34	0.70	0.46
	Dichotomy	**3.14**	**1.76**	**0.92**	**0.47**	**0.25**
LVJ	CPU	277.03	70.34	17.21	4.72	**1.35**
	GPU	64.7	36.73	21.98	13.47	6.82
	CPU-GPU	187.73	49.56	13.82	4.46	1.36
	SKJ	235.23	145.51	97.03	50.25	25.67
	Queue	71.96	33.33	16.41	9.48	6.22
	Dichotomy	**54.49**	**26.94**	**9.12**	**3.51**	1.82
TWITTER	CPU	N/A	N/A	4697.91	897.66	63.39
	GPU	**114.01**	**83.47**	**47.7**	**29.86**	9.56
	CPU-GPU	N/A	N/A	3001.98	557.41	46.19
	SKJ	N/A	N/A	1385.29	189.09	16.17
	Queue	143.36	107.95	64.04	37.96	**8.55**
	Dichotomy	136.75	110.51	64.54	42.09	9.62

Table 5.9: Join Times for the original datasets (in seconds).

either due to memory constraints or very long run time, is noted as not available (N/A). Each time reported for the *CPU* is the overall best among the three best performing algorithms (i.e. Allpairs, PPJoin, GroupJoin) as stated in Sections 4.4 and 5.2.2. To account for advances after the publication of [62], times for the skipping algorithm presented in [100], denoted as *SKJ*, which has a single threaded CPU implementation, are also reported. The source code of SKJ was provided by the authors of [100]. Respectively, for the *CPU-GPU* each time reported is the overall best among the three CPU algorithms and the best GPU verification techniques described in Sections 4.4 and 5.2.2. For the *GPU*, each time reported is the best between the total times for all the GPU operations required to perform the set similarity join, using either the prefix or bitmap filter in line with the discussion in Section 5.4.3.

Similarly to the evaluation presented in Section 5.2.4, for all the original datasets the experiments were conducted in the threshold range $\tau_n \in [0.5 - 0.9]$. Respectively, for all the increased and synthetic datasets, the threshold range is narrowed to $\tau_n \in [0.7 - 0.9]$ due to large overall join times. In addition, for the partitioning scheme, a global block size $n = 10000$ is used. In general, this block size is smaller than the available GPU global memory available (after storing the data and indices), but in the experimental evaluation led to lower amortized overhead for memory cleaning and more efficient length filtering at the block level. The experimental results for each dataset category are analyzed in turn.

	Threshold τ_n					Overall
	0.5	**0.6**	**0.7**	**0.8**	**0.9**	
AOL	-	-	1.2	1.02	1.36	1.19
BMS	1.13	1.27	1.22	1.4	-	1.26
DBLP-200K	2.15	1.91	1.56	-	-	1.88
DBLP-300K	1.7	2.65	2.25	1.05	-	1.91
DBLP-1M	1.14	-	-	-	-	1.14
ENRON	1.39	1.12	1.07	1.38	1.08	1.21
KOSARAK	1.3	1.66	1.92	2.26	2.35	1.9
LVJ	1.18	1.36	1.51	1.27	-	1.33
ORKUT	2.84	2.79	2.85	3.05	2	2.71
TWITTER	-	-	-	-	1.11	1.11
Overall	1.6	1.82	1.77	1.74	1.81	

Table 5.10: Speedups for the original datasets.

Original datasets

The main results are presented in Table 5.9, where, the *dichotomy* technique is restricted so that each type of device is allocated at least 20% of the workload. As can be seen from the table, in 70% of the cases, the hybrid techniques achieve speedup over the state-of-the-art techniques. On average 1.68X speedup is observed, with the largest one achieved at 3.05X for the ORKUT dataset with $\tau_n = 0.9$. Table 5.10 provides an overview of the speedups per dataset and per threshold for the original datasets.

On the other hand, for the rest 30% of the combinations of datasets and thresholds, none of the hybrid techniques yield any performance speedup, when the dichotomy technique is forced to allocate at least 20% to a second type of device. This happens for three reasons. First, in certain datasets such as BMS, DBLP with its variations and LVJ, for very high thresholds, i.e. $\tau_n = 0.9$, standalone prefix filtering is quite effective and the overall join times are in general very low; in these cases the queue-based hybrid technique is slightly worse than the optimal solution. Second, for datasets such as DBLP-1M and TWITTER with threshold $\tau_n \in [0.5-0.8]$, GPU-standalone bitmap filter is the fastest solution due to its high filtering ratio (effectiveness) and efficiency. In contrast, incorporating the CPU by using prefix filter in such cases only adds overhead. Third, there is the case of AOL dataset, where for threshold $\tau_n \in [0.5 - 0.6]$, the skipping algorithm SKJ [100] is the fastest solution.

The discussion about the cases where SKJ dominates is deferred when the synthetic datasets are evaluated. Regarding the first and second reasons above, it must be noted that they are direct consequence of splitting the workload in the dichotomy technique so that a processor takes at least 20% of the workload. If either CPU or GPU was allowed to process up to 100% of the workload and the proposed hybrid framework encapsulated a mechanism to derive the optimal splitting point, then the dichotomy would be the best performing solution in all cases, apart from those where SKJ dominates (96% of the cases).

		Threshold τ_n		
		0.7	0.8	0.9
BMS-25	CPU	7387.55	2875.11	1247.35
	GPU	705.65	420.69	255.16
	CPU-GPU	4752.49	1686.74	725.56
	SKJ	1825.30	561.79	146.16
	Queue	**416.25**	**201.20**	**91.10**
	Dichotomy	832.26	514.00	293.94
ENRON-25	CPU	2176.58	265.70	16.63
	GPU	1115.32	471.59	37.74
	CPU-GPU	1688.64	224.80	20.30
	SKJ	955.03	138.91	31.32
	Queue	405.58	94.87	21.20
	Dichotomy	**293.28**	**91.11**	**11.73**
LVJ-5	CPU	328.44	55.25	9.12
	GPU	527.75	307.2	133.93
	CPU-GPU	252.86	45.69	8.85
	SKJ	**140.11**	39.63	15.62
	Queue	N/A	N/A	N/A
	Dichotomy	152.71	**24.68**	**3.81**

Table 5.11: Join times for the increased datasets (in seconds).

	Threshold τ_n			**Overall**
	0.7	**0.8**	**0.9**	
BMS-25	1.69	2.09	1.6	1.79
ENRON-25	3.25	1.52	1.41	2.06
LVJ-5	-	1.6	2.32	1.94
Overall	1.77	1.74	1.81	

Table 5.12: Speedups for the increased datasets.

The bottom line of the observations is that the hybrid framework, even without fine-tuning is capable of yielding tangible benefits. If fine-tuning is performed and allowed dichotomy to split the workload arbitrarily, then the dichotomy would perform better than what reported in Table 5.9, while queue-based hybrid would become obsolete.

Increased datasets

When increasing the datasets, the benefits of the hybrid solutions become even more significant. As reported in Table 5.11, for the 89% of the total experiments, on average 1.94X speedup is measured, with the largest speedup observed being 3.25X for the ENRON-25 dataset with $\tau_n = 0.7$. Please note the threshold in these experiments is 0.7 to 0.9; for lower thresholds, the speedups increase on average. As an exception, for the LVJ-5 dataset with $\tau_n = 0.7$ the SKJ algorithm is faster.

The respective speedups for the original datasets is 1.26X, i.e., the hybrid techniques achieve further improvement by 70%. The exact speedups are presented in Table 5.12. When the prefix

filter is efficient, both the CPU and GPU can contribute greatly to the set similarity join. Hence, when the dataset cardinality increases, the hybrid techniques can utilize both edges quite effectively and as a result, speedups become more evident. However, the inability to run the queue hybrid flavor due to memory constraints in some cases must be noted. The queue's memory cost is further analyzed in Section 5.5.2.

Synthetic datasets

The aggregate results for the synthetic datasets are presented in Table 5.13. For the 44% of the total experiments, the hybrid techniques achieve 1.48X speedup on average. The largest speedup measured is 2.22x for the 10M-500K-5 with $\tau_n = 0.9$. For the rest 56%, SKJ achieves 2.08X speedup on average over the state-of-the-art and 1.49X over the hybrid techniques. Table 5.14 shows the speedups of the hybrid techniques for the synthetic datasets. By comparing the hybrid solutions against SKJ, a pattern on the runtimes, which is independent of the dataset cardinality, can be observed and, instead, is mainly based on the number of different elements and average set size or a combination of both. When the average set size is small (5), hybrid techniques perform well since prefix filtering on both the CPU and GPU remain competent, especially for threshold $\tau_n \geq 0.8$. However, as the average set size increases, SKJ performs better because of the computational cost sharing it encapsulates. In addition, with the combination of a low number of different elements (50K), sets have higher probability of sharing common elements, which favors SKJ.

However, when SKJ behaves better, it must also be highlighted that SKJ is constructed in a different manner, where indexing is performed out of the filtering phase. Thus, the join times reported thus far, in all techniques apart from SKJ, include indexing. On the other hand, SKJ's indexing can be deemed as a cost that can be easily amortized when multiple similarity queries are submitted on the same dataset.

5.5.2 Performance analysis

In this section, the performance analysis and comparison between the hybrid solutions are presented along with discussion on how each hybrid solution can be used. Furthermore, the memory cost of the queue technique is highlighted, especially for datasets with large number of different elements. Finally, the necessity of choosing a good splitting point is discussed and its impact on the overall performance for the dichotomy technique is emphasized.

Queue vs Dichotomy

In order to compare the hybrid techniques, the *gap factor*, i.e. the gap between the fastest and the slowest technique per dataset and threshold between the two hybrid proposals, is measured. As shown in Table 5.15, there is a large variation in the gap factor, especially in very large thresholds, where a gap factor of up to 22.86 for the AOL dataset at $\tau_n = 0.9$, with dichotomy being faster than queue, is measured. As the threshold value τ_n decreases, the variation in the gap factor decreases as well.

In general, queue is inferior to dichotomy. However, there are certain cases, such as BMS, KOSARAK and BMS-25 in which queue is faster. For these datasets, the respective indices can be characterized as lightweight both (i) vertically, i.e. the number of inverted lists is relatively small

		Threshold τ_n		
		0.7	0.8	0.9
5M-50K-5	CPU	25.77	20.19	1.10
	GPU	110.39	77.38	33.08
	CPU-GPU	20.17	4.25	0.97
	SKJ	**11.64**	4.07	1.20
	Queue	19.47	7.47	2.76
	Dichotomy	14.86	**3.32**	**0.7**
5M-50K-25	CPU	517.40	123.19	15.37
	GPU	1115.32	471.59	37.74
	CPU-GPU	399.64	93.27	12.40
	SKJ	**133.78**	**31.87**	**5.65**
	Queue	164.93	77.29	22.81
	Dichotomy	192.77	75.10	11.52
5M-500K-5	CPU	8.04	2.30	0.58
	GPU	112.53	79.19	34.31
	CPU-GPU	6.46	2.00	0.63
	SKJ	5.16	2.00	1.09
	Queue	15.87	7.72	2.87
	Dichotomy	**4.96**	**1.40**	**0.25**
5M-500K-25	CPU	71.54	20.13	4.87
	GPU	255.58	172.18	83.06
	CPU-GPU	57.97	16.29	4.30
	SKJ	**31.40**	**10.78**	4.50
	Queue	90.30	46.70	15.16
	Dichotomy	40.48	11.07	**2.76**

		Threshold τ_n		
		0.7	0.8	0.9
10M-50K-5	CPU	104.98	18.47	3.37
	GPU	446.68	308.60	128.90
	CPU-GPU	82.28	15.00	3.18
	SKJ	**46.21**	14.89	3.11
	Queue	74.19	26.28	8.92
	Dichotomy	60.55	**10.46**	**1.92**
10M-50K-25	CPU	2137.76	511.87	55.63
	GPU	1163.30	744.10	338.31
	CPU-GPU	1620.20	370.30	43.67
	SKJ	**574.88**	**132.23**	**16.88**
	Queue	659.10	300.57	83.84
	Dichotomy	820.61	335.10	44.17
10M-500K-5	CPU	30.26	6.95	1.40
	GPU	444.42	321.22	133.07
	CPU-GPU	24.29	5.70	1.54
	SKJ	**16.61**	5.55	2.20
	Queue	56.68	26.64	8.75
	Dichotomy	18.22	**4.21**	**0.69**
10M-500K-25	CPU	285.25	71.06	13.40
	GPU	1019.61	685.91	328.83
	CPU-GPU	226.80	56.11	11.31
	SKJ	**107.05**	**33.25**	9.96
	Queue	352.05	177.90	53.98
	Dichotomy	174.68	42.72	**8.54**

		Threshold τ_n		
		0.7	0.8	0.9
20M-50K-5	CPU	422.20	66.17	11.83
	GPU	1793.57	1238.00	512.68
	CPU-GPU	346.82	54.31	10.56
	SKJ	**199.36**	60.15	9.08
	Queue	284.10	97.01	31.23
	Dichotomy	254.44	**36.26**	**6.30**
20M-50K-25	CPU	9145.80	2168.53	206.27
	GPU	4653.30	2976.64	1348.73
	CPU-GPU	6577.16	1563.72	161.67
	SKJ	**2417.44**	**585.28**	**64.83**
	Queue	2617.97	1185.89	318.88
	Dichotomy	3508.65	1459.78	99.45
20M-500K-5	CPU	114.66	22.11	3.87
	GPU	1783.23	1257.41	528.55
	CPU-GPU	90.35	17.73	3.72
	SKJ	**60.07**	18.48	4.89
	Queue	213.35	95.68	29.29
	Dichotomy	61.82	**10.65**	**1.67**
20M-500K-25	CPU	1226.02	266.27	39.43
	GPU	4087.97	2746.53	1312.94
	CPU-GPU	878.88	202.19	32.50
	SKJ	**390.52**	**112.68**	23.83
	Queue	1393.08	692.26	200.71
	Dichotomy	715.30	132.66	**19.44**

Table 5.13: Join time for the synthetic datasets (in seconds).

due to the small number of different elements, and (ii) horizontally, i.e. the average length of an inverted list is small because of the small average set size. When this is the case, the hybrid queue tech-

	Threshold τ_n			Overall
	0.7	**0.8**	**0.9**	
5M-50K-5	-	1.22	1.38	1.3
5M-50K-25	-	-	-	-
5M-500K-5	1.04	1.42	2.32	1.59
5M-500K-25	-	-	1.55	1.55
10M-50K-5	-	1.42	1.61	1.52
10M-50K-25	-	-	-	-
10M-500K-5	-	1.31	2.02	1.67
10M-500K-25	-	-	1.16	1.16
20M-50K-5	-	1.49	1.44	1.46
20M-50K-25	-	-	-	-
20M-500K-5	-	1.43	2.32	1.87
20M-500K-25	-	-	1.22	1.22
Overall	1.04	1.42	1.66	

Table 5.14: Speedups for the synthetic datasets.

nique performs better than the dichotomy technique. On the contrary, the dichotomy technique performs better in the AOL, ENRON, LVJ and ORKUT for the original datasets, in ENRON-25 for the increased datasets and in the majority of the synthetic datasets compared to the hybrid queue technique. All of these datasets, share the common feature of having larger indices.

Further, in most cases, having larger indices favors the CPU prefix filtering, especially for high and very high thresholds. For such thresholds, it is beneficial to delegate most of the workload to the CPU and leave a smaller portion for the GPU; this can be enforced in dichotomy more efficiently. The dichotomy workload split is further analyzed in Section 5.5.2.

However, the correct interpretation of the gap factor needs to take into account the absolute runtimes as presented in Tables 5.9, 5.11, 5.13 and Figures 5.22, 5.23, 5.24. More specifically, lower gap factors in lower thresholds may correspond to more significant runtime difference. For example, on the right plot in Figure 5.23, the runtime difference for threshold 0.7 is apparent while the gap factor is 1.38; by contrast, for threshold 0.9, the runtime difference is negligible although the gap factor is 1.8.

Additionally, for completeness, it is noted that both hybrid techniques perform better whenever the prefix filter is effective. When the bitmap filter is the most efficient and the dataset cardinality is relatively small, such as in DBLP-200K and DBLP-300K, employing either dichotomy or queue, using the bitmap filter for the GPU and the prefix for the CPU, seems superior. However, for larger datasets, such as DBLP-1M and TWITTER, standalone GPU bitmap performs better than any hybrid solution for not high thresholds, e.g., equal to or lower than 0.8. This also relates to the discussion on the dichotomy splitting point further below.

Finally, depending on the dataset and query characteristics, the impact of occupancy and global memory access efficiency may largely differ among winning cases[5].

[5] Details are provided along with the source code.

