ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ροή Υ : Συστήματα Παράλληλης Επεξεργασίας

4η Άσκηση (Τελική Αναφορά)

Παράλληλος Προγραμματισμός σε επεξεργαστές Γραφικών

Κριθαρούλας Διονύσιος 03117875

Ομάδα:Parlab18

Εξάμηνο: 9ο

Εισαγωγή

Στην συγκεκριμένη εργαστηριακή άσκηση υλοποιούμε τον αλγόριθμο πολλαπλασιασμού πίνακα με πίνακα (Dense Matrix-Matrix multiplication , DMM) για επεξεργαστές γραφικών χρησιμοποιώντας το προγραμματιστικό περιβάλλον CUDA. Ο DMM είναι ένας από τους πιο σημαντικούς πυρήνες αλγεβρικών υπολογισμών. Έστω οι πίνακες εισόδου A και B με διαστάσεις MxK και KxN. Ο πυρήνας DMM υπολογίζει τον πίνακα εξόδου C=A·B διαστάσεων MxN. Σε αναλυτική μορφή κάθε στοιχείο του πίνακα εξόδου υπολογίζεται ως:

$$C_{i,j} = \sum_{k=1}^{K} A_{ik} \cdot B_{kj}, \forall i \in [1, M], \forall j \in [1, N]$$

`

2. Ζητούμενα

2.1 Υλοποίηση για επεξεργαστές Γραφικών (GPUs)

Πραγματοποιήθηκαν τρείς διαφορετικές υλοποιήσεις για τον αλγόριθμο DMM. Και οι τρείς υλοποιήσεις λειτουργούν για οποιαδήποτε τιμή των διαστάσεων N,M,K εφόσον τα αντίστοιχα δεδομένα του προβλήματος χωράνε στην κύρια μνήμη της GPU. Η κάθε επόμενη υλοποίηση έχεις σκοπό να βελτιώνει το performance της προηγούμενης. Πριν αναλύσουμε την κάθε υλοποίηση ξεχωριστά παραθέτουμε το κομμάτι του κώδικα το οποίο ορίζει το thread_block και το grid αντίστοιχα:

Το κομμάτι του κώδικα αυτού βρίσκεται στο αρχείο dmm_main.cu

Θυμίζουμε ότι το block size δείχνει το πλήθος των νημάτων που ανήκουν σε ένα thread_block. Αντίστοιχα το grid size δείχνει το μέγεθος των thread_blocks που ανήκουν σε ένα grid. Τόσο το thread_block όσο και το grid αποτελούν δισδιάστατα μεγέθη καθώς ο αλγόριθμος μας υπολογίζει τις τιμές ενός δισδιάστατου πίνακα.

Οι διαστάσεις του thread_block ορίζονται ως εξής:

- Αριθμός γραμμών του thread_block = THEAD_BLOCK_Y (το οποίο ορίζουμε κατά την εκτέλεση)
- Αριθμός στηλών του thread_block = THREAD_BLOCK_X (το οποίο ορίζουμε κατά την εκτέλεση)

Επομένως κάθε thread_block αποτελείται συνολικά από THREAD_BLOCK_Y x THREAD_BLOCK_X νήματα.

Οι διαστάσεις του grid ορίζονται ως εξής:

- Αριθμός γραμμών του grid = (N + THEAD_BLOCK_Y -1) / THREAD_BLOCK_Y (div) (το οποίο ορίζουμε κατά την εκτέλεση)

- Αριθμός στηλών του thread_block = (M + THEAD_BLOCK_X -1) / THREAD_BLOCK_X (το οποίο ορίζουμε κατά την εκτέλεση).

Όπου MxN οι διαστάσεις του πίνακα C.

