Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών
Υπολογιστών Ε.Μ.Π.

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

2016-2017

Άσκηση 4

Παράλληλος προγραμματισμός σε επεξεργαστές γραφικών

Βαρηά Χρυσούλα - 03112105
Κουτσανίτη Ειρήνη - 03112135

Ο ζητούμενος κώδικας παρουσιάζεται στο τέλος της αναφοράς.

Σειριακή Υλοποίηση

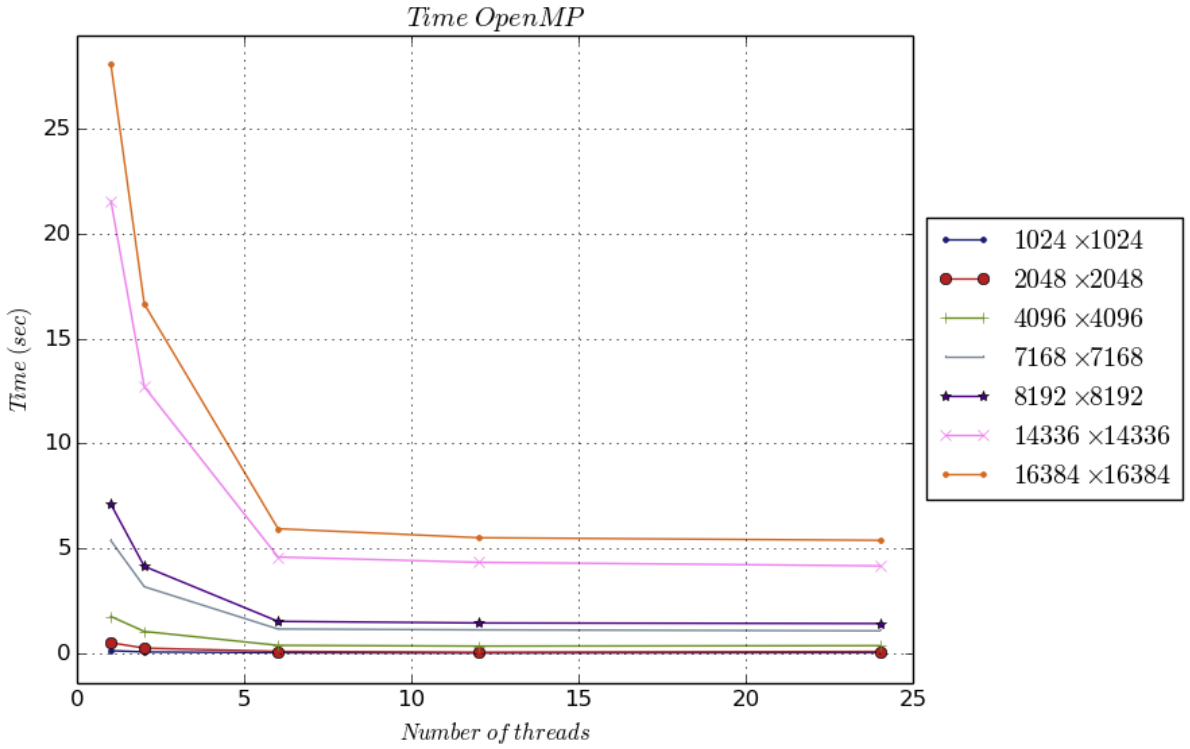
Στον πίνακα 1 παρουσιάζονται οι καλύτεροι χρόνοι εκτέλεσης της σειριακής υλοποίησης (και η αντίστοιχη επίδοσή τους σε Gflops/s), για κάθε μέγεθος πίνακα.

Πίνακας 1: Μετρήσεις σειριακής υλοποίησης

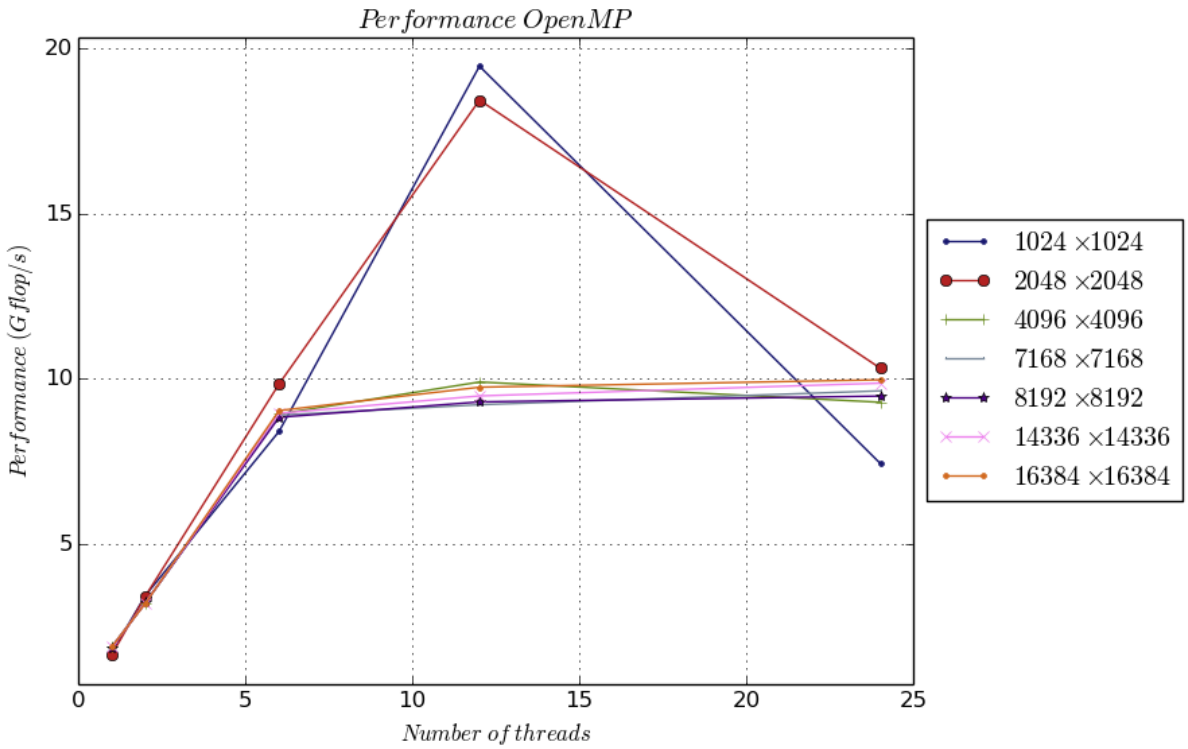
	1K	**2K**	**4K**	**7K**	**8K**	**14K**	**16K**
Time (sec)	0.119	0.496	1.810	5.493	7.103	21.432	27.986
Performance (Gflops/s)	1.743	1.607	1.597	1.777	1.842	1.842	1.876

Υλοποίηση OpenMP

Ακολουθούν οι μετρήσεις της παράλληλης υλοποίησης σε CPU, με χρήση OpenMP. Επίσης παρουσιάζονται τα διαγράμματα χρόνου συναρτήσει του αριθμού των νημάτων της CPU (εικόνα 1), για κάθε μέγεθος πίνακα, και της αντίστοιχης επίδοσης σε Gflops (εικόνα 2).