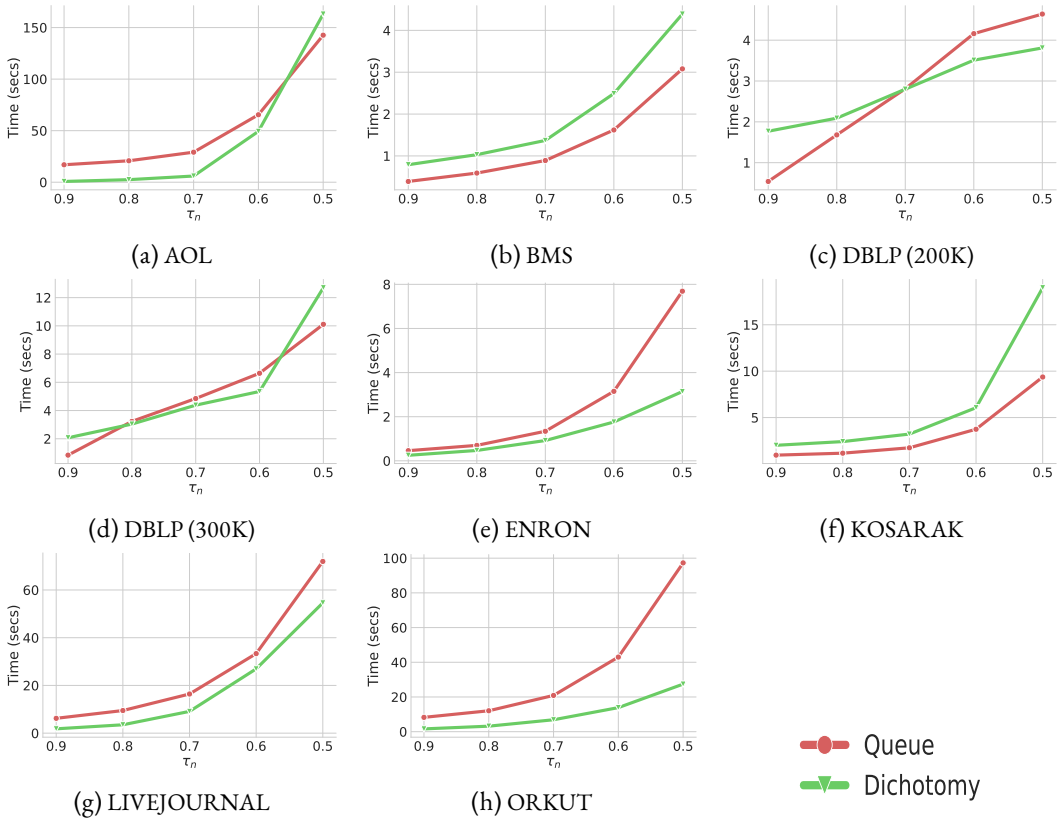
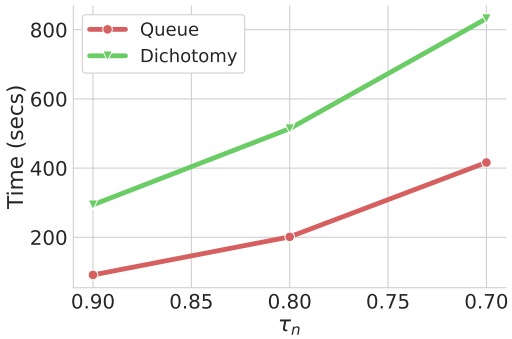


Figure 5.22: Comparison between the two work allocation strategies for the original datasets.

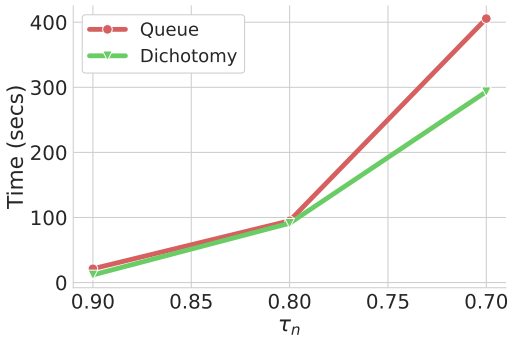
Queue memory cost

For the queue technique to work properly, the complete index must be constructed in advance at block level. This approach leads to many small indices; in particular, as many as the number of blocks, so that both the CPU and GPU can run the join operation seamlessly without any end waiting for the other. However, this also implies an increase in the memory space required to store these indices, as for each index, the boundaries of the corresponding inverted lists must be specified. Otherwise, in a single standalone index scenario, for each index probe, several binary searches would be needed in order to determine the start and end of an inverted list. This would result in an extra overhead cost, especially in the parallel environment of the GPU.

In Figure 5.25, the total memory required to store the complete standalone index is compared with the corresponding one required to store all the small indices for the queue technique. Although for all three datasets the number of blocks is in the order of thousands, specifically for the BMS-25 dataset, there is a small difference in the required memory for both scenarios. This is due to the small number of different elements (1657). However, for ENRON-25 and LVJ-5, the respective numbers of different elements are in the order of millions which leads to a sharp increase in the required memory space for the queue technique. Furthermore, the required memory space

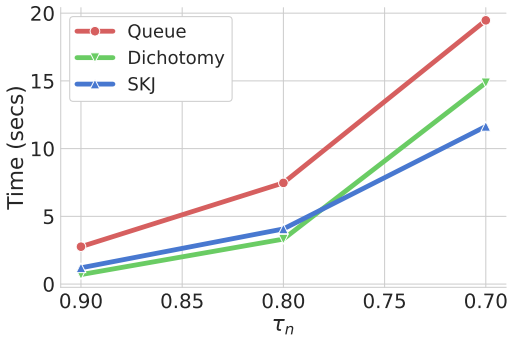


(a) BMS-25

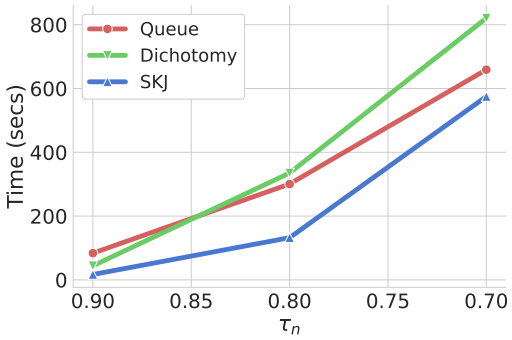


(b) ENRON-25

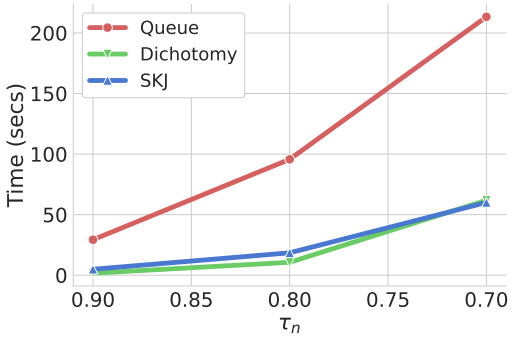
Figure 5.23: Comparison between the two work allocation strategies for the increased datasets.



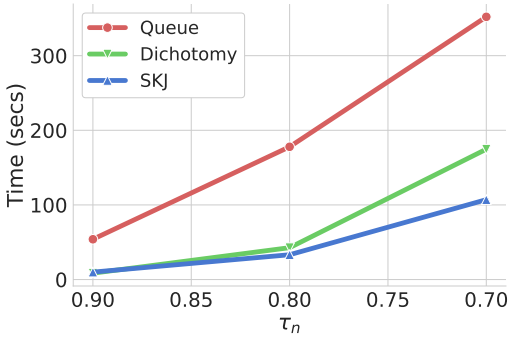
(a) 5M-50K-5



(b) 10M-50K-25



(c) 20M-500K-5



(d) 10M-500K-25

Figure 5.24: Comparison between the two work allocation strategies for the synthetic datasets.

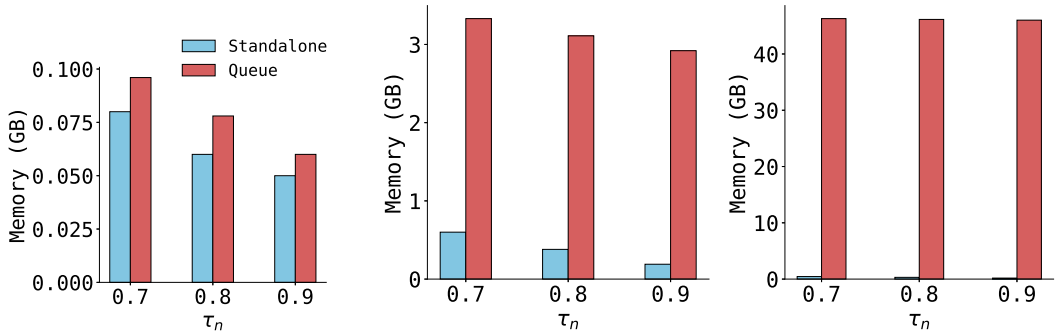
Dataset	Threshold τ_n				
	0.5	**0.6**	**0.7**	**0.8**	**0.9**
AOL	1.14	1.32	4.85	8.41	22.86
BMS	1.42	1.53	1.53	1.74	2.02
DBLP-200K	1.21	1.18	1.00	1.24	3.27
DBLP-300K	1.25	1.29	1.10	1.06	2.46
DBLP-1M	1.61	1.07	1.06	1.13	1.00
ENRON	2.44	1.78	1.45	1.48	1.84
KOSARAK	2.02	1.62	1.85	2.07	2.11
LVJ	1.32	1.23	1.79	2.70	3.41
ORKUT	3.54	3.09	3.04	3.78	5.13
TWITTER	1.04	1.02	1.00	1.10	1.12
BMS-25	-	-	1.99	2.55	3.22
ENRON-25	-	-	1.38	1.04	1.80
LVJ-5	-	-	-	-	-
5M-50K-5	-	-	1.31	2.25	3.94
5M-50K-25	-	-	1.16	1.02	1.98
5M-500K-5	-	-	3.19	5.51	11.48
5M-500K-25	-	-	2.23	4.21	5.49
10M-50K-5	-	-	1.22	2.51	4.64
10M-50K-25	-	-	1.24	1.11	1.98
10M-500K-5	-	-	3.11	6.32	12.68
10M-500K-25	-	-	2.02	4.16	6.32
20M-50K-5	-	-	1.11	2.67	4.95
20M-50K-25	-	-	1.34	1.23	3.20
20M-500K-5	-	-	3.45	8.98	17.53
20M-500K-25	-	-	1.94	5.21	10.32

Table 5.15: Gap factor per dataset and threshold.

for the LVJ-5 dataset exceeds the available memory of the experimental setup and as a result the queue technique cannot be run for this particular dataset.

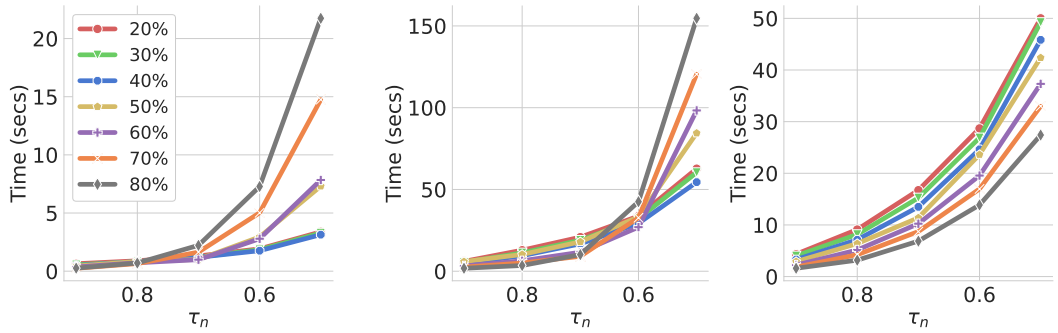
Dichotomy splitting point

The main goal of the hybrid dichotomy technique is the proper workload split among the CPU and GPU in order to achieve the best possible execution overlap. However, the selection of a good splitting point is not straightforward and, in the proposed hybrid framework, its automated decision mechanism is left as future work. For each of the experiments, the dichotomy technique ran with varying the splitting point within the range [0.2,0.8] and the lowest timings are reported in Tables 5.9,5.11 and 5.13. As already mentioned, if a mechanism to judiciously select the optimal splitting point in the range of [0,1] existed, then the queue technique would always run



(a) BMS-25 (b) ENRON-25 (c) LVJ-5

Figure 5.25: Index memory cost for the increased datasets with $n = 10000$.



(a) BMS (b) LIVEJOURNAL (c) ORKUT

Figure 5.26: Comparison of different dichotomy splitting points.

slower than dichotomy; moreover, dichotomy would also always dominate the other standalone techniques.

In Figure 5.26, the runtime difference over varying dichotomy splitting points for the BMS, LVJ and ORKUT datasets is presented. The splitting point percentage denotes the amount of workload assigned to the CPU. For the BMS and LVJ datasets, the runtime difference becomes more evident for $\tau_n < 0.7$ where filtering gradually becomes ineffective. As a result, the best performing splitting point for both datasets in lower thresholds is at 40% for the CPU and 60% for the GPU. This behavior is aligned with the findings of Section 5.2.5, summarized in Figure 5.16. However, this is not the case for the ORKUT dataset where, even for higher threshold values, there is more significant relative difference between the various splitting points. Nevertheless, for this dataset, the absolute runtime differences are less significant. Also, for this dataset, it is more beneficial to allocate more work to the CPU due to the effectiveness of the prefix filter, something already discussed above.

Dataset	τ_n	Multi-CPU	Fier et al. [29]	Hyset	Speedup
DBLP-1M	0.7	1623.39	5285.66	33.44	48.54
	0.8	426.38	1308.79	17.13	24.89
	0.9	54.98	98.88	6.91	7.95
KOSARAK	0.7	15.65	24.81	1.74	8.99
	0.8	15.08	19.82	1.16	13.00
	0.9	14.75	19.36	0.95	15.52
ORKUT	0.7	31.72	33.71	6.87	4.61
	0.8	14.9	17.68	3.18	4.68
	0.9	8.28	7.86	1.61	4.88
TWITTER	0.7	2208.75	6103.53	64.04	34.49
	0.8	111.298	1248.98	37.96	2.93
	0.9	74.03	89.18	8.55	8.65
BMS-25	0.7	-	7859.95	416.25	18.88
	0.8	-	2234.39	201.2	11.10
	0.9	-	722.33	91.1	7.92
ENRON-25	0.7	-	3172.74	293.28	10.81
	0.8	-	269.47	91.11	2.95
	0.9	51.24	21.08	11.73	1.79
LVJ-5	0.7	-	257.40	152.71	1.68
	0.8	-	50.98	24.68	2.06
	0.9	-	12.42	3.81	3.26

Table 5.16: Hyset comparison against multi-threaded CPU alternatives. Runtimes are in seconds.

Dataset	MapReduce [28]	Queue	Dichotomy	Speedup
AOL	68	16.62	0.74	91.89
ENRON	130	0.46	0.25	520
KOSARAK	113	0.95	2.01	118.94
LVJ	254	6.22	1.82	139.56

Table 5.17: Hyset comparison against MapReduce (as reported in [28]) for $\tau_n = 0.9$. Runtimes are in seconds.

5.5.3 Comparison against parallel CPU-based approaches

Over the past years, there have been attempts to employ parallelism for the set similarity join problem via the MapReduce paradigm. In [28], Fier et al. evaluate the state-of-the-art MapReduce techniques for set similarity. As their main result, they highlight the poor performance of MapReduce over a single node solution and also point out its incapability to scale for larger data volumes. More recently, in [29], Fier at al. introduced a multi-threaded CPU framework for the same problem. It must be noted that none of these works are GPU-aware. For completeness, a

simple multi-threaded CPU version is also implemented and directly evaluated with the state-of-the-art proposal found in [29].

The implemented multi-threaded solution is denoted as *Multi-CPU*. Each time reported for the *Multi-CPU* is the best among multiple configurations of the two CPU-based algorithms All-Pairs and PPJoin. The input dataset is split into p partitions equal to the number of threads launched and assign an equal number of joins to each thread along with the corresponding portion of the inverted index. This results in higher memory requirements since each thread has its own index. Also, in [29], Fier et al. introduce a more optimized multi-threaded CPU implementation for set similarity join and they claim that they achieve a $2-10$X speedup over single threaded CPU solutions. The proposed hybrid framework is compared with their solution for four original datasets and all the increased datasets for threshold values $\tau_n \geq 0.7$.

The runtimes are reported in Table 5.16. As it can be seen, there is not a single case where either the simple *Multi-CPU* or the work of Fier et al. [29] is superior to the hybrid framework. In contrast, it is observed that the hybrid framework achieves a 10.75X speedup on average against multi-threaded CPU alternatives. The highest speedups reported for DBLP-1M for $\tau_n \in [0.7 - 0.8]$ and TWITTER for $\tau_n = 0.7$ are due to the bitmap filter which is more suitable for the GPU. Thus, employing the GPU via the proposed hybrid techniques seems more beneficial in every case. The inability to launch the simple multi-threaded CPU implementation for the majority of increased datasets due to memory constraints is also noted. Finally, in [28], Fier et al. evaluate the state-of-the-art MapReduce techniques for the set similarity join problem using a 12-node cluster. As their main result, they highlight the poor performance of MapReduce over a single node solution and also point out its incapability to scale for larger data volumes. In order to emphasize the inefficiency of MapReduce, in Table 5.17 the reported runtimes of [28] are compared selectively with the framework's runtimes for four original datasets with $\tau_n = 0.9$. As it can been seen, the hybrid framework achieves a 217.6X speedup on average; the speedup range is 91.89X to 520X.