Εικόνα 1: Χρόνοι της υλοποίησης OpenMp



Εικόνα 2: Επίδοση της υλοποίησης OpenMP

Παρατηρήσεις

- Στο διάγραμμα της επίδοσης σε σχέση με το πλήθος των νημάτων παρατηρείται "κάμψη" στη μετάβαση από τα 6 στα 12 νήματα. Συγκεκριμένα για διαστάσεις πίνακα έως 2K η επίδοση βελτιώνεται μέχρι το πλήθος των νημάτων να γίνει 12 και έπειτα μειώνεται. Για μεγέθη πίνακα μεγαλύτερα από 2K η πτώση της επίδοσης παρατηρείται για λιγότερα νήματα (6 νήματα σύμφωνα με το διάγραμμα). Αυτό μπορεί να οφείλεται σε κάποιο φαινόμενο false sharing, κατά το οποίο διαφορετικά νήματα μεταβάλλουν διαφορετικά δεδομένα, τα οποία όμως μπορεί να βρίσκονται στην ίδια cache line. Ένας άλλος λόγος μεταβολής της επίδοσης είναι να δημιουργείται load imbalance, το οποίο δυσχεραίνει την εκτέλεση του προγράμματος.
- Στο διάγραμμα χρόνου, όπως είναι αναμενόμενο, αυξάνεται ο συνολικός χρόνος εκτέλεσης του προγράμματος με την αύξηση των διαστάσεων του πίνακα. Ωστόσο, όπως επισημαίνεται παραπάνω, βελτίωση της εκτέλεσης του προγράμματος παρουσιάζεται μέχρι το πλήθος των νημάτων να γίνει 6, ενώ έπειτα σταθεροποιείται.

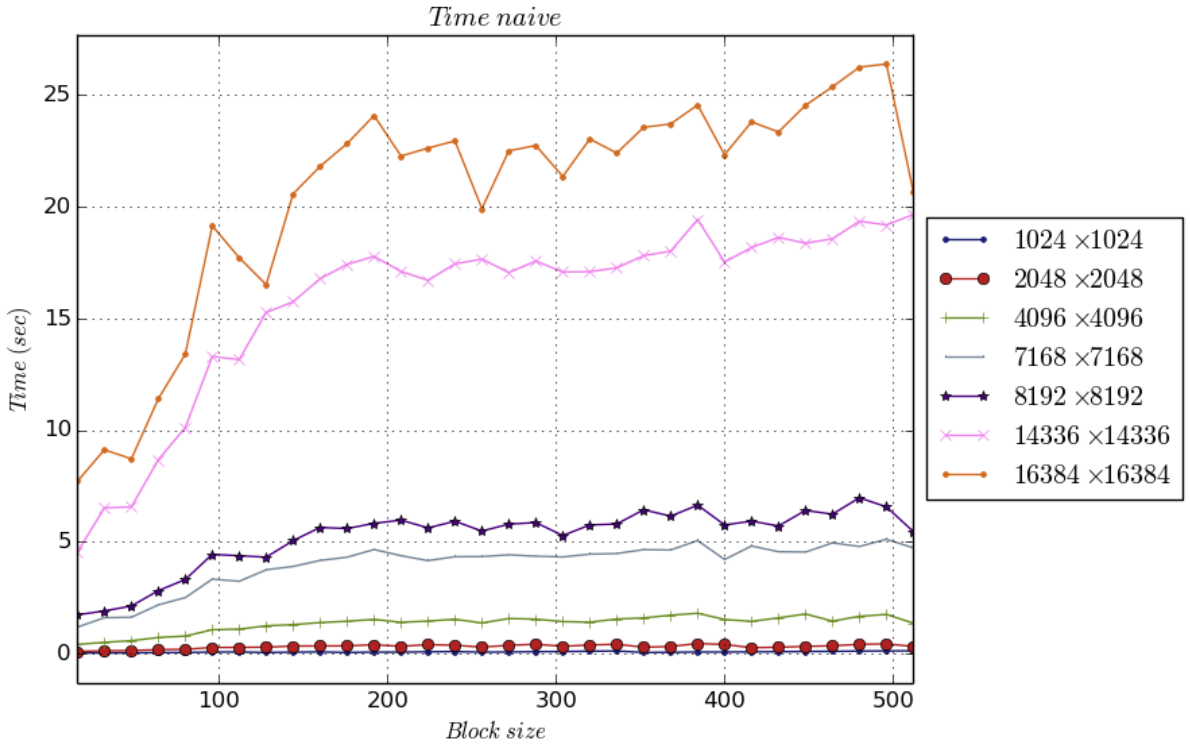
Βασικά Υλοποίηση στη GPU

Ανάλυση κώδικα

Σε όλες τις εκδόσεις υλοποίησης χρησιμοποιείται δισδιάστατο grid και μονοδιάστατο block. Συγκεκριμένα, (αφού υπολογιστεί κατάλληλα το νέο μέγεθος του πίνακα A, ώστε να είναι πολλαπλάσιο του block_size) ορίζεται μέγεθος grid ίσο με $(n/blocksize) \times (n/blocksize)$, ενώ το αντίστοιχο μέγεθος του block ορίζεται ίσο με $blocksize \times 1$. Με αυτόν τον τρόπο κάθε νήμα ενός block υπολογίζει ένα μερικό άθροισμα του γινομένου και το κάθε $y[i]$ υπολογίζεται από grid_size διαφορετικά blocks.

Επισημαίνεται ότι αρχικά στη βασική υλοποίηση χρησιμοποιούνταν μονοδιάστατα grid και block, με αποτέλεσμα κάθε νήμα να υπολογίζει μία γραμμή του πίνακα. Ωστόσο αυτή η υλοποίηση ήταν πιο αργή από την παρούσα καθώς και από την παράλληλη έκδοση OpenMP, επομένως δεν επιλέχθηκε για την εκτέλεση των μετρήσεων. Στον αντίστοιχο κώδικα υπάρχουν σε σχόλια οι υλοποιήσεις με τις μονοδιάστατες παραμέτρους.

Η βασική υλοποίηση επιτυγχάνεται από την dmv_gpu_naive στο αρχείο dmv_gpu.cu και στην εικόνα 3 παρουσιάζονται οι μετρήσεις της εκτέλεσης της.



Εικόνα 3: Χρόνοι της βασικής υλοποίησης σε GPU

Παρατηρήσεις

- Η συγκεκριμένη υλοποίηση παρουσιάζει τον καλύτερο χρόνο για block_size = 16, για όλα τα μεγέθη του πίνακα. Με την γενική αύξηση του blocksize μειώνεται η επίδοση της εκτέλεσης, ενώ σε ορισμένες περιπτώσεις παρουσιάζεται αυξομείωση της. Αίτια αυτού του φαινομένου μπορεί να είναι η μη αποδοτική χρησιμοποίηση των πολυεπεξεργαστικών στοιχείων SMs και η αύξηση του πλήθους των transactions- requests (μεγαλύτερο blocksize προκαλεί αύξηση του πλήθους των requests, τα οποία μπορεί να μην είναι δυνατό να εξυπηρετηθούν από ένα κοινό transaction).

- Με block_size = 496 οι χρόνοι είναι σχεδόν ίδιοι με τους αντίστοιχους της σειριακής υλοποίησης. Ο λόγος για τον οποίο συμβαίνει αυτό είναι ο τρόπος ανάγνωσης δεδομένων από την καθολική μνήμη. Κάθε νήμα του block απαιτεί την ανάγνωση τιμών των A και x, τα οποία είναι αποθηκευμένα στην καθολική μνήμη. Ωστόσο οι τιμές, οι οποίες απαιτούνται από κάθε νήμα σε ένα warp δεν είναι συνεχόμενες στη φυσική μνήμη, με αποτέλεσμα να χρειάζονται περισσότερα transactions ανά segment. Αυτό έχεις ως αποτέλεσμα τη σειριοποίηση των requests κάθε warp νημάτων, το οποίο οδηγεί σε καθυστέρηση και μείωση του throughput εντολών.