The main conclusion of these experiments is that the hybrid solution cannot be outperformed by multi-threaded CPU implementations; however, as discussed next, devising efficient solutions within the hybrid context that are capable of benefiting from the multi-threaded CPU techniques is one of the identified open issues.

5.6 Remarks

Exact set similarity join is a notoriously expensive operation, for which several techniques and algorithms have been proposed. In a single machine setup, the most prominent solutions incorporate the use of a GPU as demonstrated in the experimental evaluation in Section 5.2. To this end, the findings summarized in Section 5.2.5, are used as baseline for the development of a hybrid framework that utilizes both the CPU and GPU to tackle the problem.

The proposed hybrid framework, which encapsulates state-of-the-art CPU and GPU-enabled solutions for the exact set similarity join problem with a view to deriving a higher-level technique, manages to execute fast regardless of changes in the dataset and query characteristics. Through extensive evaluation and performance analysis, speedups of up to 3.25X over standalone solutions are shown, along with the overcoming of the main problem of the GPU-enabled set similarity joins

thus far, namely that different techniques are dominant under different conditions. In addition, the proposed hybrid framework outperforms multi-threaded CPU solutions.

Although the proposed hybrid framework achieves speedup in the majority of cases, the heterogeneity in existing filters hinders the possibility for even larger speedups. Moreover, the prefix filtering dependence on an index structure makes it more CPU-oriented. There are certain cases, especially for lightweight indices as shown in the queue technique, in which the GPU can contribute significantly to prefix filtering. On the contrary, bitmap filtering favors the parallel environment of the GPU exclusively, since it allows a more uniform memory access and compute utilization. As a result, when bitmap filtering is the most effective, involving the CPU does not seem beneficial in most cases. Following up, the full space of significant parameters has not been explored yet. The average set size and the number of distinct set elements play a significant role, since they are directly related to the amount of work. However, a wider range of combinations of dataset size, set element frequency distributions, number of different set elements and average set sizes needs to be explored.

Nevertheless, the proposed hybrid framework can be regarded as a step towards a globally dominant solution. In particular, based on the performance analysis presented in Section 5.5.2, it is shown that the dichotomy technique lays the foundation for a good workload allocation between the CPU and GPU. Moreover, the development of the dichotomy technique has also given rise to a new important research issue, namely the development of an automated way to select a good splitting point, alongside a cost model to select the appropriate CPU and GPU technique per scenario. This requires both the development of efficient cost models and cost-based splitting techniques.

6 Set Intersection

Set intersection is an essential operation in numerous application domains, such as information retrieval for text search engines using an inverted index [6, 21], graph analytics for triangle counting and community detection [41, 86, 98], and database systems for merging RID-lists [78] and performing fast (potentially bitwise) operations on data in columnar format [34]. Over the past years, there has been a considerable research work on improving graph analytics on a GPU, mostly in the context of graph triangle counting, where set intersection dominates the running time [9, 30, 33, 40, 73, 97]. The majority of these studies focus on improving the level of parallelism by reducing redundant comparisons and distributing the workload evenly among GPU threads. Set intersection on GPUs has also been examined in the context of set similarity joins as previously shown in Chapters 4 and 5.

In this chapter, the state-of-the-art GPU techniques for set intersection, when the sets are large, i.e., they may contain up to millions of elements, are compared and evaluated, while a novel technique is introduced. The main rationale behind the high-level approach is to capitalize on existing techniques to the largest possible extent. Therefore, the goal is to transfer existing knowledge and this exploitation of existing solutions takes place at two distinct and complementary levels: (i) GPU-oriented solutions for small set intersection are transferred to a setting that sets are large; the reduction of the set intersection problem to the more generic problem of matrix multiplication is also leveraged. (ii) Set intersection solutions are transferred to solve the set containment problem in both its simpler binary form and in a more generic form, where the degree of containment is reported for every pair of sets checked.

More specifically, techniques that belong to six different rationales, namely intersect path, optimized binary search, hash, bitmap and set similarity join-based solutions, and matrix multiplication are gathered, while novel promising hybrid solutions that combine existing techniques to better fit into the setting are also included. Moreover, it is shown that the proposed techniques are not only sufficient to address the set containment problem but they are also superior to the current state-of-the-art as reported in the literature. However, a key observation is that there is no dominant solution and a main contribution of the present work is to define under which conditions a specific technique should be employed. The results complement earlier results, e.g., in [39], which show that a specific technique, such as binary search-based one is a dominant solution in a specific case; in the present work, evidence as to which technique to choose in different scenarios after examining a broader set of alternatives is provided, and the suggestions are not always aligned with the ones in [39].

Experiments are performed both when the intersection of one pair of (very large) sets (single-instance problem) and the intersections among all set pairs in a database are computed (multi-instance problem). Useful insights are derived, and more specifically, experimental evidence is provided on the superiority of (i) two intersection path flavors for the single-instance problem and

(ii) bitmap-based or similarity join-based or matrix multiplication-based solutions for the multi-instance problem depending on the size of the sets and the size of the element universe. The cases where CPU-GPU co-processing is beneficial, are also discussed. Last, by comparing GPU-based techniques against parallel CPU counterparts, the benefits of using a graphics card are shown in a clearer and fair manner. The benefits of using GPU variants are at the order of magnitude.

The rest of the chapter is organized as follows. Section 6.1 gives a background on the problem and an overview of the related work. Section 6.2 describes and evaluates the GPU techniques for set intersection. In Section 6.3, the technical details and the experiments for set containment are presented. Finally, Section 6.4 discusses the remarks of the present work.

6.1 Preliminaries

In this section, a formal notation for the set intersection and the set containment problems is given. Next, the exact scope of the present work is explained, along with an overview of the related work about set intersection and set containment.

6.1.1 Set Intersection Problem Description

Single-instance set intersection (SISI) problem: given i) a finite universe of elements E, and ii) a set $A = \{e_1^A, \ldots, e_n^A\}$ of size n and a set $B = \{e_1^B, \ldots, e_m^B\}$ of size m, where $e_i^{\{A,B\}} \in E$, set intersection $A \cap B$ produces a new set S containing all the common elements among A and B.

Multi-instance set intersection (MISI) problem: in the multi-instance set intersection problem, a collection of k sets $C = \{S_1, \ldots, S_k\}$ are given, and the aim is to find each individual set intersection among all $\binom{k}{2}$ of pairs (S_i,S_j), where $0 < i < j \leq k$.

Given the definitions above, the solutions of SISI are of $\Omega(n + m)$ time complexity, whereas MISI solutions are of quadratic complexity in k without the problem definition leaving any space for pair pruning, as, for example, in problems such as set similarity joins. Therefore, the focus is not on evaluating the behavior of algorithms differing in asymptotic complexity, since all techniques are of similar complexity; rather, the goal is to assess the impact of different potentially low-level engineering techniques, especially when n and m are at the order of millions and the size of E is orders of magnitude larger. Space complexity differs between techniques; some of them may require additional data structures to operate.

6.1.2 Set Containment Problem Description

A closely related problem to set intersection is set containment. Essentially, given two sets A and B, set containment indicates if one set is contained within the other, i.e. that one set is a subset of the other. Set containment can be tackled either by union-based methods or intersection-based methods. Since all the proposed GPU-enabled techniques are essentially set intersection techniques, the set containment problems are defined in a manner that is consistent with the aforementioned set intersection problems' rationale, as follows.

Single-instance set containment (SISC) problem: given i) a finite universe of elements E, and ii) a set $A = \{e_1^A, \ldots, e_n^A\}$ of size n and a set $B = \{e_1^B, \ldots, e_m^B\}$ of size m, where $e_i^{\{A,B\}} \in E$, set containment holds true if $|A \cap B| = min(n, m)$.

Multi-instance set containment (MISC) problem: this is also known as set containment join and is defined in manner similar to MISI. In the multi-instance set containment problem, given a collection of k sets $C = \{S_1, \ldots, S_k\}$, the goal is to find for each set pair among all $\binom{k}{2}$ of pairs (S_i,S_j), where $0 < i < j \leq k$, if the set containment condition holds true.

Two remarks based on the definitions above are as follows. Firstly, contrary to other common MISC definitions of set containment join, e.g., in [24], set containment is not restricted to refer only to a setting where the contained sets need to belong to a specific collection only, but it always checked whether the smaller set is contained into the larger one. The case where the two sets are of equal size is also covered: if, in that case, one set is contained into the other, this implies that the two sets are the same. Secondly, it is trivial to extend the set containment result not to be binary but, instead, to correspond to the *degree of containment* of the smaller set into the larger one: $|A \cap B|/min(n, m) \in [0, 1]$.

All techniques solving the SISI (resp. MISI) problems, can be used to solve the SISC (resp. MISC) problems as well; however the reverse holds only if SISC/MISC solutions compute the degree of containment, i.e., not just establishing whether one set is fully contained in the other one.

6.1.3 Scope of the present work

Over the past years, there has been a lot of research that encapsulates GPU techniques for the set intersection problem, as will be discussed shortly. In the majority of cases, the problem of set intersection is tackled in the context of triangle counting to accelerate graph analytics. Triangle counting is a special case of set intersection, which stems from the need to find quickly intersection counts among vertex adjacency lists of small lengths. As a result, the techniques proposed for triangle counting are tailored to specific algorithmic optimizations. On the other hand, , there is no existing work that investigates set containment on a single GPU either directly or indirectly to date. In the context of the present work, the motivation is twofold. Firstly, adapt and evaluate set intersection techniques are extracted, adapted and evaluated under a more demanding large set intersection scenario performing also a comparison against all methodologies proposed that can address the SISI and MISI problems for large sets. Secondly, set containment is built on top of the evaluated set intersection techniques in a single GPU setting, and is compared against the state-of-the-art CPU set containment solutions for the MISC problem.

6.1.4 Overview of existing solutions

In this section, details of existing set intersection and set containment techniques are discussed separately. The former refer to a GPGPU setting, while the latter assumes a traditional CPU-only environment, since to date, set containment has not been examined using GPUs. The relevant techniques are mostly presented in chronological order.

Set Intersection

Ding et al. [25] were the first to implement a parallel GPU set intersection algorithm. In their proposed solution, they use the so-called *Parallel Merge Find (PMF)* algorithm to compute a set intersection. In essence, given two sets, the shorter one is partitioned into disjoint segments,

with each segment assigned to a different GPU thread. Then, the last element of each segment is searched in the longer set to find the corresponding or closest positions for the partitioning of the longer set. As a result, each GPU thread becomes capable of computing its own intersection among segments in parallel.

In [105], Wu et al. highlight the inefficiency of the approach followed in [25] for sets of smaller size. Subsequently, they propose a GPU technique for set intersection that takes advantage of the fast on-chip shared memory. First, an empty array of size equal to the size of the shorter set is allocated in shared memory. Next, each GPU thread is assigned a specific number of elements from the shorter set and conducts binary searches in the longer set. If an element is found, its corresponding cell in the shared memory array is set to 1. Finally, a scan operation on the shared memory array is performed along with a stream compaction procedure, so that threads produce the final intersection.

In [106], Wu et al. extend their original work described above by introducing heuristic strategies to balance the workload across thread blocks more efficiently. Additionally, the proposal in [2] further extends the original work in [106] by introducing a linear regression approach to reducing the search range of a binary search and a hash segmentation approach as an alternative to binary search.

In [33], Green et al. present the *Intersect Path (IP)* algorithm for set intersection, which is a variation of the well established GPU *Merge Path* algorithm for merging lists as already discussed in Section 4.3.2. In addition, *IP* can be considered as an extension of *PMF* since it encapsulates a similar set partition logic. First, input sets are partitioned into segments that can be intersected independently by multiple thread blocks. Second, for each thread block, workload is distributed in such a way that near equal number of elements to intersect are allocated to threads. In [30], the authors extend the work of [33] by proposing an adaptive load balancing technique that dynamically assigns work to GPU threads based on work estimations. The authors of [76] and [97] follow a similar approach to set intersection. More specifically, their main concept is, for each GPU thread, to sequentially compute a set intersection by using a two-pointers merge algorithm. In the context of triangle counting, such an approach is applicable since the requirement is (i) for each GPU thread to compute a two-set intersection count independently, and (ii) these partial counts to be accumulated to compute the final global count. However, in the setting of set intersection over two large sets, their solution is inferior to GPU-based competitors and similar to a sequential CPU approach.

In [9], Bisson et al. propose a different approach, namely, GPU set intersection to be based on bitmaps and atomic operations. Given two sets, to compute their intersection, the GPU threads create the bitmap representation of the first set in parallel and then, iterate over the elements of the second set to search for the corresponding set bits. Based on the average set size, the workload allocation can be per thread, per warp or per block.

In [40], Hu et al. demonstrate that set intersection is faster though employing efficient binary search-based techniques than *IP*-based techniques, arguing that the latter suffers from nontrivial overhead of partitioning the input sets and non-coalesced memory accesses. On the other hand, their proposed algorithm optimizes binary search at a warp level to achieve coalesced memory access and to alleviate the need for workload balancing. In addition, by caching the first levels of the binary search tree they employ in the shared memory, they can achieve further speedup gains.

In [73], Pandey et al. propose a hash-based technique for set intersection. In brief, first, their algorithm hashes the shorter set into buckets, and then iterates over the larger set and hashes each element to the corresponding bucket. Afterwards, a linear search is conducted within each bucket to find the intersections.

Last, in the context of set similarity join, the authors of [80] propose a technique for conducting set intersections by using a static inverted index and atomic operations. By setting the similarity degree equal to zero, their solution can be used to solve the SISI/SISC and MISI/MISC problems.

As already mentioned, some of these works, such as [9, 33, 40, 76, 97] are more complete proposals targeting the triangle counting problem; in the present work the focus falls only to the part that is relevant to the SISI and MISI problems.

Set Containment

There are several solutions proposed for set containment in the literature. In [111], the authors, based on the set operators that they employ, classify existing solutions into two categories, namely (i) intersection-oriented solutions [13, 20, 24, 43, 51], and (ii) union-oriented solutions [58, 110]. Since no prior GPU solution for set containment exists, a brief overview of recent CPU solutions is given.

Intersection-oriented solutions. The core concept of intersection-oriented solutions is first to build an inverted index on the input collection of sets S, and then, for each set $s \in S$, to intersect all the inverted lists corresponding to the elements of s. The main advantage of intersection-oriented solutions is that they are verification-free. However, the main drawback is the dominant cost of the intersection of the inverted lists, especially in the case of long inverted lists. Hence, every intersection-oriented solution proposed aims to decrease this cost.

In [43], the authors propose a solution called *PRETTI*, which employs a prefix tree structure to collectively process input sets. The use of the prefix tree reduces the number of intersections of inverted lists among sets with the same prefix, thus amortizing the computational cost. In [59], the authors extend *PRETTI* by replacing the prefix tree with a more compact tree structure, called Patricia tree, in which all the nodes along a single path are merged into a single node. As a result, the total number of tree nodes traversed is decreased, improving the overall performance.

In [13], Bouros et al. propose a different solution, closer to set similarity join works, called *LIMIT*. Furthermore, *LIMIT* initially builds a shorter prefix tree of height h, considering only the first h elements of each set. After, the set containment join is conducted by using a filtering-verification framework. In brief, in the filtering phase, *LIMIT* either outputs set pairs directly if they share a common prefix and the set size is less or equal to h, or generates candidate pairs that are fully evaluated in the verification phase.

In [51], Kunkel et al. propose a solution, called *PIEJOIN*, in which they transform the prefix tree into linear arrays by using preorder ranks and preorder intervals. In addition, by enriching each tree node with the interval information, they can deduce efficiently whether a certain set element is contained in the corresponding subtree, without traversing it. The authors also introduce a parallel version of *PIEJOIN* using a shared-memory model. In the parallel version, the linearized prefix tree is partitioned based on the order in which tree nodes are stored. In their evaluation, they show that their parallelization efficiency depends heavily on input dataset characteristics.

In [24], Deng et al. propose a solution, called *LCJOIN*, in which they demonstrate a novel method to intersect all inverted lists simultaneously while traversing a prefix tree, and thus reducing the computational cost significantly. Furthermore, they introduce a data partitioning method in which they determine whether partitions should use the complete inverted index or construct a local index on-the-fly. In their evaluation, the authors show that *LCJOIN* outperforms most of the state-of-the-art set containment join solutions;[1] in the present work, GPU solutions are directly compared against an implementation of *LCJOIN*.

Last, in [20], Deep et al. investigate the use of matrix multiplication to accelerate the set containment join problem. Their solution, called *MMJoin*, first employs a data partitioning scheme in which input sets are classified as light or heavy ones based on their size. Similarly, set elements are also classified into light or heavy ones based on the inverted list sizes. Next, light and heavy sets with light set elements are directly processed using the inverted index, whereas heavy sets with heavy set elements are processed using matrix multiplication. In their evaluation, the authors show that matrix multiplication is highly conducive to achieving higher performance on dense datasets. As will be discussed in the next section, this approach is leveraged in the proposed solution, motivated also by the fact that matrix multiplication naturally lends itself to efficient parallelization on a GPU. In addition, the same concept is also applied to address set intersection.

Union-oriented solutions. The main rationale of union-oriented solutions is to generate signatures alongside the inverted index construction on the input collection of sets S. Then, for each set $s \in S$, the corresponding techniques generate candidate sets for evaluation by finding the union of the inverted lists for the relevant signatures. Finally, candidate set pairs undergo full verification to form the final output. The two dominant costs of union-oriented solutions are the cost of the union of inverted lists and the cost of the verification phase. As shown in [13] and [110], union-oriented solutions [37, 63] are outperformed by intersection-oriented solutions due to these costs. However, progress on improving union-oriented solutions, by integrating perks from intersection-oriented solutions, has been made recently [58, 110].

6.2 Techniques for Set Intersection

In the previous section, six different methodologies, based on (1) IP, (2) optimized binary search, (3) bitmap operations, (4) hashing, (5) set similarity joins and (6) matrix multiplication, have been identified respectively. Here, first, five different state-of-the-art GPU techniques for set intersection are presented, one for each of the first five methodologies in more detail. In addition, some modifications to these techniques are discussed in order to target large set intersection; which yield two novel hybrid solutions. The works presented in [76, 97] are not considered since they mostly rely on the preprocessing of input data and conduct intersection with a simple merge fashion algorithm. By contrast, techniques that are more adaptable in a general set intersection scenario are presented and evaluated. Next, it is explained how matrix multiplication can be leveraged for set intersection with an emphasis on the memory constraints posed by this approach. Overall, a concise presentation for each evaluated technique is given below followed by extensive experiments and discussion on the characteristics of the techniques based on the results.

[1] There is no published comparison between [24] and [20].

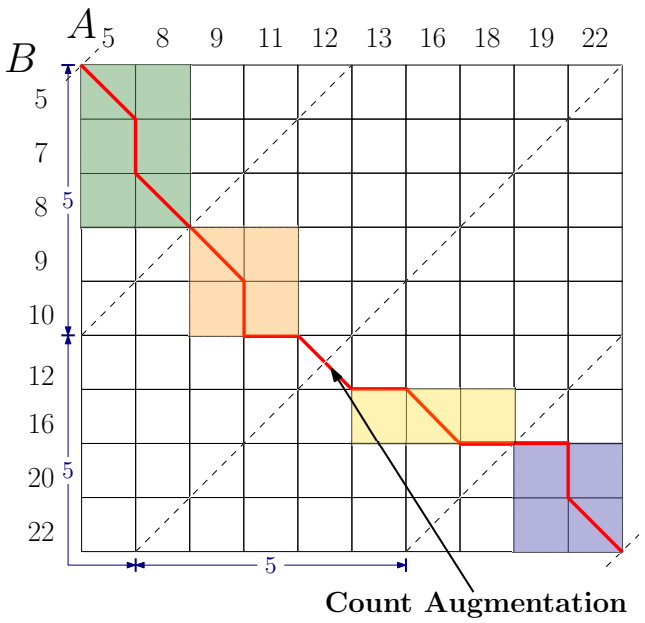


Figure 6.1: Intersect Path example using 4 thread blocks.

6.2.1 GPU Implementations of the different methodologies

Intersect Path (IP) [33]

As already discussed in Section 4.3.2, given two ordered sets A and B, *IP* considers the traversal of a grid, noted as *Intersect Matrix*, of size $|A| \times |B|$. Beginning from the top left corner of the grid, the path can move downwards if $A[i] > B[j]$, to the right if $A[i] < B[j]$, and diagonally if $A[i] = B[j]$, until it eventually reaches the bottom right corner. There are two partitioning stages, one on the kernel grid level and the other on the block level. On the grid level, equidistant cross diagonals are placed on the Intersect Matrix. The number of diagonals is equal to the number of thread blocks plus one, in order to delimit the boundaries of each block. Using binary search, the point of intersection between a cross diagonal and the path is found. As a result, each thread block is assigned to intersect disjoint subsets of the input. In case a cross diagonal intersects with the path inside a matrix cell, a count augmentation is required beforehand, since otherwise, this intersection would not be assigned to any thread block. An example of this scenario is shown in Figure 6.1. On the block level, the same cross diagonal approach applies in order to distribute workload among threads. Each thread may conduct a serial or binary search-based intersection.

Optimized Binary Search (OBS) [40]

Given two ordered sets A and B, *OBS* caches the first levels of the binary search tree of the larger set in shared memory to reduce expensive global memory reads. For example, as shown in Figure 6.2, the higher level nodes of the binary tree reside in shared memory, whereas the leafs are located in global memory. The smaller set, which is B in the example, is used for lookup and, as a result, there are $|B|$ total binary search lookups. Each thread is assigned a lookup and, in each iteration, up to

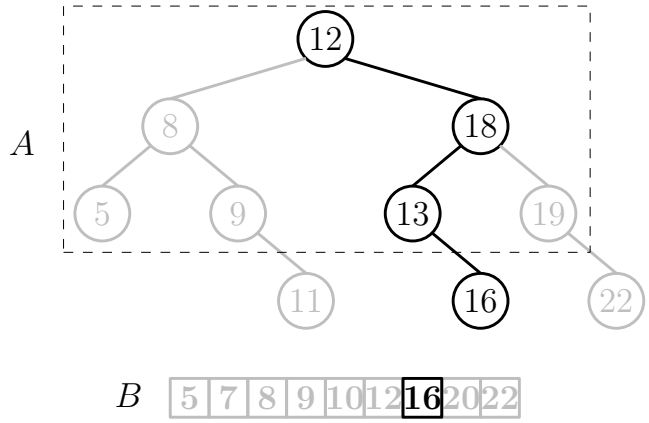


Figure 6.2: Optimized Binary Search example.

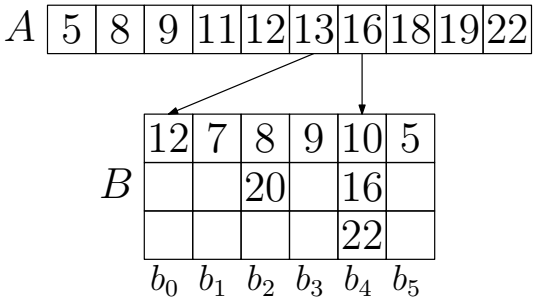


Figure 6.3: Hash-based Intersection with six buckets of size three each and $h(x) = x\%6$.

32 (i.e., the size of warp) lookups are executed simultaneously. For the multi-instance problem, a simple optimization is to cache each set to shared memory one at a time, and then iterate over every subsequent set to perform the intersection for each pair. This requires sets to be sorted by their size.

Hash-based Intersection (HI) [73]

Given two sets A and B, *HI* first hashes the shorter set and constructs buckets in parallel, and then, iterates and hashes every element of the larger set into the corresponding bucket as already described in Section 6.1. In case of a collision, a linear search is conducted to find whether an entry with equal value exists in the bucket. An example of *HI* is shown in Figure 6.3. The initial hashing of the smaller set is preferred in order to reduce the number of collisions. Buckets are statically allocated once in the global memory space. The size of each bucket, i.e. the number of entries from the shorter set in the bucket, is stored in shared memory. Thus, to ensure correctness for the MISI problem, only the buckets sizes in shared memory are need to cleared, as every next set hashing will overwrite the previous one.

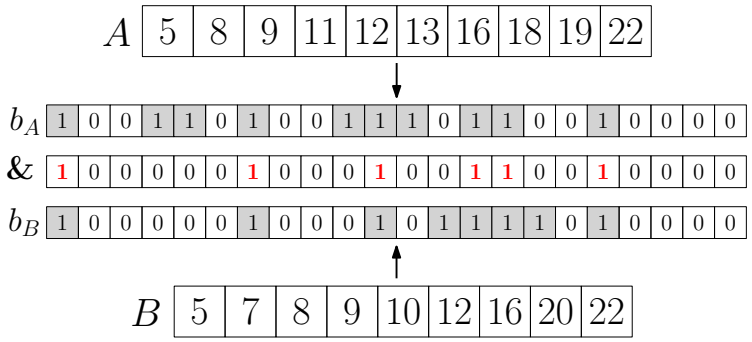


Figure 6.4: Naive Bitmap-based Intersection example.

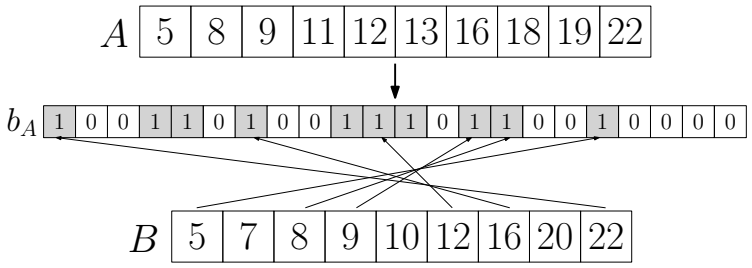


Figure 6.5: Dynamic Bitmap-based Intersection example.

Bitmap-based Intersection (BI)

Given two sets A and B, *BI* conducts the set intersection on their bitmap representations, with each bitmap requiring $|E|/8$ bytes of memory regardless of the set sizes. More specifically, there are two flavors of *BI*, namely (i) naive, where all bitmaps fit in global memory, and (ii) dynamic, where bitmaps are built on the fly in the case that all of them cannot be stored in the available global memory. The latter is similar to the work of [9]. Bitmaps are constructed in parallel using the `atomicOr` function. In the naive scenario, each GPU thread fetches two bitmap words, one from each set, and conducts a `popcount` operation on the resulting logical `AND` word to compute the subset intersection. An example of the naive scenario is shown in Figure 6.4. In the dynamic scenario, each GPU thread block first constructs the bitmap representation of the current set, and then, for every next set, the threads iterate over its elements and check whether the respective bits are set (see the example in Figure 6.5).

Static-index Intersection (sf-gssjoin) [80]

As already introduced in Section 5.1, given k sets, *sf-gssjoin* first constructs a static inverted index over all set elements. For each set, the inverted lists for all its elements are concatenated in a logical vector. Next, this vector is partitioned evenly with each partition assigned to a specific GPU thread. Thus, each thread processes independently its corresponding partition and contributes to the intersections of the current set in cooperation with other threads. To ensure correctness, threads increment the intersection counts by using atomic operations. An example of *sf-gssjoin* is shown in Figure 6.6.

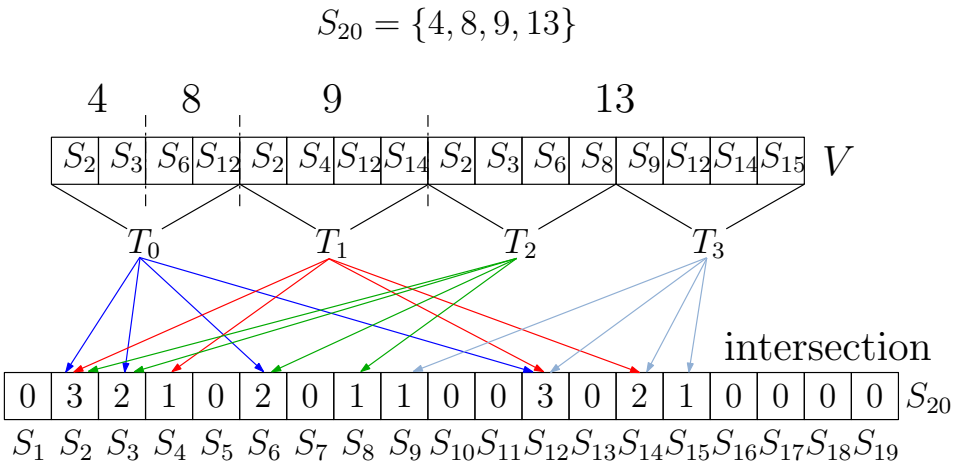


Figure 6.6: Static-index Intersection example using 4 GPU threads.

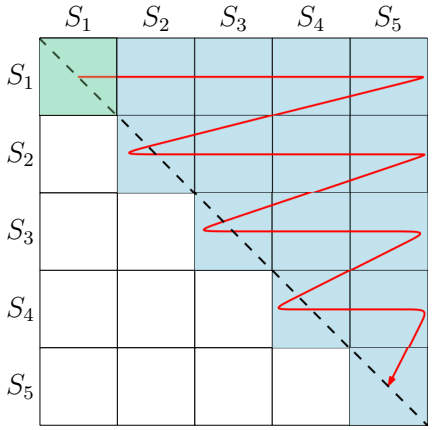


Figure 6.7: Process the multi-instance scenario incrementally by using a tile-based approach.

Discussion and Hybrids

All of the presented techniques with the exception of *BI-naive* and *IP*, conduct a single instance set intersection on a single block or warp. This results in severe GPU under-utilization, especially for sets of size in the order of millions, since a single execution unit is assigned with the complete workload. To tackle this issue, the integration of the kernel grid level partitioning of *IP* with *OBS* and *HI*, which have not been proposed previously in the literature, is investigated. The resulting novel hybrid techniques are noted as *IP-OBS* and *IP-HI* respectively.

In addition, for the single instance scenario, in *BI-dynamic*, the algorithm is modified by constructing the bitmap of the first set across multiple blocks and then by partitioning evenly the second set into disjoint subsets with each one assigned to a specific block.

Finally, a common problem encountered in GPGPU computing is the need to process data that do not fit into GPU memory. In the setting of MISI, it is required to compute and store $\binom{k}{2}$ intersections. In addition to the space required to store the intersections, there is also the prerequisite

of sufficient memory space so that each technique can store its auxiliary data structures. Therefore, it becomes apparent that for large values of k the complete result cannot be computed in a single GPU pass. In such cases, the simplest approach to process the multi-instance scenario is to partition the output into tiles and conduct set intersection incrementally by invoking the GPU for each tile. An illustration of this approach is depicted in Figure 6.7.

6.2.2 Leveraging Matrix Multiplication for Set Intersection

A different approach to evaluating set intersection involves the use of matrix multiplication. Essentially, by representing a collection of k-sets S as a binary $k \times |E|$ matrix M_S, the intersections for the MISI problem can be computed by multiplying M_S with its transpose matrix M_S^T. An example of using matrix multiplication for the MISI problem with $|S| = 5$ and $|E| = 6$ is shown in Figure 6.8.

In general, multiplying two matrices of size $n \times n$ naively, requires $O(n^3)$ time when done in a simple but easily parallelizable manner. There has been a plethora of research on fast matrix multiplication. Due to their massive parallelism, assisted by the simplicity of the numerical computations of the problem, GPUs excel at matrix multiplication, achieving significant speedups over CPU implementations [42].

However, in the context of MISI, certain restrictions must be met in order to be able to conduct set intersection as a matrix multiplication operation efficiently. The most prominent concerns the required $O(k|E|)$ memory to represent S as a binary matrix M_S. For example, consider a set collection S with $k = 10^3$ and $|E| = 10^7$. The memory required to store M_S as a matrix of floats is approximately 40GB which exceeds greatly any GPU memory. In addition, the required $O(k^2)$ memory to store the output must also be taken into account, which imposes further restrictions on the datasets that matrix multiplication-based set intersection can process on a GPU. The latter restriction is addressed through invoking multiple kernel calls, as explained previously. Orthogonally to multiple kernel calls, the binary matrix could be split into sub-matrices in case that it does not fit into the main memory. However the memory allocation and clearing overheads associated outweigh any benefits, compared also to other solutions; thus, such a solution is not employed in practice. This is further discussed in the subsequent evaluation section.

The main consequence of this approach is that matrix multiplication ceases to be applicable in datasets with very large element universe size. Overall, the time complexity for matrix multiplication (MM) in the MISI scenario is $O(k^2|E|)$, whereas the required memory is $O(k^2 + k|E|)$. Hence, the factors of k and $|E|$ dictate if MM can be employed for set intersection in a single run, whereas, if M_S can fit into the GPU's main memory, large k values result in multiple kernel calls.

A final note is that, since the implementations operate on binary matrices, the `SGEMM` operation provided by `cuBLAS` library [19] is used, after performing all adaptations required to incorporate it into the framework.

6.2.3 Evaluation

In this section, the techniques presented in the previous sections are evaluated, while the experiments also include CPU variants for completeness. More specifically, the GPU techniques are

$$\begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{array}\!\!\begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array} \times \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} s_1 & s_2 & s_3 & s_4 & s_5 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} \end{array} = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} s_1 & s_2 & s_3 & s_4 & s_5 \end{array} \\ \begin{pmatrix} 2 & 1 & 0 & 1 & 2 \\ 1 & 3 & 1 & 1 & 2 \\ 0 & 1 & 3 & 2 & 1 \\ 1 & 1 & 2 & 3 & 2 \\ 2 & 2 & 1 & 2 & 4 \end{pmatrix} \end{array}$$

Figure 6.8: Using matrix multiplication for set intersection.