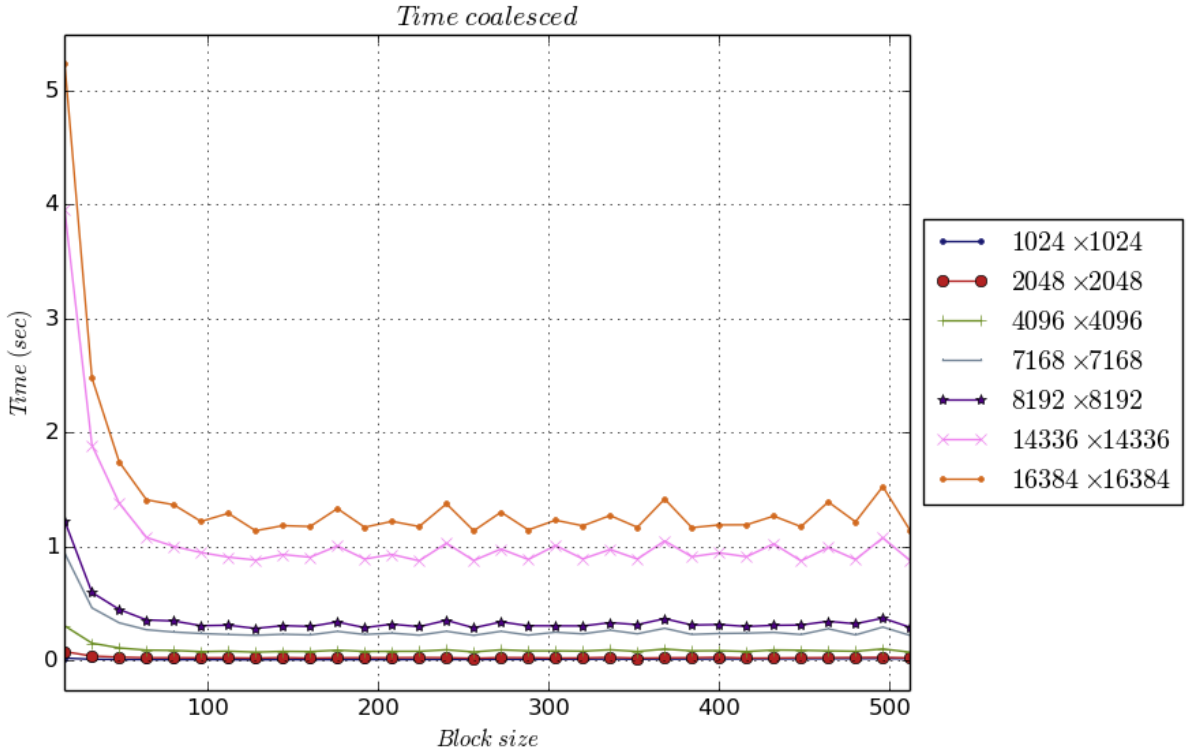
- Τέλος παρατηρείται ότι με την αύξηση του μεγέθους του πίνακα, παρουσιάζεται αντίστοιχη αύξηση του χρόνου εκτέλεσης, καθώς μεγαλώνει ο φόρτος εργασίας κάθε νήματος. Για περιορισμένες διαστάσεις (1K και 2K) αυτή η αύξηση δεν είναι σημαντική, ωστόσο γίνεται εντονότερη κατά τη μετάβαση από μέγεθος πίνακα 8K σε 14K. Με βάση τις παραπάνω παρατηρήσεις, ο κύριος λόγος του συγκεκριμένου φαινομένου είναι η αύξηση της καθυστέρησης της μεταφοράς των δεδομένων από την καθολική μνήμη.

Συνένωση των προσβάσεων στην κύρια μνήμη (memory access coalescing)

Ανάλυση κώδικα

Η ιδέα της συγκεκριμένης υλοποίησης είναι να γίνει εκμετάλλευση των δεδομένων, τα οποία βρίσκονται σε συνεχόμενες θέσεις στην καθολική μνήμη. Αυτό μπορεί να προκαλέσει μείωση του πλήθους των transactions, καθώς περισσότερα απαραίτητα δεδομένα θα μεταφέρονται με ένα transaction. Η υλοποίηση της συγκεκριμένης έκδοσης επιτυγχάνεται από τη dmv_gpu_coalesced στο αρχείο dmv_gpu.cu. Ο πίνακας A είναι αποθηκευμένος κατά γραμμές και γι' αυτό τον λόγο απαιτείται η δημιουργία του ανάστροφου πίνακα A^T, η οποία επιτυγχάνεται στο αρχείο dmv_main.cu με χρήση της συνάρτησης mat_transpose. Με αυτόν τον τρόπο κάθε νήμα απαιτεί στοιχεία μιας στήλης του πίνακα A^T και σε κάθε επανάληψη γειτονικά νήματα απαιτούν γειτονικά στοιχεία, τα οποία βρίσκονται στην ίδια γραμμή του πίνακα A^T (δηλαδή μπορούν να μεταφερθούν με ένα transcaction).

Στην εικόνα 4 παρουσιάζονται οι μετρήσεις αυτής της υλοποίησης.



Εικόνα 4: Χρόνοι της coalesced υλοποίησης σε GPU

Παρατηρήσεις

- Η απόδοση της συγκεκριμένης έκδοσης ως προς τον χρόνο είναι πολύ καλύτερη από αυτή των openMP, serial και naive υλοποιήσεων, το οποίο οφείλεται, όπως αναφέρθηκε προηγουμένως, στην καλύτερη εκμετάλλευση της καθολικής μνήμης και το μειωμένο πλήθος των transactions. Οι χειρότεροι χρόνοι αυτής της υλοποίησης είναι συγκρίσιμοι με τους καλύτερους της βασικής υλοποίησης, για ίδιο μέγεθος block.

- Σε αντίθεση με την προηγούμενη υλοποίηση, τα καλύτερα αποτελέσματα παρουσιάζονται για block_size > 100. Ανάλογα όμως με το μέγεθος του block, φαίνεται να υπάρχουν αυξομοιώσεις στην επίδοση και οι καλύτερες τιμές εμφανίζονται για block_size πολλαπλάσιο του 32. Με αυτόν τον τρόπο αξιοποιείται καλύτερα η μνήμη, καθώς σε κάθε transaction μεταφέρονται όσο το δυνατόν περισσότερα χρήσιμα δεδομένα για κάθε νήμα. Αν δεν είναι πολλαπλάσιο του 32, δεν μπορεί να επιτευχθεί αποδοτική αξιοποίηση των memory requests, υπό την έννοια ότι χρειάζονται περισσότερα transactions για τα τελευταία νήματα του block (τα οποία οργανώνονται σε ένα warp με επιπλέον ανενεργά νήματα). Επίσης ως προς την αξιοποίηση των πολυεπεξεργαστικών στοιχείων (SMs), για καλύτερη απόδοση, θα πρέπει κάθε τέτοια μονάδα της GPU να έχει αρκετά ενεργά warps προκειμένου να ισορροπεί την καθυστέρηση, η οποία προκαλείται από memory και instruction pipeline latency.

- Σχετικά με το compute capability: α) Για compute capability 1.0 απαιτείται το warp να χωρίζεται σε δύο ισομερή τμήματα για την υλοποίηση κάποιου transaction (1 transaction ανά 16 νήματα στην καλύτερη περίπτωση). Κάθε transaction μπορεί να είναι 64 ή 128 bytes, ανάλογα με το μέγεθος των λέξεων, τις οποίες προσπελαύνουν τα νήματα του warp. β) Για compute capability 1.2 και 2.0 μπορεί να γίνει 1 transcaction ανά warp (ιδανικά 1 transaction για όλα τα νήματα του warp).

Χρήση της τοπικής on-chip μνήμης

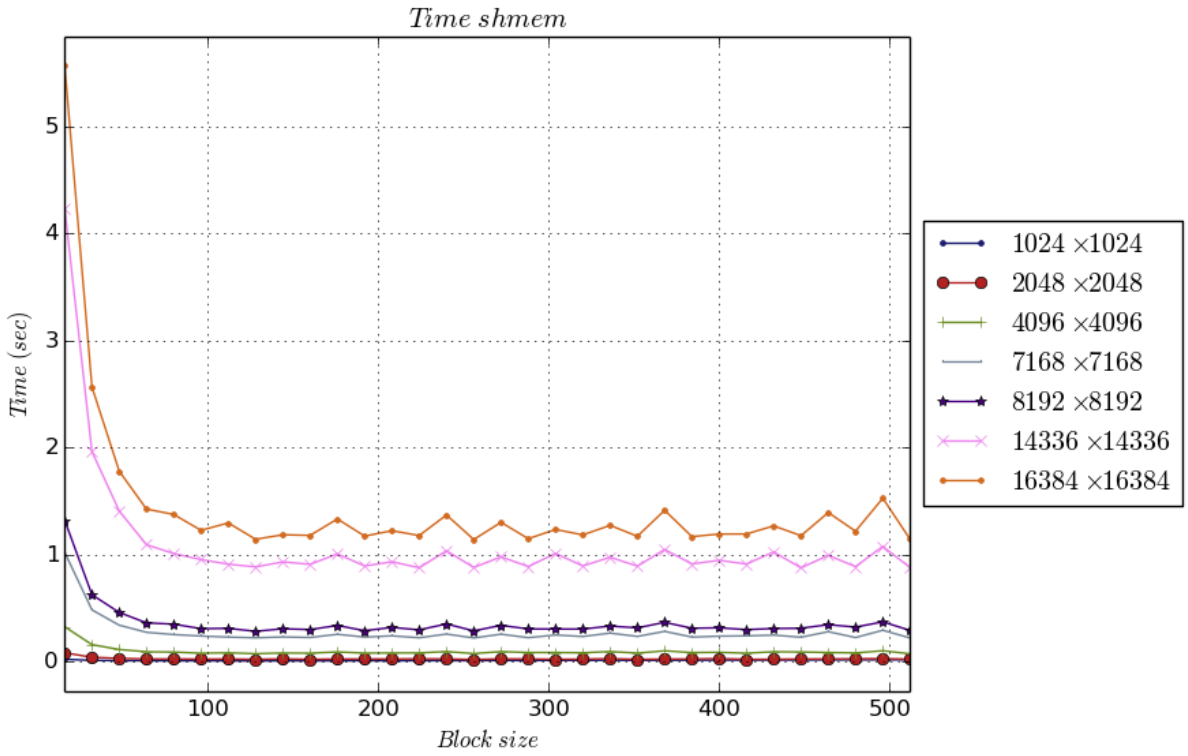
Ανάλυση κώδικα

Η υλοποίηση της συγκεκριμένης έκδοσης επιτυγχάνεται στη dmv_gpu_shmem του αρχείου dmv_gpu.cu. Όπως και στην προηγούμενη υλοποίηση αξιοποιείται η συνένωση των transactions προς την καθολική μνήμη. Η βασική διαφοροποίηση αυτής της υλοποίησης είναι η αξιοποίηση της τοπικής cache για να γίνεται τμηματικά prefetching του διανύσματος x. Η δήλωση της μοιραζόμενης μνήμης γίνεται δυναμικά, το μέγεθος της οποίας καθορίζεται κατά το runtime στάδιο στο αρχείο dmv_main.cu και είναι ίσο με το μέγεθος του block. Κατά την εκτέλεση του κώδικα της GPU, αρχικά γίνεται prefetching ενός τμήματος του διανύσματος x μεγέθους blocksize. Μετά από αυτό το στάδιο απαιτείται συγχρονισμός των νημάτων, ο οποίος επιτυγχάνεται με χρήση της __syncthreads. Ο συγχρονισμός στη συγκεκριμένη περίπτωση εξασφαλίζει ότι έγινε prefetch όλων των απαραίτητων δεδομένων για τους επόμενους υπολογισμούς. Με αυτον τον τρόπο αντιμετωπίζονται πιθανά race conditions. Μόλις επιτευχθεί ο παραπάνω συγχρονισμός και εξυπηρετηθούν τα αντίστοιχα memory requests, κάθε νήμα υπολογίζει ένα επιμέρους πολλαπλασιαστικό όρο, ο οποίος είναι τμήμα του τελικού αποτελέσματος και το προσθέτει στην τιμή του $y[i]$.

Στην εικόνα 5 παρουσιάζονται οι αντίστοιχες μετρήσεις.

Παρατηρήσεις

- Αρχικά παρατηρείται ότι δεν υπάρχει βελτίωση ως προς τον χρόνο σε σχέση με την προηγούμενη υλοποιημένη έκδοση, ενώ η αλλαγή του blocksize επηρεάζει με παρόμοιο τρόπο το performance. Στην αρχιτεκτονική της κάρτας γραφικών Fermi υπάρχει μνήμη on-chip 64KB



Εικόνα 5: Χρόνοι της shmem υλοποίησης σε GPU

(48KB χρησιμοποιούνται ως μοιραζόμενη μνήμη και 16KB ως L1 cache για την μονάδα SM, ή το αντίστροφο), L2 cache (η οποία μοιράζεται μεταξύ 16 SMs μονάδων)και καθολική μνήμη, στην οποία έχουν πρόσβαση όλα τα threads της GPU.

Όταν επιχειρείται η ανάγνωση από την καθολική μνήμη, γίνεται αναζήτηση πρώτα στην L1 cache. Αν τα δεδομένα δεν υπάρχουν στην L1 cache, γίνεται αναζήτηση στην L2 cache και αν δεν βρεθούν εκεί, γίνεται ανάγνωση από την καθολική μνήμη. Τα δεδομένα, τα οποία διαβάζονται από την καθολική μνήμη, μεταφέρονται στις L2, L1 caches για μελλοντική πρόσβαση σε αυτά. Από τα παραπάνω προκύπτει ότι αν το μέγεθος του block είναι μικρό (χωράει στην L1 cache), δεν είναι αναγκαίο να γίνει ρητά prefetching των δεδομένων. Αυτό μπορεί να εξηγεί γιατί η επίδοση της coalesced έκδοσης είναι ίδια ή υψηλότερη από της συγκεκριμένης υλοποίησης, καθώς τα δεδομένα μετά την πρώτη ανάγνωση αποθηκεύονται στην L1 cache. Επίσης στην υλοποίηση με μοιραζόμενη μνήμη, η διαδικασία του prefetching απαιτεί τον συγχρονισμό των νημάτων, το οποίο μπορεί να προκαλεί επιπρόσθετο latency.