Dataset	Cardinality	Avg set size	Element Universe
BMS-10K	$1 \cdot 10^4$	35	
BMS-50K	$5 \cdot 10^4$	22	$1.6 \cdot 10^3$
BMS-100K	$1 \cdot 10^5$	17	
ENRON-10K	$1 \cdot 10^4$	794	
ENRON-50K	$5 \cdot 10^4$	374	$1.1 \cdot 10^6$
ENRON-100K	$1 \cdot 10^5$	258	
ORKUT-10K	$1 \cdot 10^4$	1001	
ORKUT-50K	$5 \cdot 10^4$	984	$8.7 \cdot 10^6$
ORKUT-100K	$1 \cdot 10^5$	905	
TWITTER-10K	$1 \cdot 10^4$	150	
TWITTER-50K	$5 \cdot 10^4$	142	$3.7 \cdot 10^4$
TWITTER-100K	$1 \cdot 10^5$	140	
WORDS-10K	$1 \cdot 10^4$	3494	
WORDS-50K	$5 \cdot 10^4$	2396	$1.1 \cdot 10^4$
WORDS-100K	$1 \cdot 10^5$	2021	

Table 6.1: Real world dataset characteristics.

compared against three CPU alternatives, namely, (i) *SIMD*, which performs intersection on sorted integers using SIMD instructions [53], (ii) *std::set_intersection*, which uses the C++ standard library and (iii) *boost::dynamic_bitset* which uses the Boost library in a similar fashion as *BI-dynamic*. In addition, for the multi-instance problem, the CPU solutions are run in parallel by using OpenMP with 12 threads. The experimental setup used is the same as describe in Section 4.4, with a single change the increase of main memory to 72GB, in order to support larger matrix multiplications.

First artificial datasets are used, where set elements follow the normal, uniform and zipf like distribution. To distinguish each dataset, the notation *Distribution-Element universe-Average set size* is used. The element universe is varied from 10^8 up to 10^9. Respectively, the average set size is varied from 10^6 up to 10^7. For the multi-instance scenario, four real world datasets previously employed in Section 5.2 and one from [104], are also used. More specifically, for each sorted dataset the $k \in \{10000, 50000, 100000\}$ largest sets are extracted. Table 6.1 gives an overview of the real world datasets' characteristics.

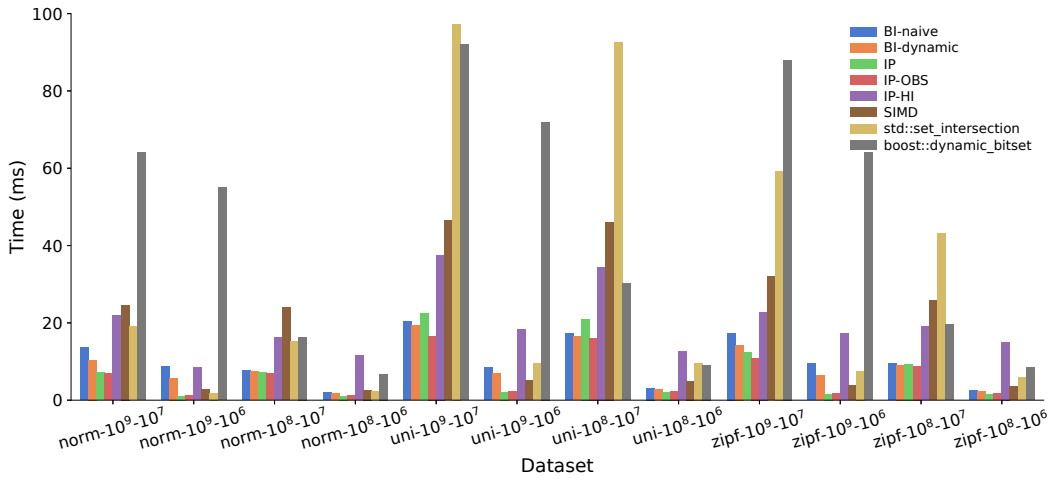


Figure 6.9: Single-instance intersection experiments on artificial datasets.

SISI experiments

The single instance scenario is examined using twelve artificial datasets (combinations of 3 distributions, 2 element universe sizes and 2 set sizes). Several block sizes have been used, and the results of the best configuration for each technique is presented.

As shown in Figure 6.9, *IP* and *IP-OBS* are the clear winners for the SISI experiments, in the sense that one of them is the best overall performing technique across every dataset. Moreover, the differences between their performance is relatively small: up to 12% for the normal and zipf distribution, and up to 35% for the uniform distribution. Overall, these techniques achieve average 2.3X speedup compared to the best performing CPU technique; the best performing CPU technique differs between the cases, but in average, *SIMD* is the most robust CPU technique. In general, the speedup is similar for all distributions types and increases with higher universe and average set size, reaching 2.92X. The speedup of *IP* or *IP-OBS* over the worst performing CPU or GPU technique exceeds an order of magnitude; speedups up to 48.67X over a CPU technique and up to 10.97X over a GPU technique can be observed.

In addition, *IP-HI* exhibits the worst performance regarding GPU techniques and seems unable to perform better than CPU for many settings; this shows that simply relying on the *IP* workload allocation rationale is not adequate. Finally, *BI*-based techniques behave better for smaller universe sizes and non-very sparse bitmap representations; as the sparsity in the bitmap representation increases, solutions like *IP* or *IP-OBS* that operate directly on set elements instead of bits are dominant by a large margin.

Extensive profiling was conducted using the official tool provided by NVIDIA, `nvprof`, to explain the observations above[2]. The higher performance of *IP-OBS* especially when compared to *BI* is mainly attributed to the higher significance of IPC, which stands for Instructions per cycle, (and related metrics) in the problem as shown in Figure 6.10. The problem is not arithmetic operation-intensive but involves several complex inter-dependencies between computations and

[2]The full profiling data are provided along with the source code.

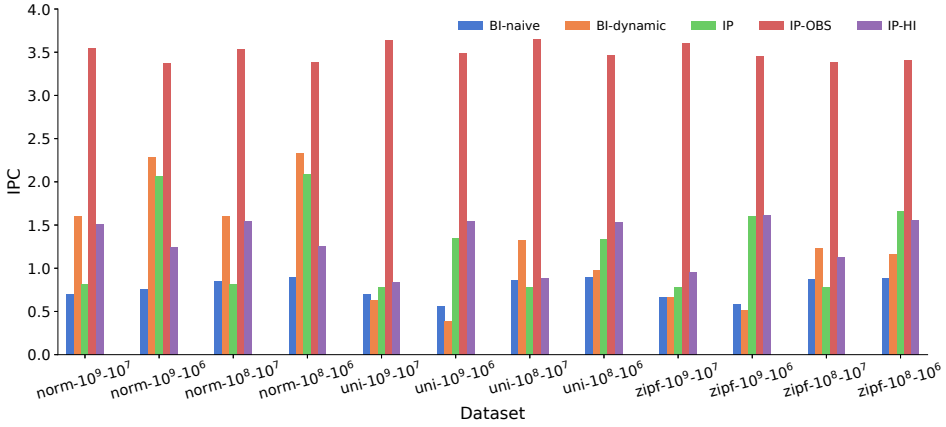


Figure 6.10: SISI IPC profiling (higher is better).

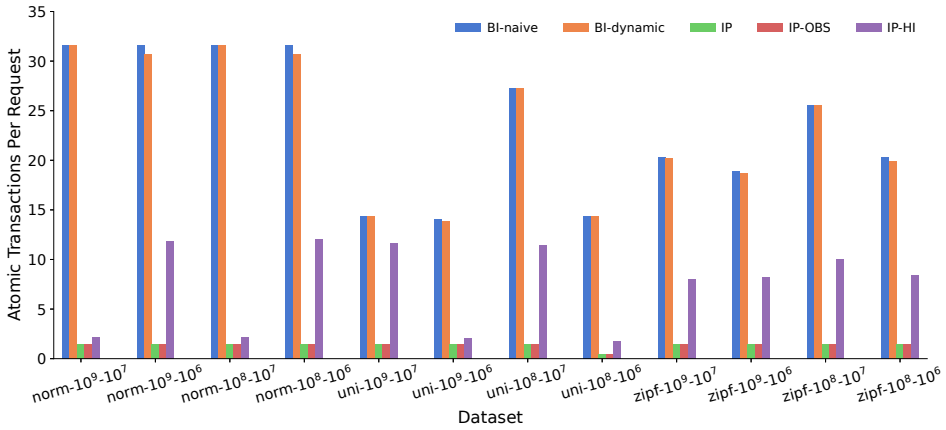


Figure 6.11: SISI atomic transactions per request (lower is better).

accesses to memory spaces; i.e., as a GPU program, it is mostly latency-bounded and as such, traditional efficiency metrics like flops do not apply. *BI*-based solutions have a larger memory footprint because they create the bitmap signatures for both sets, which incurs an overhead. However, IPC alone cannot explain the similar performance between *IP* and *IP-OBS*. Figure 6.11 reveals that both these techniques excel at atomic transactions per request. According to the documentation, the optimal target ranges between 1-2 transactions per request and this is achieved by these two techniques only [69]. Finally, *IP-HI* suffers, except the higher number of atomic transactions per request, from a lot of bank conflicts in shared memory and minor register spilling due to collisions and increased number of random memory accesses in larger sets; this explains that it behaves worse than the other *IP*-based variants.

The main lesson learned is that crafting efficient GPU techniques for the SISI problem is a complex task that requires deep understanding of the interplay between all latency aspects in GPU programming. Counter-intuitively, commonly used metrics may be misleading. For example, *IP-OBS* has a much higher number of instructions issued, but these instructions manage to exe-

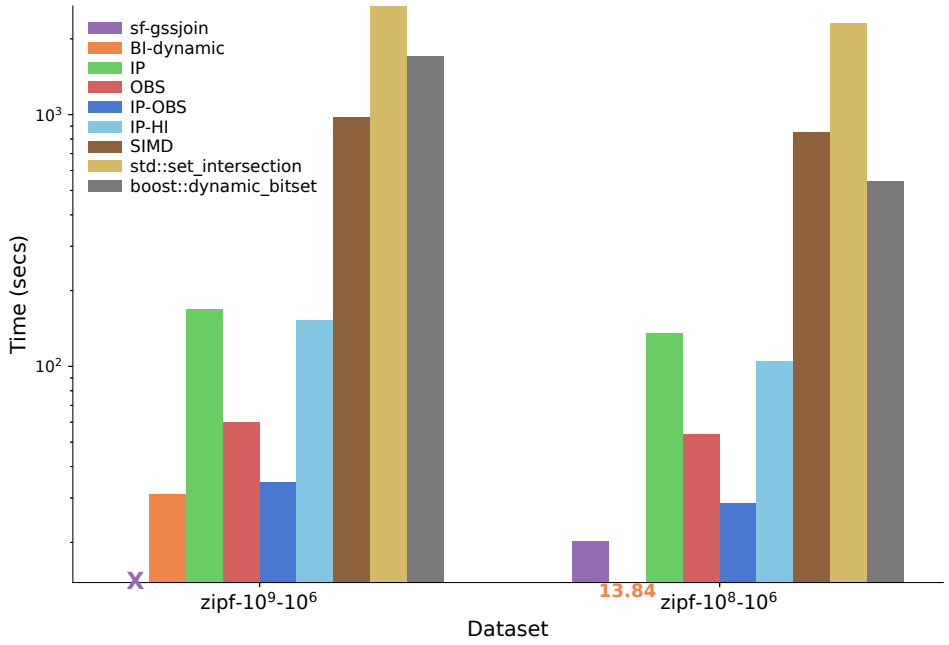


Figure 6.12: Multi-Instance intersection experiments on artificial datasets with $k = 1000$. The X symbol indicates when the experiment could not run due to memory shortage. When the bars are too short, the running times are also depicted.

cute fast; similarly, metrics such as memory throughput and utilization metrics are insufficient to explain the behavior of the hybrid techniques. Overall, what matters most is finding a beneficial equilibrium between the amount and type of computations and memory-related bottlenecks that these computations may trigger.

MISI experiments

For the multi-instance evaluation, two sets of experiments were conducted. In the first one, two artificially generated datasets consisting of large sets are used; both follow the zipf distribution, which is the closest to real world. In the second one, real world datasets are used and evaluate the intersection techniques on smaller set sizes.

As shown in Figure 6.12, in both datasets examined with $k = 1000$, *BI-dynamic* is the best performing technique and achieves average 10X speedup compared to the best performing CPU technique. Furthermore, *sf-gssjoin* is the second best performing technique, when manages to launch, i.e. when the static index fits in the global memory, which is not the case when the element universe is 10^9 in the experiments. On the other hand, *IP-OBS* is more robust and its performance is the closest to *BI-dynamic* across both datasets. In addition, a 80% performance increase for the hybrid *IP-OBS* technique over the standalone *OBS* is observed. Furthermore, even though *IP* and *IP-HI* are the worst performing GPU techniques, they manage to achieve 1.57X speedup compared to the best performing CPU technique on average. Last, the inability to launch *MM* due to the large element universe, which requires an excessive amount of memory, is also noted; in

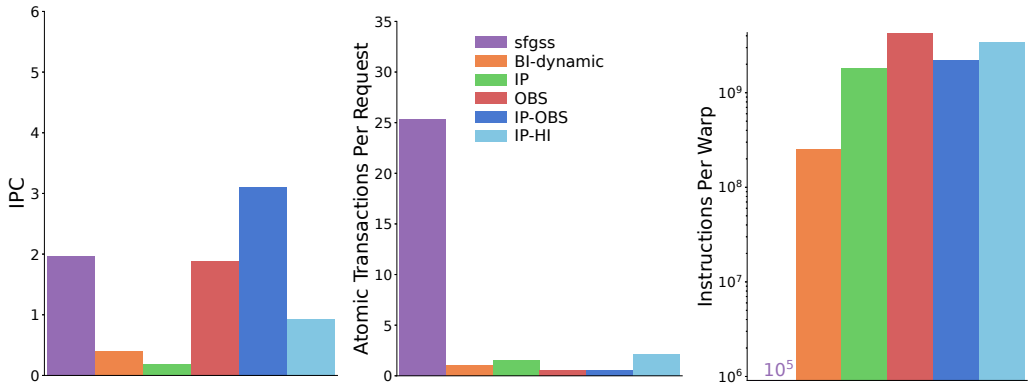


Figure 6.13: MISI profiling for zipf-10^8-10^6 dataset: IPC (left), Atomic Transactions per request (middle), Instructions per warp (right).

such a case, if the matrix multiplication were to be performed through splitting the binary matrix into sub-matrices, the associated overhead of allocating GPU memory would be several orders of magnitude higher than the slowest CPU technique.

In Figure 6.13 some additional profiling information referring to the right part of Figure 6.12 is presented. It can be observed that the dominant behavior of *BI-dynamic* in this dataset is attributed to the *combination* of two factors, namely low number of atomic transactions per request and not very high number of instructions per warp, which outweighs the relatively low IPC number. Out of these factors, the atomic transactions is the most important one and this explains the fact that *sf-gssjoin* is inferior to *BI-dynamic* despite its much lower instructions per warp. In the SISI case, *BI* solutions allocated a single set intersection computation to multiple blocks; otherwise the utilization would be prohibitively low. In the MISI case, a single set intersection is allocated to a single block, which leads to less atomic transactions per request. The impact of the atomic transactions is mitigated when the set sizes are smaller, as in the real-world datasets discussed below. The other low-level remarks when discussing the SISI case still hold.

As shown in Figure 6.14, for the real world datasets, there are two clear winners, namely *MM* and *sf-gssjoin*. *MM* achieves on average a 22X speedup over the best performing CPU technique in the cases where it is the best performing technique. Correspondingly, *sf-gssjoin* achieves a 25X speedup when it is the most efficient technique. In other words, GPU solutions outperform parallel CPU solutions in the experimental setting by more 23.5X on average. In Appendix B, exact numbers and more cases in terms of dataset sizes are presented.