- Σε παλιότερες αρχιτεκτονικές ενδεχομένως η L1 cache να ήταν μικρότερη και να ήταν χρήσιμο να την διαχειρίζεται ο προγραμματιστής μέσω της shared memory.
- Όπως αναφέρθηκε προηγουμένως, η χρησιμοποίηση των πολυεπεξεργαστικών στοιχείων SMs είναι υψηλότερη όταν το μέγεθος του block είναι πολλαπλάσιο του 32, καθώς περισσότερα νήματα είναι ενεργά σε κάθε SM. Στα διαγράμματα, τα οποία παρουσιάζονται στην αναφορά, αυτό μπορεί να εκφράζεται ως την εναλλασσόμενη αυξομείωση της επίδοσης σε σχέση με την αλλαγή του blocksize.

Χρήση της βιβλιοθήκης cuBLAS

Ανάλυση κώδικα

Η υλοποίηση της συγκεκριμένης έκδοσης βρίσκεται στο αρχείο dmv_main.cu. Χρησιμοποιείται η συνάρτηση cublasSgmev, η οποία εκτελεί την πράξη y = alpha * op(A) * x + beta * y. Επομένως για τον σωστό υπολογισμό του πολλαπλασιασμού, απαιτείται κατάλληλη αρχικοποίηση των παραμέτρων $alpha = 1.0$ και $beta = 0.0$.

Η παράμετρος trans καθορίζει αν op(A) = A ή op(A) = transpose(A). Επειδή ο πίνακας A αποθηκεύεται κατά στήλες και για να μη γίνεται σε κάθε επανάληψη αντιστροφή του, αυτός αντιστρέφεται μία φορά στην αρχή και έπειτα η συνάρτηση καλείται με κατάλληλο τελεστή CUBLAS_OP_N.

Παρακάτω παρουσιάζονται οι μετρήσεις του χρόνου και της επίδοση την υλοποίησης για κάθε μέγεθος εισόδου (πίνακας 2).

Πίνακας 2: Μετρήσεις cuBLAS υλοποίησης

	1K	**2K**	**4K**	**7K**	**8K**	**14K**	**16K**
Time (sec)	0.007	0.019	0.075	0.233	0.301	0.893	1.237
Performance (Gflops/s)	31.009	43.283	43.695	44.025	44.576	45.935	43.356

Παρατηρήσεις

- Οι χρόνοι εκτέλεσης της έκδοσης cuBLAS είναι παρόμοιοι με αυτούς, οι οποίοι υπολογίστηκαν στις υλοποιήσεις των εκδόσεων coalesced και shmem.

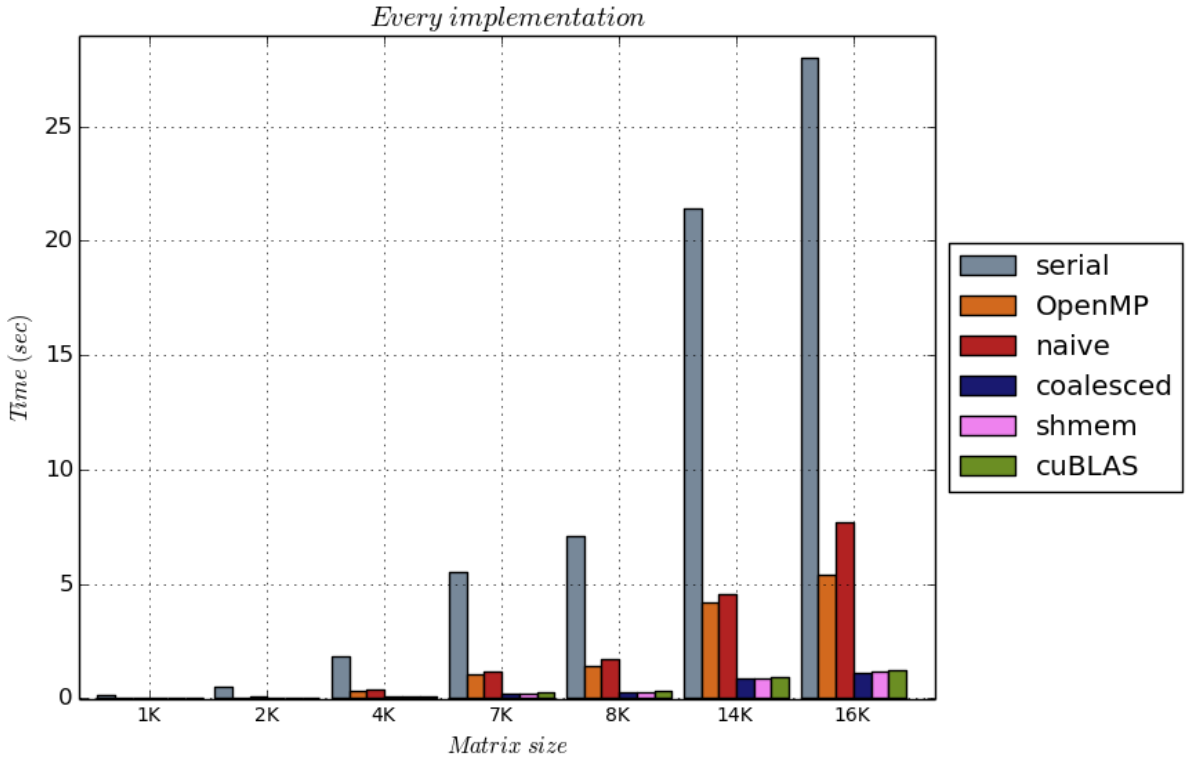
Συνολικά διαγράμματα και γενικές παρατηρήσεις

Παρουσιάζονται οι καλύτερες μετρήσεις χρόνου (εικόνα 6) και επίδοσης (εικόνα **??**) για όλες τις υλοποιήσεις. Λόγω έλλειψης χώρου τα χαρακτηριστικά (αριθμός threads και μέγεθος block_size) των καλύτερων μετρήσεων κάθε υλοποίησης παρουσιάζονται στον πίνακα 3.

Η επίδοση της GPU καθορίζεται άμεσα από το μέγεθος του block, καθώς όπως αναφέρθηκε παραπάνω για καλύτερη απόδοση είναι επιθυμητό κάθε μονάδα SM να περιέχει μεγάλο πλήθος ενεργών threads. Με αυτόν τον τρόπο η GPU "υπερκαλύπτει" την υπάρχουσα καθυστέρηση, εκτελώντας παράλληλα άλλες εργασίες. Μια μετρική, η οποία αξιολογεί τη χρησιμοποίηση κάθε SM είναι το

Πίνακας 3: Καλύτερες επιδόσεις όλων των υλοποιήσεων

	1K	**2k**	**4K**	**7K**	**8K**	**14K**	**16K**
serial	0.119	0.496	1.810	5.493	7.103	21.432	27.986
OpenMP	0.011	0.046	0.339	1.066	1.415	4.163	5.382
#threads	12	12	12	24	24	24	24
naive	0.032	0.103	0.413	1.189	1.738	4.542	7.712
block_size	16	16	16	16	16	16	16
coalesced	0.006	0.019	0.073	0.221	0.281	0.874	1.138
block_size	128	256	512	512	128	448	256
shared memory	0.007	0.019	0.073	0.221	0.283	0.875	1.140
block_size	128	256	512	512	128	448	256
cuBLAS	0.007	0.019	0.075	0.233	0.301	0.893	1.237

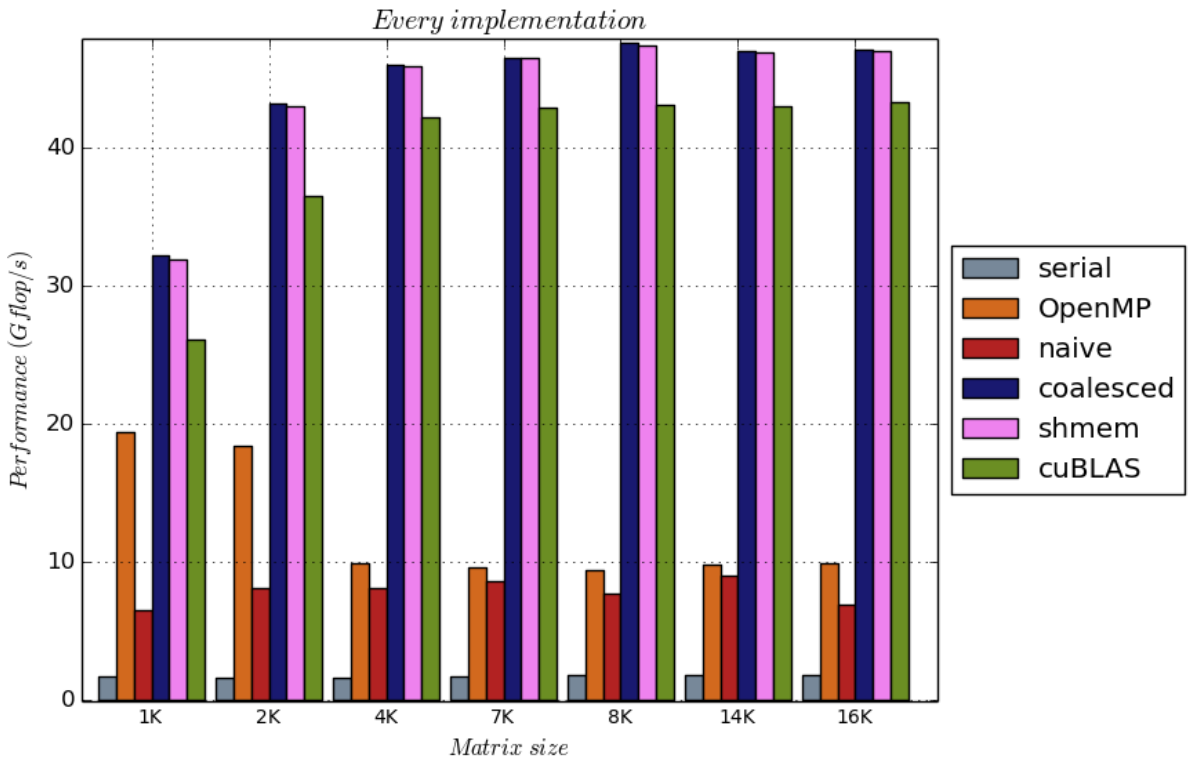


Εικόνα 6: Χρόνοι όλων των υλοποιήσεων

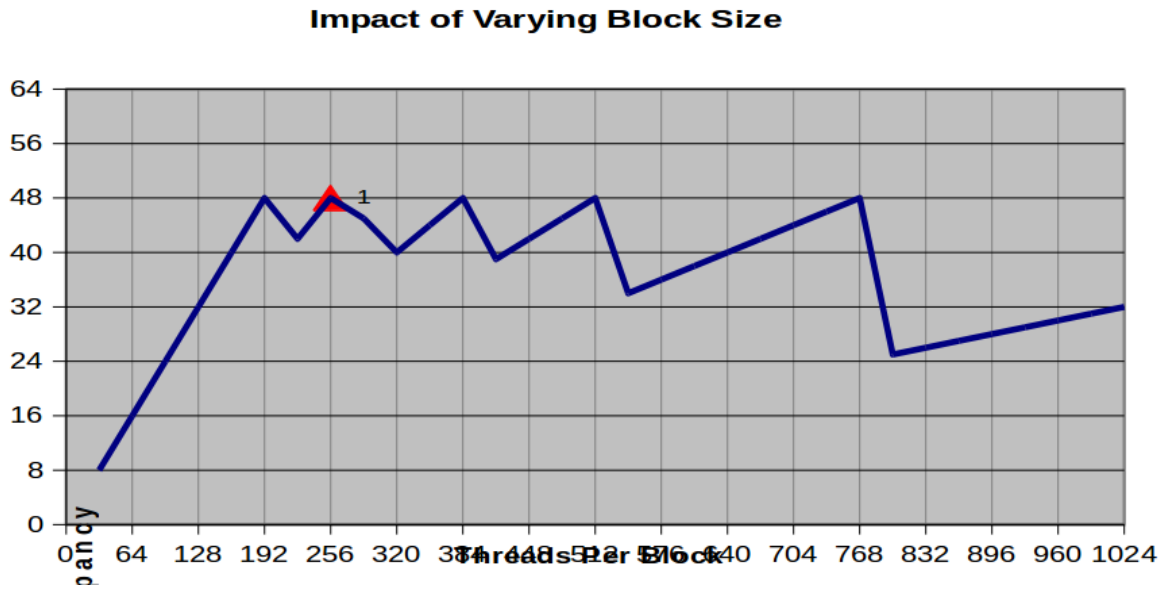
occupancy (δηλαδή το πλήθος των ενεργών warps σε μια πολυεπεξεργαστική μονάδα προς το συνολικό αριθμό warps, τα οποία μπορούν να τρέχουν συγχρόνως). Σε μια SM μονάδα μπορεί να εκτελείται περιορισμένος αριθμός από threads συγχρόνως (maximum SM threads)και προκύπτει ότι σε αυτή χωράνε (maximum SM threads / threads in block) blocks. Αν το μέγεθος του block (ως προς το πλήθος των threads, τα οποία περιέχει) δεν διαιρεί το πλήθος maximum SM threads τότε το πολυεπεξεργαστικό στοιχείο δεν αξιοποιείται πλήρως, καθώς υπάρχουν ανενεργά warps. Σε αυτή την περίπτωση η επίδοση της εκτέλεσης μπορεί να είναι χαμηλότερη.

Ωστόσο η χαμηλή επίδοση μπορεί να μην οφείλεται πάντα στο blocksize, καθώς περιοριστικοί παράγοντες είναι επίσης το πλήθος των καταχωρητών ανά thread και το μέγεθος της μοιραζόμενης μνήμης, όπως αναφέρεται στις διαφάνειες της αντίστοιχης διάλεξης.

Για τις 3 εκδόσεις της GPU (naive, coalesced, shmem) χρησιμοποιούνται 15 ή 16 καταχωρητές ανά νήμα (15 στην naive έκδοση και 16 στις άλλες δύο) και μέχρι 2048 bytes μοιραζόμενης μνήμης (σύμφωνα με τα δεδομένα μεταγλώττισης). Με χρήση του cuda occupancy calculator, από τα διαγράμματα που προκύπτουν παρατηρείται ότι ο αριθμός των ενεργών warps παραμένει σταθερός για τιμές παραμέτρων μικρότερες ή ίσες με τις προαναφερόμενες. Από αυτό βγαίνει το συμπέρασμα ότι η παράμετρος, η οποία φαίνεται να μεταβάλλει τον αριθμό των ενεργών warps είναι το πλήθος των νημάτων ανά μπλοκ (δηλαδή το αντίστοιχο block_size). Οι μεταβολές των ενεργών warps συναρτήσει του μεγέθους του block παρουσιάζονται στο σχήμα ??.