The reason behind *MM*'s efficiency is twofold and relates to the dataset characteristics. Firstly, for the BMS dataset, the small size of the element universe alone, which is at the order of 10^3, suffices to render matrix multiplication the most advantageous approach. Secondly, for the WORDS dataset, the reason behind *MM*'s superiority is that both the element universe size is rather small (order of 10^4) and the average set size only an order of magnitude smaller, hence the increased density of this dataset favors *MM* in contrast to techniques that operate directly on set elements. As already discusses, such density also benefits *BI-dynamic*. As datasets become more sparse, the efficiency of *MM* deteriorates, as it can been seen for the TWITTER dataset, where *MM* performs worse than the rest of GPU techniques despite the fact that the element universe is not that large

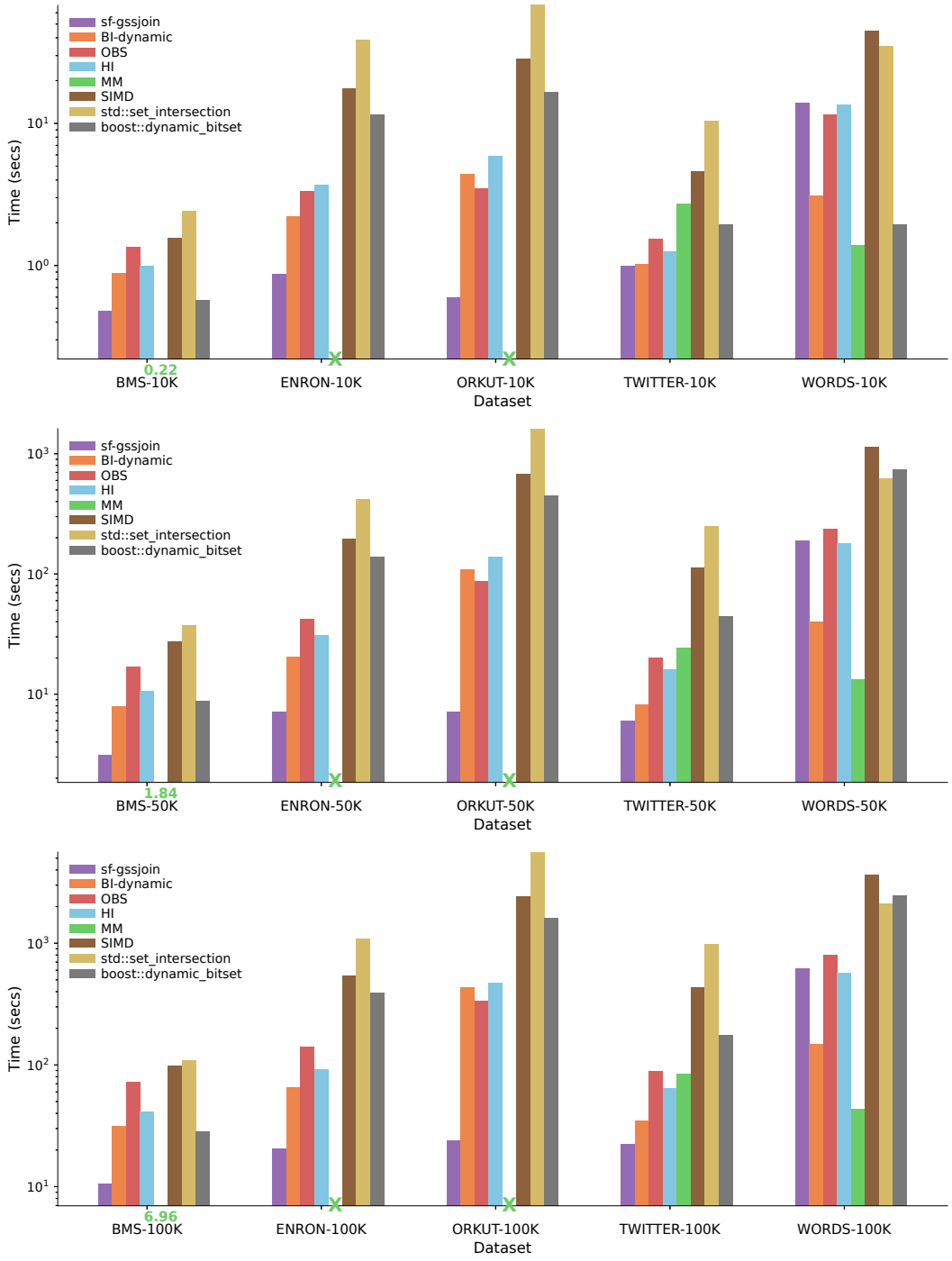


Figure 6.14: Multi-Instance intersection experiments on real world datasets.

(same order of magnitude as WORDS). In addition, for larger element universes, i.e. ENRON and ORKUT, *MM* is unable to launch, since the required memory to store the binary matrices exceeds the available GPU memory and is clearly inferior to resort to sub-matrix multiplications.

On the other hand, *sf-gssjoin* remains more robust throughout different dataset characteristics. Moreover, in its winning cases, the small average set size leads in small static indices, which yields an optimal workload distribution among GPU threads and overall, results in better GPU utilization. In contrast, *BI-dynamic*, *OBS* and *HI* conduct set intersection at block level. Thus, there is an inherent workload imbalance among GPU thread blocks. It must be noted that *IP*, and consequently its two workload allocation rationales, *IP-OBS* and *IP-HI* cannot always execute due to memory constraints. More specifically, the required memory to store the diagonals for the grid level partitioning of *IP* is $\binom{k}{2} \times (blocks + 1)$. As a result, there is an upperbound to the number of blocks to comply with the global memory restriction. However, due to the small set sizes, *IP* partitioning is considered as an excessive approach to conduct set intersection on these datasets.

Based on the experimental evaluation, the conclusion is that for very large MISI problems in terms of set size, *BI-dynamic* and *IP-OBS* are preferable. On the other hand, when dealing with multiple instance smaller (but still large) set intersections, the grid level partitioning of *IP* adds overhead and results in severe GPU under-utilization. In such cases, standalone *OBS* and *HI* perform better but are surpassed by either *MM* or *sf-gssjoin*. More specifically, *MM* is the dominant solution for small element universe sizes (order of 10^3) or for larger element universe sizes and dense datasets provided that the binary matrix fits into the main memory of the GPU; otherwise *sf-gssjoin* is the best performing technique. Finally, as in the SISI case, the development of efficient GPU techniques should place emphasis on the atomic transactions along with aspects, such as IPC.

6.3 Testing set containment joins using also co-processing

In this section, the set containment join problem in the GPGPU setting is investigated. As mentioned in Section 6.1, all techniques presented so far can solve the SISC/MISC problems with negligible differences in runtimes due to the extremely lightweight extra computation needed to compute the degree of containment. Also, as already discussed, no GPU-based solution for the set containment join has been proposed so far. Therefore, focus falls on CPU techniques that can be fully or partially transferred to the GPU. To this end, *MMJoin* [20] is selected, since it is the most prominent technique that can be accelerated with the GPU. Next, an overview of *MMJoin* is given, along with the modifications required to yield a co-processing CPU-GPU implementation. Afterwards, the GPU set intersection techniques presented in the previous section are evaluated and adapted for the set containment join, against the CPU-standalone *MMJoin* and its GPU-enabled version. Last, experiments regarding the comparison against the second state-of-the-art (binary) set containment solution, namely LCJoin [24], are also presented and discussed.

6.3.1 Matrix Multiplication Set Containment Join (MMJoin)

Given a collection of k sets S, *MMJoin* evaluates the set containment join problem as a join query with projection using an output-sensitive algorithm. More specifically, *MMJoin* uses a twofold data partitioning scheme, parameterized by two integer constants $\Delta_1, \Delta_2 \geq 1$, to mitigate the

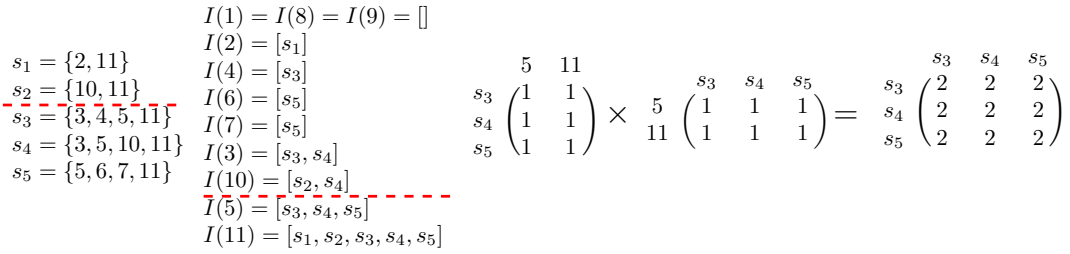


Figure 6.15: An example of the MMJoin algorithm with $\Delta_1 = \Delta_2 = 3$.

memory allocation-related computational cost in fast matrix multiplication. First, each set $s \in S$ is classified as light if $|s| < \Delta_1$, or heavy otherwise. Similarly, each set element $e \in E$, with $I(e)$ its corresponding inverted list, is classified as light if $|I(e)| < \Delta_2$, or heavy otherwise. To process the set containment join, *MMJoin* uses a static inverted index to compute set intersections between (i) light sets for both light and heavy set elements and (ii) heavy sets for light set elements. For the the heavy elements of the heavy sets, intersections are computed via matrix multiplication. The reason for this is that computing the intersections by accessing the inverted lists of the heavy set elements incurs a high computational cost, while matrix multiplication can mitigate this cost. An example of the *MMJoin* algorithm is depicted in Figure 6.15.

In the original implementation of *MMJoin*, the authors use the Eigen library [26] to perform matrix multiplication on the CPU. Since GPUs support fast matrix multiplication, *MMJoin* is modified by offloading the complete matrix multiplication operation to the GPU, i.e., *MM* is partially applied. Furthermore, the order in which the set containment join is processed, is also modified. More specifically, first the complete inverted index is built on the CPU, and then all light sets are processed. After, the GPU is invoked for fast matrix multiplication, in a similar fashion to the MISI problem, in order to process the heavy set elements of the heavy sets. Last, the light set elements of the heavy set are processed on the CPU using the inverted index.

6.3.2 MISC EXPERIMENTS

The best performing GPU set intersection techniques, as presented in Section 6.2.3, adapted for set containment, are evaluated against two flavors of *MMJoin*[3], (i) *MMJoin-CPU* which runs completely on the CPU, and (ii) *MMJoin-GPU* which invokes the GPU for the matrix multiplication. In addition, both *MMJoin-CPU* and *MMJoin-GPU* run in parallel by using OpenMP with 12 threads. The same artificial and real world datasets, as in Section 6.2.3, are employed. For the majority of datasets, the original cost optimizer is used to calculate the Δ_1, Δ_2, as presented in [20]. For the ENRON and ORKUT datasets, Δ_1, Δ_2 are set manually so that the input binary matrices fit in the GPU memory. Since the GPU set intersection techniques store the complete $\binom{k}{2}$ intersections in main memory, the binary set containment or the set containment degree can be easily calculated in a linear pass directly.[4]

As shown in Figure 6.16, for both artificial datasets with $k = 1000$, *BI-dynamic* is the best performing technique and achieves average 8.6X speedup compared to the *MMJoin-GPU*. For real

[3] The source code of *MMJoin* was provided by the authors of [20].
[4] SISC experiments do not differ from MISC and are thus omitted.

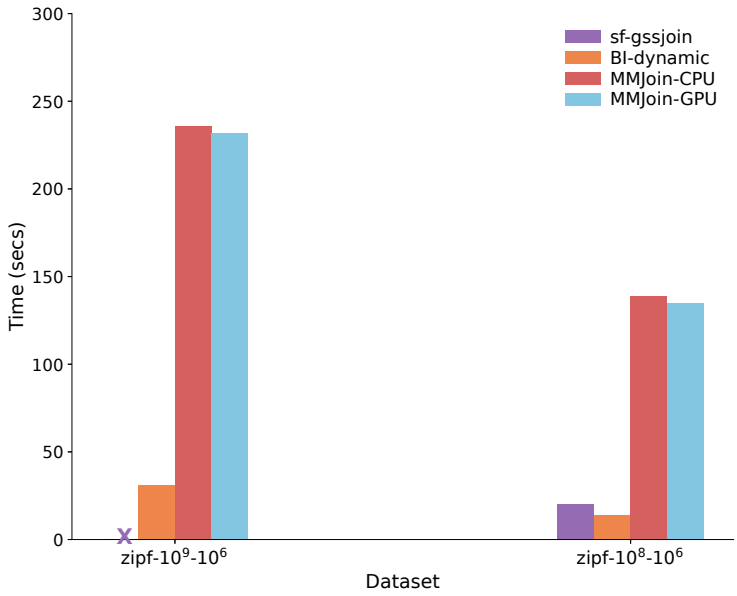


Figure 6.16: Multi-Instance set containment experiments on artificial datasets.

world datasets, as shown in Figure 6.17, in the majority of the experiments, the adapted GPU set intersection techniques are more efficient and on average, they achieve a 2.3X speedup compared to *MMJoin*.

The main observation drawn from Figure 6.17 is that all the conclusions drawn from the MISI experiments, as summarized at the end of Section 6.2.3 still hold with three exceptions, where *MMJoin* seems superior. Each of these cases are discussed in turn. Firstly, parallel *MMJoin-CPU* yields the lowest running times for TWITTER-10K. However, the actual times must be considered; these are 0.52 secs compared to 0.99 secs achieved by *sf-gssjoin*; i.e., the differences in actual runtimes are very small if not negligible. The same applies for the TWITTER-50K.[5] The exact times are given in the appendix. The third case, in which *MMJoin* beats GPU-only solutions is for WORDS-100K. This case needs to be discussed in more detail. *MMJoin-GPU* runs faster than *MM* because the binary matrix is very large and not all parts of it are dense; therefore allocating its sparser parts to CPU, to be processed through the inverted index, is beneficial.

For completeness, an implementation of *LCJOIN* following the algorithm presented in [24] is compared against the rest of the evaluated solutions. However, as shown in Figure 6.18, the GPU-based techniques are faster by two orders of magnitude.[6]

In summary, the main conclusion from these experiments is that GPU-based set intersection techniques can be employed to solve set containment joins more efficiently that the current CPU-based state-of-the-art, and in specific large cases, co-processing solutions need to be considered in addition to techniques that run on a GPU exclusively.

[5] In TWITTER-50K, all 50K sets are classified as heavy; also, out of the 37704 set elements, only 421 are classified as heavy and the rest are classified as light.

[6] The implementation of *LCJOIN* is included in the source code repository; the implementation of the authors in [24] was not provided.

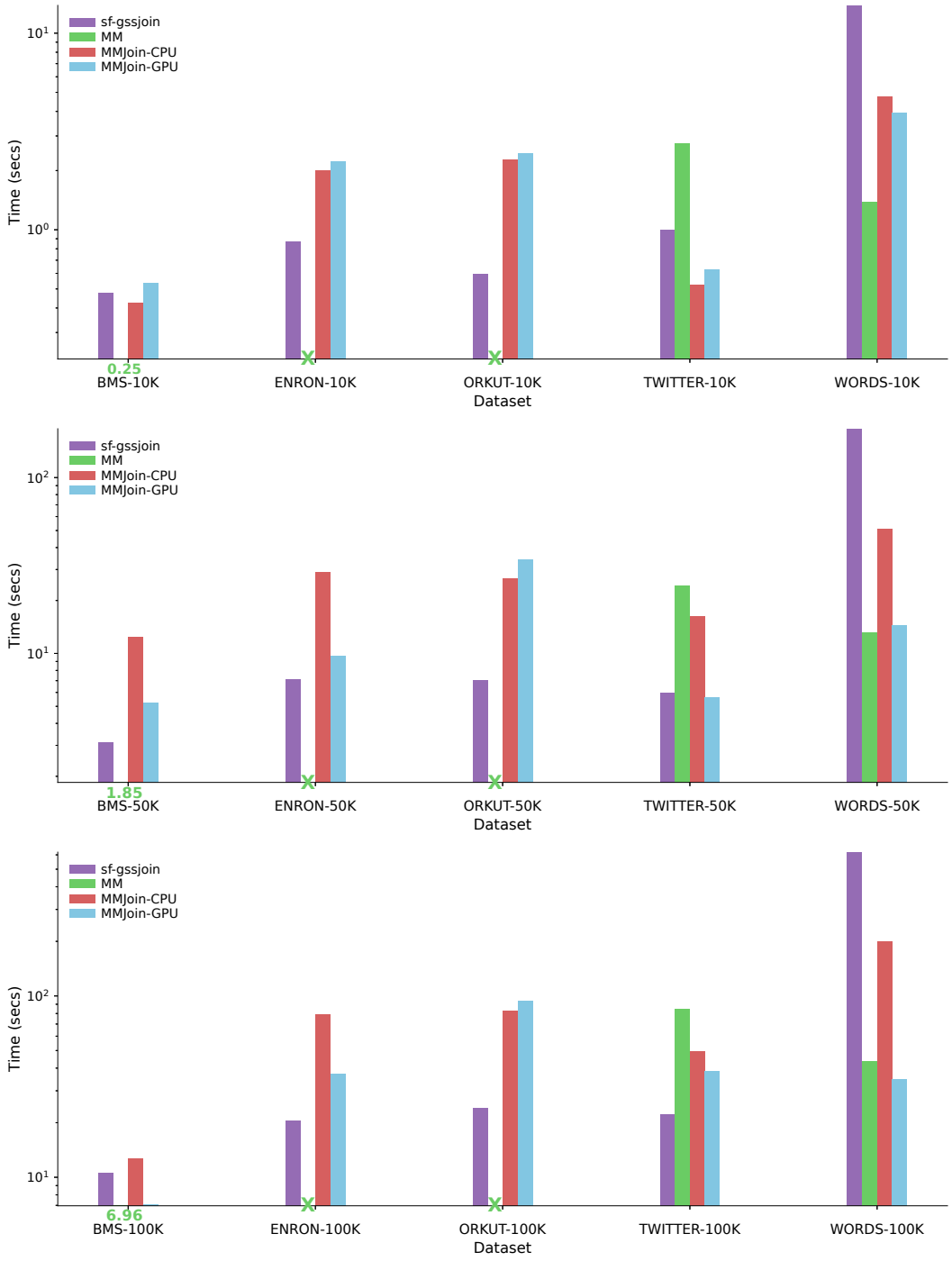


Figure 6.17: Multi-Instance set containment experiments on real world datasets.