Εικόνα 7: Επιδόσεις όλων των υλοποιήσεων



Εικόνα 8: Τα ενεργά warps ανάλογα με το block_size

Κώδικας

dmv_gpu.cu

```
/*
```

```
 *  dmv_gpu.cu -- Template for DMV GPU kernels
 *
 *  Copyright (C) 2010-2013, Computing Systems Laboratory (CSLab)
 *  Copyright (C) 2010-2013, Vasileios Karakasis
 */
#include <stdio.h>
#include "dmv.h"

/*
 *  Utility function to get the thread ID within the
 *  global working space.
 */
__device__ int get_global_tid()
{
    return (gridDim.x*blockIdx.y + blockIdx.x)*blockDim.x*blockDim.y
        +
        blockDim.x*threadIdx.y + threadIdx.x;
}

/*
 *  Utility function to get the thread ID within the
 *  local/block working space.
 */
__device__ int get_local_tid()
{
    return blockDim.x*threadIdx.y + threadIdx.x;
}

/*
 *  Naive kernel
 */
__global__ void dmv_gpu_naive(const value_t *a, const value_t *x,
   value_t *y,
                              size_t n)
{
    int i, j, a_j, limitj;
    value_t _yi = 0;

    j = blockIdx.x*blockDim.x;
    limitj = j + blockDim.x;
    i = threadIdx.x + blockIdx.y*blockDim.x;
    a_j = i*n + j;

    for (; j < limitj; ++j, ++a_j)
        _yi += a[a_j]*x[j];
    atomicAdd(&y[i], _yi);
}
/*
 *  Naive kernel 1-dim grid
```

```
 */
/*
__global__ void dmv_gpu_naive(const value_t *a, const value_t *x,
   value_t *y,
                              size_t n
{
   int tid = get_global_tid();
   size_t j, a_j;
   register value_t _yi = 0;
   if (tid < n) {
       for (j = 0, a_j=tid*n; j < n; ++j, ++a_j) {
           _yi += a[a_j]*x[j];
       }
       y[tid] = _yi;
   }
}
 */

/*
 *  Coalesced memory acceses
 */
__global__ void dmv_gpu_coalesced(const value_t *a, const value_t *x
   ,
                                  value_t *y, size_t n)
{
    int i, j, a_j, limitj;
    value_t _yi = 0;

    j = blockIdx.x*blockDim.x;
    limitj = j + blockDim.x;
    i = threadIdx.x + blockIdx.y*blockDim.x;
    a_j = j*n + i;

    for (; j < limitj; ++j, a_j += n)
        _yi += a[a_j]*x[j];
    atomicAdd(&y[i], _yi);
}
/*
 *  Coalesced memory acceses 1-dim grid
 */
/*
__global__ void dmv_gpu_coalesced(const value_t *a, const value_t *x
   ,
                                  value_t *y, size_t n)
{
   int tid = get_global_tid();
   size_t j, a_j;
   register value_t _yi = 0;
   if (tid < n) {
```

```
        for (j = 0, a_j=tid; j < n; ++j, a_j += n) {
            _yi += a[a_j]*x[j];
        }
        y[tid] = _yi;
    }
}
 */


/*
 *  Use of shared memory
 */
__global__ void dmv_gpu_shmem(const value_t *a, const value_t *x,
    value_t *y,
                                size_t n)
{
    int i, j, a_j, limitj;
    value_t _yi = 0;
    extern __shared__ value_t sh_x[];

    j = blockIdx.x*blockDim.x;
    limitj = j + blockDim.x;
    i = threadIdx.x + blockIdx.y*blockDim.x;
    a_j = j*n + i;

    sh_x[threadIdx.x] = x[j + threadIdx.x];
    __syncthreads();
    for (; j < limitj; ++j, a_j += n)
        _yi += a[a_j]*x[j];
    atomicAdd(&y[i], _yi);
}
/*
 *  Use of shared memory 1-dim grid
 */
/*
__global__ void dmv_gpu_shmem(const value_t *a, const value_t *x,
    value_t *y,
                                size_t n)
{
    int loc_tid = get_local_tid();
    int tid = get_global_tid();
    size_t i, j, a_j;
    register value_t _yi = 0;
    extern __shared__ value_t sh_x[];
    for (i = 0; i < n; i += blockDim.x) {
        sh_x[loc_tid] = x[i+loc_tid];
        __syncthreads();
        for (j = 0, a_j = tid + i*n; j < blockDim.x; ++j, a_j += n)
            {
```

```
            _yi += a[a_j]*sh_x[j];
        }
        __syncthreads();
    }
    y[tid] = _yi;
}
 */
```

dmv_main.cu

```
/*
 *  dmv_main.cu -- DMV front-end program.
 *
 *  Copyright (C) 2010-2012, Computing Systems Laboratory (CSLab)
 *  Copyright (C) 2010-2012, Vasileios Karakasis
 */

#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>
#include "alloc.h"
#include "dmv.h"
#include "error.h"
#include "gpu_util.h"
#include "timer.h"
#include <cublas_v2.h>

#ifndef VALUES_MAX
#   define VALUES_MAX MAKE_VALUE_CONSTANT(1.)
#endif

#ifndef EPS
#   define EPS MAKE_VALUE_CONSTANT(1.e-6)
#endif

#ifndef NR_ITER
#   define NR_ITER 100
#endif

static void check_result(const value_t *test, const value_t *orig,
   size_t n)
{
    printf("Checking ... ");
    size_t  i_fail = vec_equals(test, orig, n, EPS);
    if (!i_fail) {
        printf("PASSED\n");
    } else {
        printf("FAILED (index: %ld)\n", i_fail - 1);
        printf("%" VALUE_FORMAT " != " "%" VALUE_FORMAT "\n",
               test[i_fail-1], orig[i_fail-1]);
```

```
    }
}

static void report_results(xtimer_t *timer, size_t n)
{
    double  elapsed_time = timer_elapsed_time(timer);
    size_t  flops        = 2*n*n*NR_ITER;

    printf("Elapsed time: %lf s\n", elapsed_time);
    printf("Performance:  %lf Gflop/s\n", flops*1.e-9 / elapsed_time
        );
}

static void print_usage()
{
    printf("Usage: [GPU_KERNEL=<kernel_no>] [GPU_BLOCK_SIZE=<size>] "
            "%s <matrix size>\n", program_name);
    printf("GPU_KERNEL defaults to 0\n");
    printf("GPU_BLOCK_SIZE defaults to 256\n");
    printf("Available kernels [id:name]:\n");
    size_t i;
    for (i = 0; i < GPU_KERNEL_END; ++i) {
        printf("\t%zd:%s\n", i, gpu_kernels[i].name);
    }
}

int main(int argc, char **argv)
{
    set_program_name(argv[0]);
    if (argc < 2) {
        warning(0, "too few arguments");
        print_usage();
        exit(EXIT_FAILURE);
    }

    size_t n = atoi(argv[1]);
    if (!n)
        error(0, "invalid argument: %s", argv[1]);

    /* Read block size and kernel to launch from the environment */
    const char *env_gpu_kernel = getenv("GPU_KERNEL");
    const char *env_gpu_block_size = getenv("GPU_BLOCK_SIZE");
    int kernel = (env_gpu_kernel) ? atoi(env_gpu_kernel) : GPU_NAIVE
        ;
    int block_size = (env_gpu_block_size) ? atoi(env_gpu_block_size)
        : 256;
    size_t orig_n = n;  // original matrix size
```

```
    int grid_size = n % block_size == 0 ? n / block_size : n /
       block_size + 1;  // grid size

    /*
     *  Adjust appropriately (increase
     *  only) the matrix size here if that helps you with your
     *  kernel code, e.g., to avoid divergent warps.
     */
    n = grid_size * block_size;

    printf("Matrix size: %zd\n", orig_n);
    printf("Adjusted matrix size: %zd\n", n);

    /*
     * Allocate the structures.
     *
     * Initialization to zero is crucial if you adjusted the matrix
     * size.
     */
    value_t **A = (value_t **) calloc_2d(n, n, sizeof(**A));
    if (!A)
        error(1, "alloc_2d failed");

    value_t *x = (value_t *) calloc(n, sizeof(*x));
    if (!x)
        error(1, "malloc failed");

    value_t *y_serial = (value_t *) calloc(n, sizeof(*y_serial));
    if (!y_serial)
        error(1, "malloc failed");

    value_t *y = (value_t *) calloc(n, sizeof(*y));
    if (!y)
        error(1, "malloc failed");

    /* Initialize */
    srand48(0);
    mat_init_rand(A, orig_n, VALUES_MAX);
    vec_init_rand(x, orig_n, VALUES_MAX);
    vec_init(y_serial, orig_n, MAKE_VALUE_CONSTANT(0.0));
    vec_init(y, orig_n, MAKE_VALUE_CONSTANT(0.0));

    /* Setup timers */
    xtimer_t timer;

    /* Compute serial */
#ifdef SERIAL_KERNEL
    printf(">>>> Begin of record <<<<\n");
    printf("Serial version:\n");
```

```
    timer_clear(&timer);
    timer_start(&timer);
    for (size_t i = 0; i < NR_ITER; ++i)
        dmv_serial(A, x, y_serial, orig_n);
    timer_stop(&timer);
    report_results(&timer, orig_n);
    printf(">>>> End of record <<<<\n");
#endif  // SERIAL_KERNEL

#ifdef OPENMP_KERNEL
    /* Compute OpenMP */
    printf(">>>> Begin of record <<<<\n");
    printf("OpenMP version:\n");
    timer_clear(&timer);
    timer_start(&timer);
    for (size_t i = 0; i < NR_ITER; ++i)
        dmv_omp(A, x, y, orig_n);
    timer_stop(&timer);
#ifndef _NOCHECK_
    check_result(y, y_serial, orig_n);
#endif
    report_results(&timer, orig_n);
    printf(">>>> End of record <<<<\n");
#endif  // OPENMP_KERNEL

#ifdef GPU_KERNEL
    /*
     *  Set up the blocks, grid and shared memory depending on
     *  the kernel. Make any transformations to the input
     *  matrix here.
     */

    if (kernel > 0)
        mat_transpose(A, n);

    dim3 gpu_block(block_size, 1);   // set up the block dimensions
    dim3 gpu_grid(grid_size, grid_size);    // set up the grid
       dimensions
    size_t shmem_size = 0;  // set up the shared memory size
    if (kernel == 2) {
        shmem_size = block_size * sizeof(value_t);
    }

    /* Cublas initializations */
    value_t alpha=1.0, beta=0;
    cublasHandle_t handle;
    cublasCreate(&handle);

    printf(">>>> Begin of record <<<<\n");
```

```
    printf("Block␣size:␣%dx%d\n", gpu_block.x, gpu_block.y);
    printf("Grid␣size␣:␣%dx%d\n", gpu_grid.x, gpu_grid.y);
    printf("Shared␣memory␣size:␣%ld␣bytes\n", shmem_size);

    /* GPU allocations */
    value_t *gpu_A = (value_t *) gpu_alloc(n*n*sizeof(*gpu_A));
    if (!gpu_A)
        error(0, "gpu_alloc␣failed:␣%s", gpu_get_last_errmsg());

    value_t *gpu_x = (value_t *) gpu_alloc(n*sizeof(*gpu_x));
    if (!gpu_x)
        error(0, "gpu_alloc␣failed:␣%s", gpu_get_last_errmsg());

    value_t *gpu_y = (value_t *) gpu_alloc(n*sizeof(*gpu_y));
    if (!gpu_y)
        error(0, "gpu_alloc␣failed:␣%s", gpu_get_last_errmsg());

    /* Copy data to GPU */
    if (copy_to_gpu(A[0], gpu_A, n*n*sizeof(*gpu_A)) < 0)
        error(0, "copy_to_gpu␣failed:␣%s", gpu_get_last_errmsg());

    if (copy_to_gpu(x, gpu_x, n*sizeof(*gpu_x)) < 0)
        error(0, "copy_to_gpu␣failed:␣%s", gpu_get_last_errmsg());

    /* Reset y and copy it to GPU */
    vec_init(y, n, MAKE_VALUE_CONSTANT(0.0));
    if (copy_to_gpu(y, gpu_y, n*sizeof(*gpu_y)) < 0)
        error(0, "copy_to_gpu␣failed:␣%s", gpu_get_last_errmsg());

    if (kernel >= GPU_KERNEL_END)
        printf("GPU␣kernel␣version:␣cuBLAS\n");
        // error(0, "the requested kernel does not exist");
    else
        printf("GPU␣kernel␣version:␣%s\n", gpu_kernels[kernel].name)
            ;

    /* Execute and time the kernel */
    timer_clear(&timer);
    timer_start(&timer);
    for (size_t i = 0; i < NR_ITER; ++i) {
        if (kernel < GPU_KERNEL_END) {
            gpu_kernels[kernel].fn<<<gpu_grid,gpu_block,shmem_size
                >>>
                (gpu_A, gpu_x, gpu_y, n);
        }
        else {
            cublasSgemv(handle, CUBLAS_OP_N, n, n, &alpha, gpu_A, n,
                gpu_x, 1, &beta, gpu_y, 1);
        }
```

```
#ifdef _DEBUG_
        cudaError_t err;
        if ( (err = cudaGetLastError()) != cudaSuccess)
            error(0, "gpu kernel failed to launch: %s",
               gpu_get_errmsg(err));
#endif
        cudaThreadSynchronize();
    }
    timer_stop(&timer);

    /* Copy result back to host and check */
    if (copy_from_gpu(y, gpu_y, n*sizeof(*y)) < 0)
        error(0, "copy_from_gpu failed: %s", gpu_get_last_errmsg());

    /* Fix resuts */
    if (kernel < 3) {
        for (int i=0; i<n; i++) {
            y[i] /= NR_ITER;
        }
    }


#ifndef _NOCHECK_
    check_result(y, y_serial, orig_n);
#endif
    report_results(&timer, orig_n);
    printf(">>>> End of record <<<<\n");
#endif  // GPU_KERNEL

    /* Free resources on host */
    free_2d((void **) A);
    free(x);
    free(y);
    free(y_serial);

#ifdef GPU_KERNEL
    /* Free resources on GPU */
    gpu_free(gpu_A);
    gpu_free(gpu_x);
    gpu_free(gpu_y);
#endif  // GPU_KERNEL

    return EXIT_SUCCESS;
}
```