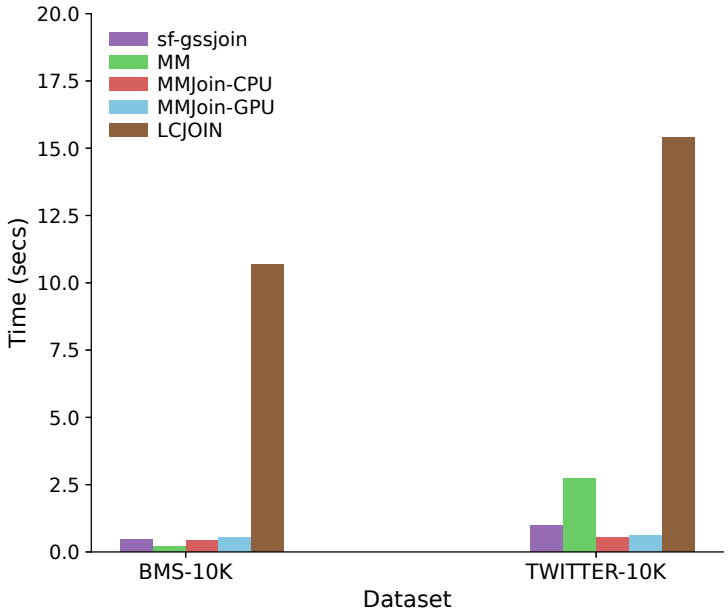


Figure 6.18: Comparison of LCJOIN against GPU-based solutions.

6.4 Remarks

In this chapter, a wide variety of approaches to large set intersection using a GPU was investigated. A variety of GPU set intersection techniques that were previously applied to graph analytics were of adapted and evaluated, and although these techniques are better suited for intersecting sets of smaller size, it is experimentally shown that, through certain enhancements, they can be easily adapted for large set intersection. Furthermore, the best approaches in the single- and multi-instance cases are explained, and a novel hybrid, namely, combining IP with OBS, is introduced which proves to be a dominant solution for the former cases, and competitive in the latter ones. Also, employing bitmap-based solutions pays off in the multi-instance case when set sizes are very large, while, if the set sizes are relatively smaller, multi-instance set intersection stands to benefit from either adapting a set similarity join technique or employing matrix multiplication. In addition, it is shown experimentally that these techniques are superior to the existing state-of-the-art techniques for set containment join and can further benefit from CPU-GPU co-processing.

Overall, the presented work covers all the known competitive GPU-enabled approaches to large set intersection. The evaluation however is conducted using a single GPU. A direction for future work is to exploit multiple GPUs for even larger problem instances.

7 Conclusion

Over the past years, the GPGPU paradigm has emerged as a key technology for high-performance computing. Furthermore, the key characteristic of GPGPU is the massive parallelism that GPUs offer at low cost. This has led to tremendous speedups in many applications that require intensive numerical computations and can be parallelized easily.

On the other hand, for more complex and advanced data management problems, performance speedups using GPGPU are of less magnitude but still evident, if implemented correctly. Moreover, algorithm design for such problems on the GPGPU paradigm requires dexterity, since it can be regarded as a task of trying to judiciously detect the equilibrium point between overheads and benefits.

This thesis studied the implementation of more advanced type of joins and set operators on the GPGPU paradigm. Throughout the course of this thesis, the main rationale in the development of the proposed techniques was to take advantage of the different and complementary characteristics offered by CPUs and GPUs.

Starting off, Chapter 3 dealt with theta joins and provided two main techniques along with several optimizations to further improve performance. The best performing technique adopts a sliding window approach, and by resource reuse and through memory hierarchy exploitation, which involves the use of the fastest on chip-memories, manages to achieve speedups of an order of magnitude for the aggregation query and of 2.7X for the select query over other CPU alternatives. Regarding the latter, there are two open issues, (i) transfer times over the slow PCI-E that dominate the total runtime, and (ii) result output writing. For the former, the use of faster interconnects such as NVLink 2.0 would result to the drop of transfer times by an order of magnitude, as demonstrated in [60]. For the latter issue, a possible solution to alleviate the serialization of running threads is by using the shared memory and write the results to global memory in batches. Nevertheless, this is left as future work.

Chapters 4 and 5 investigated the set similarity join problem. In Chapter 4, a co-processing CPU-GPU technique is proposed with the goal to utilize both processors and driven by the fact that every existing technique in literature assigns the complete workload only to one of the two processors. As a result, due to the execution overlap between the CPU and GPU tasks, significant speedups, which can reach up to 2.6X, are achieved over other alternatives in several cases.

In Chapter 5, an empirical evaluation of the state-of-the-art GPU-enabled set similarity techniques is presented, along with several key insights into under which circumstances each technique performs better. The main outcome is that there is no dominant winner; each technique, even not using the GPU at all, has its own sweet spot based on specific dataset characteristics such as the average set size and the number of distinct set elements. These findings were the springboard to the development of the proposed hybrid framework, which encapsulates the state-of-the-art set similarity join techniques. By introducing two workload allocation strategies, the hybrid frame-

work manages to utilize both the CPU and the GPU with high efficiency, while still achieving speedups of up to 3.25X over standalone techniques and of an order of magnitude over other parallel CPU solutions.

Overall, there is still room for further improvements for the set similarity join problem in the GPGPU context. The most prominent is the development of more robust GPU-oriented filters, since (i) the heterogeneity of the prefix filter hinders the GPU's parallelism, especially for datasets with large set sizes, and (ii) bitmap performs well under certain conditions. Another improvement is the extension of the hybrid framework with an automated way to split the workload, alongside a cost model to select the appropriate CPU and GPU technique per scenario. This entails further research on a wider range of combinations of dataset characteristics such as dataset size, set element frequency distributions, number of different set elements and average set sizes. Last, to handle even larger datasets the use the proposed hybrid framework on a multi-GPU setup should be investigated. All of the above are left as future work.

Chapter 6 dealt with set intersection and set containment operators on large sets by adapting techniques initially proposed for graph analytics and matrix multiplication on the GPU, while also investigating new hybrids for completeness. The comprehensive evaluation highlights the main characteristics of the techniques examined when both a single pair of two large sets are processed and all pairs in a dataset are examined, while it provides strong evidence that state-of-the-art set containment stands to significantly benefit from advances in GPU-enabled set intersection. The evaluation results reveal that there is no dominant solution but depending on the exact problem and the dataset characteristics, different techniques are the most efficient ones.

Recently, the work in [87], without investigating set similarity joins explicitly, argues that advanced data analytics should run on CPU only unless the initial data is already stored in GPU's global memory. The work of this thesis provides counter-evidence regarding this. Despite any overheads incurred by the CPU-GPU interconnects, careful crafting of CPU-GPU co-processing schemes for advanced data analytics may lead to tangible speedups as shown in the experiments of this thesis, even when the data initially resides on the CPU side. The bottom line is that data management tasks may differ widely in their nature; thus, while for several cases the claims of [87] hold, each problem must be investigated thoroughly and without prejudice.

As previously stated, hardware advances regarding faster interconnects such as NVLink 2.0 would allow vertical scaling, which in turn may have a big positive impact on the efficiency of hybrid techniques [54, 60]. Furthermore, horizontal scaling in a distributed setting, to process data of volumes that cannot be handled by traditional approaches, is also possible, especially with Spark 3.0 which is GPU-aware and provides convenient abstractions. However, trade-offs between easiness of programming and efficiency exist. Nevertheless, achieving benefits does not come easily, careful crafting, especially w.r.t. memory management and maintaining a high level of parallelism at warp level, is still required. Finally, the work presented in this thesis was based on a single GPU setup; thus, an interesting direction of research would be the investigation of the proposed techniques in a multi-GPU environment.

Appendix A
Published Scientific Papers

Accepted Papers

Peer Reviewed Journals

1. Christos Bellas and Anastasios Gounaris: GPU processing of theta-joins. *Concurrency and Computation: Practice and Experience* Volume 29 (18) (2017)

2. Christos Bellas and Anastasios Gounaris: An empirical evaluation of exact set similarity join techniques using GPUs. *Information Systems* Volume 89 (101485) (2020)

3. Christos Bellas and Anastasios Gounaris: HySet: A hybrid framework for exact set similarity join using a GPU. *Parallel Computing* Volumes 104-105 (102790) (2021)

4. Christos Bellas and Anastasios Gounaris: Exploiting GPUs for fast intersection of large sets. *Information Systems* (to appear)

International Peer Reviewed Conferences

1. Christos Bellas and Anastasios Gounaris: Exact Set Similarity Joins for Large Datasets in the GPGPU paradigm. *DaMoN* 2019: 5:1–5:10

2. Christos Bellas and Anastasios Gounaris: An Evaluation of Large Set Intersection Techniques on GPUs. *DOLAP* 2021: 111–115

Appendix B
Additional Experiments

BMS					
Technique	10K	25K	50K	75K	100K
sf-gssjoin	0.47	1.2	3.13	6.18	10.51
BI-dynamic	0.88	2.01	7.9	17.73	31.38
OBS	1.35	4.39	17.02	39.91	71.97
HI	0.99	2.72	10.56	23.85	41.00
MM	**0.22**	**0.75**	**1.84**	**4.35**	**6.96**
SIMD	1.56	8.11	27.39	72.52	97.85
std	2.39	11.11	37.45	57.44	109.49
boost	**0.57**	**2.67**	**8.79**	**17.43**	**28.17**
Speedup	2.6	3.56	4.75	4.00	4.04

ENRON					
Technique	10K	25K	50K	75K	100K
sf-gssjoin	**0.86**	**2.35**	**7.09**	**12.50**	**20.36**
BI-dynamic	2.21	6.59	20.41	39.33	64.92
OBS	3.32	12.54	41.80	85.50	141.59
HI	3.68	10.75	30.67	58.43	92.59
MM	-	-	-	-	-
SIMD	17.64	73.09	196.85	363.75	541.25
std	38.70	148.80	419.78	736.87	1091.96
boost	**11.49**	**47.50**	137.90	254.76	391.88
Speedup	13.22	20.16	19.43	20.37	19.24

ORKUT					
Technique	10K	25K	50K	75K	100K
sf-gssjoin	**0.59**	**2.21**	**7.07**	**14.85**	**24.06**
BI-dynamic	4.41	26.38	109.47	245.39	435.30
OBS	3.49	19.36	86.63	192.77	337.00
HI	5.89	36.01	138.74	294.20	473.64
MM	-	-	-	-	-
SIMD	28.58	188.18	679.13	1352.21	2408.17
std	67.66	418.08	1614.08	3270.73	5601.96
boost	**16.44**	**105.14**	**446.96**	**931.66**	**1611.23**
Speedup	27.69	47.51	63.14	62.71	66.95

TWITTER					
Technique	10K	25K	50K	75K	100K
sf-gssjoin	**0.99**	**2.20**	**5.98**	**12.48**	**22.25**
BI-dynamic	1.01	2.20	8.16	18.88	34.95
OBS	1.54	4.88	20.10	47.13	88.14
HI	1.25	3.98	15.95	36.48	64.52
MM	2.73	8.53	24.31	51.01	84.08
SIMD	4.55	28.30	112.91	246.90	435.47
std	10.41	62.90	247.86	550.69	982.36
boost	**1.94**	**11.48**	**44.47**	**99.46**	**174.23**
Speedup	1.95	5.21	7.43	7.96	7.83

WORDS					
Technique	10K	25K	50K	75K	100K
sf-gssjoin	13.85	54.26	189.54	371.13	620.93
BI-dynamic	3.08	10.46	39.78	79.45	146.92
OBS	11.48	65.40	235.98	488.92	804.00
HI	13.41	58.97	179.70	352.14	567.65
MM	**1.38**	**4.75**	**13.22**	**26.79**	**43.36**
SIMD	64.19	332.04	1135.02	2287.81	3638.85
std	**34.92**	**180.15**	**617.83**	**1263.55**	**2124.46**
boost	44.54	224.63	744.71	1397.57	2461.01
Speedup	25.23	37.91	46.71	47.115	48.99

Table Appendix B.1: MISI runtimes in seconds with the respective speedups.

	BMS				
Technique	10K	25K	50K	75K	100K
sf-gssjoin	0.48	1.21	3.14	6.19	10.51
MM	**0.25**	**0.83**	**1.85**	**4.48**	**6.96**
MMJoin-CPU	**0.42**	2.34	12.44	11.23	12.62
MMJoin-GPU	0.53	**1.59**	**5.25**	**6.57**	**7.03**
Speedup	1.7	1.91	2.83	1.47	1.01

	ENRON				
Technique	10K	25K	50K	75K	100K
sf-gssjoin	**0.87**	**2.36**	**7.16**	**12.53**	**20.38**
MM	-	-	-	-	-
MMJoin-CPU	**2.00**	12.86	28.92	63.43	78.9
MMJoin-GPU	2.22	**4.59**	**9.66**	**23.97**	**37.11**
Speedup	2.29	1.94	1.35	1.91	1.82

	ORKUT				
Technique	10K	25K	50K	75K	100K
sf-gssjoin	**0.59**	**2.21**	**7.08**	**14.86**	**24.07**
MM	-	-	-	-	-
MMJoin-CPU	**2.27**	**6.30**	**26.62**	**43.75**	**82.20**
MMJoin-GPU	2.43	6.49	34.15	44.71	93.32
Speedup	3.78	2.84	3.75	2.94	3.41

	TWITTER				
Technique	10K	25K	50K	75K	100K
sf-gssjoin	**0.99**	**2.20**	**5.98**	**12.48**	**22.25**
MM	2.77	8.59	24.40	51.14	84.02
MMJoin-CPU	**0.52**	3.02	16.22	45.09	49.57
MMJoin-GPU	0.62	**1.73**	**5.66**	**11.77**	**38.28**
Speedup	0.52	0.78	0.94	0.94	1.72

	WORDS				
Technique	10K	25K	50K	75K	100K
sf-gssjoin	13.86	54.26	189.57	371.14	621.20
MM	**1.38**	**4.82**	**13.31**	**26.89**	**43.48**
MMJoin-CPU	4.73	13.74	50.91	118.92	198.88
MMJoin-GPU	**3.92**	**9.28**	**14.37**	**29.76**	**34.74**
Speedup	2.82	1.92	1.07	1.10	0.79

Table Appendix B.2: MISC runtimes in seconds with the respective speedups.

Bibliography

[1] AMD. 2021. URL: https://www.amd.com/.

[2] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. "Efficient parallel lists intersection and index compression algorithms using graphics processing units". In: *Proceedings of the VLDB Endowment* 4.8 (2011), pp. 470–481.

[3] N. Augsten and M. H. Böhlen. "Similarity joins in relational database systems". In: *Synthesis Lectures on Data Management* 5.5 (2013), pp. 1–124.

[4] P. Bakkum and S. Chakradhar. "Efficient data management for GPU databases". In: *High Performance Computing on Graphics Processing Units* (2012).

[5] R. Baraglia, G. D. F. Morales, and C. Lucchese. "Document Similarity Self-Join with MapReduce". In: *ICDM*. 2010, pp. 731–736.

[6] J. Barbay, A. López-Ortiz, and T. Lu. "Faster adaptive set intersections for text searching". In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2006, pp. 146–157.

[7] R. J. Bayardo, Y. Ma, and R. Srikant. "Scaling up all pairs similarity search". In: *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*. 2007, pp. 131–140.

[8] P. Beame, P. Koutris, and D. Suciu. "Skew in parallel query processing". In: *PODS*. 2014, pp. 212–223.

[9] M. Bisson and M. Fatica. "High performance exact triangle counting on gpus". In: *IEEE Transactions on Parallel and Distributed Systems* 28.12 (2017), pp. 3501–3510.

[10] C. Böhm, R. Noll, C. Plant, and A. Zherdin. "Index-supported Similarity Join on Graphics Processors." In: *BTW*. Vol. 144. 2009, pp. 57–66.

[11] S. Borkar and A. A. Chien. "The future of microprocessors". In: *Communications of the ACM* 54.5 (2011), pp. 67–77.

[12] P. Bouros, S. Ge, and N. Mamoulis. "Spatio-textual similarity joins". In: *PVLDB* 6.1 (2012), pp. 1–12.

[13] P. Bouros, N. Mamoulis, S. Ge, and M. Terrovitis. "Set containment join revisited". In: *Knowl. Inf. Syst.* 49.1 (2016), pp. 375–402.

[14] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra. "Graphics processing unit (GPU) programming strategies and trends in GPU computing". In: *Journal of Parallel and Distributed Computing* 73.1 (2013), pp. 4–13.

[15] S. Chaudhuri, V. Ganti, and R. Kaushik. "A primitive operator for similarity joins in data cleaning". In: *Proc. ICDE, 2006*.

[16] J. Cheng, M. Grossman, and T. McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[17] S. Chu, M. Balazinska, and D. Suciu. "From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System". In: *ACM SIGMOD International Conference on Management of Data*. 2015, pp. 63–78.

[18] M. S. Cruz, Y. Kozawa, T. Amagasa, and H. Kitagawa. "GPU acceleration of set similarity joins". In: *International Conference on Database and Expert Systems Applications*. Springer. 2015, pp. 384–398.

[19] *cuBLAS*. https://developer.nvidia.com/cublas.

[20] S. Deep, X. Hu, and P. Koutris. "Fast Join Project Query Evaluation using Matrix Multiplication". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo. ACM, 2020, pp. 1213–1223.

[21] E. D. Demaine, A. López-Ortiz, and J. I. Munro. "Experiments on adaptive set intersections for text retrieval systems". In: *Workshop on Algorithm Engineering and Experimentation*. Springer. 2001, pp. 91–104.

[22] D. Deng, G. Li, H. Wen, and J. Feng. "An efficient partition based method for exact set similarity joins". In: *Proceedings of the VLDB Endowment* 9.4 (2015), pp. 360–371.

[23] D. Deng, Y. Tao, and G. Li. "Overlap set similarity joins with theoretical guarantees". In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 905–920.

[24] D. Deng, C. Yang, S. Shang, F. Zhu, L. Liu, and L. Shao. "LCJoin: Set Containment Join via List Crosscutting". In: *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 2019, pp. 362–373.

[25] S. Ding, J. He, H. Yan, and T. Suel. "Using graphics processors for high performance IR query processing". In: *Proceedings of the 18th international conference on World wide web*. 2009, pp. 421–430.

[26] *eigen*. https://eigen.tuxfamily.org/.

[27] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. "Scalable and Adaptive Online Joins". In: *PVLDB* 7.6 (2014), pp. 441–452.

[28] F. Fier, N. Augsten, P. Bouros, U. Leser, and J.-C. Freytag. "Set similarity joins on MapReduce: an experimental survey". In: *Proceedings of the VLDB Endowment* 11.10 (2018), pp. 1110–1122.

[29] F. Fier, T. Wang, E. Zhu, and J.-C. Freytag. "Parallelizing Filter-Verification Based Exact Set Similarity Joins on Multicores". In: *International Conference on Similarity Search and Applications*. Springer. 2020, pp. 62–75.

[30] J. Fox, O. Green, K. Gabert, X. An, and D. A. Bader. "Fast and adaptive list intersections on the gpu". In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE. 2018, pp. 1–7.

[31] A. Go, R. Bhayani, and L. Huang. "Twitter sentiment classification using distant supervision". In: *CS224N Project Report, Stanford* 1.12 (2009).

[32] O. Green, R. McColl, and D. A. Bader. "GPU merge path: a GPU merging algorithm". In: *Proceedings of the 26th ACM international conference on Supercomputing*. ACM. 2012, pp. 331–340.

[33] O. Green, P. Yalamanchili, and L.-M. Munguıéa. "Fast triangle counting on the GPU". In: *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press. 2014, pp. 1–8.

[34] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques, 3rd edition*. Morgan Kaufmann, 2011.

[35] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. "Relational joins on graphics processors". In: *ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 511–524.

[36] J. He, M. Lu, and B. He. "Revisiting Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture". In: *PVLDB* 6.10 (2013), pp. 889–900.

[37] S. Helmer and G. Moerkotte. "Evaluation of Main Memory Join Algorithms for Joins with Set Comparison Join Predicates". In: *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*. Ed. by M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld. Morgan Kaufmann, 1997, pp. 386–395.

[38] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear. *The art of multiprocessor programming*. Newnes, 2020.

[39] L. Hu, L. Zou, and Y. Liu. "Accelerating Triangle Counting on GPU". In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 736–748.

[40] Y. Hu, H. Liu, and H. H. Huang. "Tricore: Parallel triangle counting on gpus". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 171–182.

[41] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. "Querying k-truss community in large and dynamic graphs". In: *Proceedings of the 2014 ACM SIGMOD Int. Conf. on Management of data*. 2014, pp. 1311–1322.

[42] Z. Huang, N. Ma, S. Wang, and Y. Peng. "GPU computing performance analysis on matrix multiplication". In: *The Journal of Engineering* 2019.23 (2019), pp. 9043–9048.

[43] R. Jampani and V. Pudi. "Using Prefix-Trees for Efficiently Computing Set Joins". In: *Database Systems for Advanced Applications, 10th International Conference, DASFAA 2005, Beijing, China, April 17-20, 2005, Proceedings*. Ed. by L. Zhou, B. C. Ooi, and X. Meng. Vol. 3453. Lecture Notes in Computer Science. Springer, 2005, pp. 761–772.

[44] J. Ji, J. Li, S. Yan, Q. Tian, and B. Zhang. "Min-max hash for jaccard similarity". In: *2013 IEEE 13th International Conference on Data Mining*. IEEE. 2013, pp. 301–309.

[45] Y. Jiang, G. Li, J. Feng, and W. Li. "String Similarity Joins: An Experimental Evaluation". In: *PVLDB* 7.8 (2014), pp. 625–636.

[46] J. Johnson, M. Douze, and H. Jégou. "Billion-Scale Similarity Search with GPUs". In: *IEEE Trans. Big Data* 7.3 (2021), pp. 535–547. DOI: 10.1109/TBDATA.2019.2921572. URL: https://doi.org/10.1109/TBDATA.2019.2921572.

[47] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. "GPU join processing revisited". In: *Eighth Int. Workshop on Data Management on New Hardware*. ACM. 2012, pp. 55–62.

[48] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and P. Kalnis. "Lightning Fast and Space Efficient Inequality Joins". In: *PVLDB* 8.13 (2015), pp. 2074–2085.

[49] D. Kirk and W. H. Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[50] I. Koumarelas, A. Naskos, and A. Gounaris. "Binary Theta-Joins using MapReduce: Efficiency Analysis and Improvements". In: *Int. Workshop on Algorithms for MapReduce and Beyond (BMR) (in conjunction with EDBT/ICDT'2014)*. 2014.

[51] A. Kunkel, A. Rheinländer, C. Schiefer, S. Helmer, P. Bouros, and U. Leser. "PIEJoin: Towards Parallel Set Containment Joins". In: *Proceedings of the 28th International Conference on Scientific and Statistical Database Management, SSDBM 2016, Budapest, Hungary, July 18-20, 2016*. Ed. by P. Baumann, I. Manolescu-Goujot, L. Trani, Y. E. Ioannidis, G. G. Barnaföldi, L. Dobos, and E. Bányai. ACM, 2016, 11:1–11:12.

[52] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, et al. "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU". In: *Proceedings of the 37th annual international symposium on Computer architecture*. 2010, pp. 451–460.

[53] D. Lemire, L. Boytsov, and N. Kurz. "SIMD compression and the intersection of sorted integers". In: *Software: Practice and Experience* 46.6 (2016), pp. 723–749.

[54] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker. "Evaluating modern GPU interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect". In: *IEEE Transactions on Parallel and Distributed Systems* 31.1 (2019), pp. 94–110.

[55] P. Li, A. Owen, and C.-H. Zhang. "One permutation hashing". In: *Advances in Neural Information Processing Systems*. 2012, pp. 3113–3121.

[56] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. "A fast similarity join algorithm using graphics processing units". In: *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*. IEEE. 2008, pp. 1111–1120.

[57] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010.

[58] J. Luo, W. Zhang, S. Shi, H. Gao, J. Li, W. Wu, and S. Jiang. "FreshJoin: An Efficient and Adaptive Algorithm for Set Containment Join". In: *Data Sci. Eng.* 4.4 (2019), pp. 293–308.

[59] Y. Luo, G. H. L. Fletcher, J. Hidders, and P. D. Bra. "Efficient and scalable trie-based algorithms for computing set containment relations". In: *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. Ed. by J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman. IEEE Computer Society, 2015, pp. 303–314. DOI: 10.1109/ICDE.2015.7113293. URL: https://doi.org/10.1109/ICDE.2015.7113293.

[60] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. "Pump up the volume: Processing large data on gpus with fast interconnects". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 1633–1649.

[61] W. Mann and N. Augsten. "PEL: Position-Enhanced Length Filter for Set Similarity Joins". In: *Proceedings of the 26th GI-Workshop Grundlagen von Datenbanken*. 2014, pp. 89–94.

[62] W. Mann, N. Augsten, and P. Bouros. "An Empirical Evaluation of Set Similarity Join Techniques". In: *Proceedings of the VLDB Endowment* 9.9 (2016), pp. 636–647.

[63] S. Melnik and H. Garcia-Molina. "Adaptive algorithms for set containment joins". In: *ACM Trans. Database Syst.* 28 (2003), pp. 56–99.

[64] A. Metwally and C. Faloutsos. "V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors". In: *PVLDB* 5.8 (2012), pp. 704–715.

[65] P. Mishra and M. H. Eich. "Join processing in relational databases". In: *ACM Computing Surveys (CSUR)* 24.1 (1992), pp. 63–113.

[66] S. Mittal and J. S. Vetter. "A survey of CPU-GPU heterogeneous computing techniques". In: *ACM Computing Surveys (CSUR)* 47.4 (2015), p. 69.

[67] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, Á. L. Garcıéa, I. Heredia, P. Malıék, and L. Hluchỳ. "Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey". In: *Artificial Intelligence Review* 52.1 (2019), pp. 77–124.

[68] NVIDIA. *CUDA C++ Programming Guide*. 2021. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[69] NVIDIA. *CUDA Toolkit Documentation*. http://docs.nvidia.com/cuda.

[70] A. Okcan and M. Riedewald. "Processing theta-joins using MapReduce". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM. 2011, pp. 949–960.

[71] OpenCL. 2021. URL: https://opencl.org/.

[72] J. Pan and D. Manocha. "Fast GPU-based locality sensitive hashing for k-nearest neighbor computation". In: *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems*. ACM. 2011, pp. 211–220.

[73] S. Pandey, X. S. Li, A. Buluc, J. Xu, and H. Liu. "H-INDEX: Hash-Indexing for Parallel Triangle Counting on GPUs". In: *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2019, pp. 1–7.

[74] M. Pietron, P. Russek, and K. Wiatr. "Accelerating Select where and Select Join Queries on a GPU". In: *Computer Science (AGH)* 14.2 (2013), pp. 243–252.

[75] H. Pirk, S. Manegold, and M. Kersten. "Waste not… Efficient co-processing of relational data". In: *30th Int. Conf. on Data Engineering (ICDE)*. IEEE. 2014, pp. 508–519.

[76] A. Polak. "Counting triangles in large graphs on GPU". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2016, pp. 740–746.

[77] R. D. Quirino, S. Ribeiro-Junior, L. A. Ribeiro, and W. S. Martins. "Efficient Filter-Based Algorithms for Exact Set Similarity Join on GPUs". In: *International Conference on Enterprise Information Systems*. Springer. 2017, pp. 74–95.

[78] V. Raman, L. Qiao, W. Han, I. Narang, Y.-L. Chen, K.-H. Yang, and F.-L. Ling. "Lazy, adaptive rid-list intersection, and its application to index anding". In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 2007, pp. 773–784.

[79] L. A. Ribeiro and T. Härder. "Prefix filtering to improve set similarity joins". In: *Information Systems* 36.1 (2011), pp. 62–78.

[80] S. Ribeiro-Junior, R. D. Quirino, L. A. Ribeiro, and W. S. Martins. "Fast parallel set similarity joins on many-core architectures". In: *Journal of Information and Data Management* 8.3 (2017), p. 255.

[81] S. Ribeiro-Júnior, R. D. Quirino, L. A. Ribeiro, and W. S. Martins. "gSSJoin: a GPU-based Set Similarity Join Algorithm." In: *SBBD*. 2016, pp. 64–75.

[82] R. Rui, H. Li, and Y. Tu. "Join algorithms on GPUs: A revisit after seven years". In: *2015 IEEE International Conference on Big Data, Big Data*. 2015, pp. 2541–2550.

[83] R. Rui and Y.-C. Tu. "Fast equi-join algorithms on gpus: Design and implementation". In: *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*. ACM. 2017, p. 17.

[84] E. F. Sandes, G. L. Teodoro, and A. C. Melo. "Bitmap filter: Speeding up exact set similarity joins with bitwise operations". In: *Information Systems* 88 (2020), p. 101449.

[85] A. D. Sarma, Y. He, and S. Chaudhuri. "ClusterJoin: A Similarity Joins Framework using Map-Reduce". In: *PVLDB* 7.12 (2014), pp. 1059–1070.

[86] T. Schank and D. Wagner. "Finding, counting and listing all triangles in large graphs, an experimental study". In: *International workshop on experimental and efficient algorithms*. Springer. 2005, pp. 606–609.

[87] A. Shanbhag, S. Madden, and X. Yu. "A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics". In: *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 2020, pp. 1617–1632.

[88] E. A. Sitaridi and K. A. Ross. "Ameliorating memory contention of OLAP operators on GPU processors". In: *Proceedings of the Eighth International Workshop on Data Management on New Hardware*. ACM. 2012, pp. 39–47.

[89] E. A. Sitaridi and K. A. Ross. "Optimizing select conditions on gpus". In: *Ninth Int. Workshop on Data Management on New Hardware*. ACM. 2013, p. 4.

[90] E. Spertus, M. Sahami, and O. Buyukkokten. "Evaluating similarity measures: a large-scale study in the orkut social network". In: *Proc. SIGKDD, 2005*.

[91] Top500. *The Top500 List (June 2021)*. 2021. URL: https://www.top500.org/lists/top500/2021/06/.

[92] R. Vernica, M. J. Carey, and C. Li. "Efficient parallel set-similarity joins using MapReduce". In: *SIGMOD Conference*. 2010, pp. 495–506.

[93] A. Vitorovic, M. Elseidy, and C. Koch. "Load Balancing and Skew Resilience for Parallel Joins". In: *Proc. of ICDE*. 2016.

[94] V. Volkov. "Better performance at lower occupancy". In: *Proceedings of the GPU Technology Conference, GTC*. Vol. 10. 2010, p. 16.

[95] J. Wang, G. Li, and J. Feng. "Can we beat the prefix filtering?: an adaptive framework for similarity join and search". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2012, pp. 85–96.

[96] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. "Concurrent Analytical Query Processing with GPUs". In: *PVLDB* 7.11 (2014), pp. 1011–1022.

[97] L. Wang, Y. Wang, C. Yang, and J. D. Owens. "A comparative study on exact triangle counting algorithms on the gpu". In: *Proceedings of the ACM Workshop on High Performance Graph Processing*. 2016, pp. 1–8.

[98] N. Wang, J. Zhang, K.-L. Tan, and A. K. Tung. "On triangulation-based dense neighborhood graph discovery". In: *Proceedings of the VLDB Endowment* 4.2 (2010), pp. 58–68.

[99] P. Wang, Y. Qi, Y. Zhang, Q. Zhai, C. Wang, J. C. Lui, and X. Guan. "A memory-efficient sketch method for estimating high similarities in streaming sets". In: *ACM SIGKDD*. 2019, pp. 25–33.

[100] X. Wang, L. Qin, X. Lin, Y. Zhang, and L. Chang. "Leveraging set relations in exact set similarity join". In: *Proceedings of the VLDB Endowment* 10.9 (2017), pp. 925–936.

[101] Y. Wang, A. Shrivastava, and J. Ryu. "FLASH: Randomized Algorithms Accelerated over CPU-GPU for Ultra-High Dimensional Similarity Search". In: *arXiv preprint arXiv:1709.01190* (2017).

[102] P. G. D. Ward, Z. He, R. Zhang, and J. Qi. "Real-time continuous intersection joins over large sets of moving objects using graphic processing units". In: *VLDB J.* 23.6 (2014), pp. 965–985.

[103] P. Wieschollek, O. Wang, A. Sorkine-Hornung, and H. P. A. Lensch. "Efficient Large-Scale Approximate Nearest Neighbor Search on the GPU". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[104] *words*. https://archive.ics.edu/ml/datasets/bag+of+words.

[105] D. Wu, F. Zhang, N. Ao, F. Wang, X. Liu, and G. Wang. "A batched gpu algorithm for set intersection". In: *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*. IEEE. 2009, pp. 752–756.

[106] D. Wu, F. Zhang, N. Ao, G. Wang, X. Liu, and J. Liu. "Efficient lists intersection by cpu-gpu cooperative computing". In: *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–8.

[107] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. "Efficient similarity joins for near-duplicate detection". In: *ACM Transactions on Database Systems (TODS)* 36.3 (2011), pp. 1–41.

[108] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. "Efficient similarity joins for near-duplicate detection". In: *ACM Trans. Database Syst.* 36.3 (2011), 15:1–15:41.

[109] J. Yang, W. Zhang, X. Wang, Y. Zhang, and X. Lin. "Distributed Streaming Set Similarity Join". In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE. 2020, pp. 565–576.

[110] J. Yang, W. Zhang, S. Yang, Y. Zhang, and X. Lin. "TT-Join: Efficient Set Containment Join". In: *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 2017, pp. 509–520.

[111] J. Yang, W. Zhang, S. Yang, Y. Zhang, X. Lin, and L. Yuan. "Efficient set containment join". In: *VLDB J.* 27.4 (2018), pp. 471–495.

[112] S. Zhang, J. He, B. He, and M. Lu. "OmniDB: Towards Portable and Efficient Query Processing on Parallel CPU/GPU Architectures". In: *PVLDB* 6.12 (2013), pp. 1374–1377.

[113] X. Zhang, L. Chen, and M. Wang. "Efficient Multi-way Theta-Join Processing Using MapReduce". In: *PVLDB* 5.11 (2012), pp. 1184–1195.

[114] G. Zhou and H. Chen. "Parallel cube computation on modern CPUs and GPUs". In: *The Journal of Supercomputing* 61.3 (2012), pp. 394–417.

[115] J. Zhou, Q. Guo, H. V. Jagadish, L. Krcál, S. Liu, W. Luan, A. K. H. Tung, Y. Yang, and Y. Zheng. "A Generic Inverted Index Framework for Similarity Search on the GPU". In: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. 2018, pp. 893–904.