Παρατήρηση

Καθώς δεν γνωρίζουμε ότι το N και το M θα είναι πολλαπλάσια του πλήθους των γραμμών και του πλήθους των στηλών του thread_block αντίστοιχα δεν μπορούμε να ορίσουμε απλά τις διαστάσεις του grid ως N/THREAD_BLOCK_Y και M/THREAD_BLOCK_X αντίστοιχα. Υπάρχει περίπτωση λοιπόν να χρειαστεί να κάνουμε padding προσθέτοντας επιπλέον thread_blocks στην διάσταση (ή και τις δύο διαστάσεις) όπου το N ή το M δεν είναι πολλαπλάσια του THREAD_BLOCK_Y και THREAD_BLOCK_X αντίστοιχα. Σε μία τέτοια περίπτωση βέβαια θα έχουμε ανενεργά νήματα τα οποία πρέπει να εμποδιστούν από το να κάνουν access σε στοιχεία του πίνακα C. Κάτι τέτοιο λαμβάνεται υπόψιν και στις τρεις υλοποιήσεις μας.

Στην συνέχεια αναλύεται η κάθε υλοποίηση ξεχωριστά και επιπλέον δίνονται απαντήσεις στις ερωτήσεις της αναφοράς:

Βασική (naive) Υλοποίηση

Η συγκεκριμένη υλοποίηση αναθέτει σε κάθε νήμα εκτέλεσης τον υπολογισμό ενός στοιχείου του πίνακα C. Οι πίνακες A,B βρίσκονται αποθηκευμένοι στην global μνήμη της GPU στην οποία έχουν access όλα τα νήματα ανεξαρτήτως σε πιο thread block (και SM αντίστοιχα) αντιστοιχούν. Ο πυρήνας (kernel) που υλοποιεί τον DMM αλγόριθμο για την συγκεκριμένη υλοποίηση είναι ο ακόλουθος:

```
/*
 *  Naive kernel
 */
__global__ void dmm_gpu_naive(const value_t *A, const value_t *B, value_t *C,
                              const size_t M, const size_t N, const size_t K) {
  /*
   * FILLME: fill the code.
   */

   int row = blockIdx.y*blockDim.y + threadIdx.y;
   int col = blockIdx.x*blockDim.x + threadIdx.x;

   value_t sum = 0;

   if (row >=M || col >=N) return;

   for (int k = 0; k<K ; k++){
        sum += A[row*K + k]*B[k*N+col];
        }

    C[row*N + col] = sum;

}
```

Ο κώδικας αυτό εκτελείται από όλα τα νήματα της GPU. Αρχικά προσδιορίζουμε το (ID.x , ID.y) = (row, col) του νήματος που εκτελείται (Τα νήματα έχουν δισδιάστατο ID αφού κάθε νήμα αντιστοιχεί στον υπολογισμό ενός στοιχείου του πίνακα C). Ο υπολογισμός γίνεται ως εξής:

- row = blockIdx.y*blockDim.y + threadIdx.y

όπου:

a) blockIdx.y : η y συντεταγμένη του thread_block εντός του grid.
b) blockDim.y : η y διάσταση του thread_block.
c) threadIdx.y: η συντεταγμένη του thread εντός του thread_block.

Το τρέχων νήμα αναφέρεται στο thread_block στο οποίο ανήκει.

- col = blockIdx.x*blockDim.x + threadIdx.x

όπου:

d) blockIdx.x : η y συντεταγμένη του thread_block εντός του grid.

e) blockDim.x : η y διάσταση του thread_block.
f) threadIdx.x: η συντεταγμένη του thread εντός του thread_block.

Το τρέχων νήμα αναφέρεται στο thread_block στο οποίο ανήκει.

Στην συνέχεια ελέγχουμε είτε εάν το row είναι μεγαλύτερο του M είτε εάν το col είναι μεγαλύτερο του N. Με αυτόν τον τρόπο αποφεύγουμε την περίπτωση που αναφέραμε στην αρχή του κεφαλαίου 2.1 (διαστάσεις πίνακα C όχι πολλαπλάσια των διαστάσεων του grid και άρα padding).

Έπειτα εκτελούμε τον υπολογισμό του στοιχείου C[row][col] (η γραφή είναι λίγο διαφορετική καθώς έχουμε τον πίνακα ξετυλιγμένο κατά γραμμές σε μία διάσταση αλλά αναφερόμαστε σε αυτό το στοιχείο). Ο υπολογισμός αυτός γίνεται επαναληπτικά με βάση τον τύπο που παραθέσαμε στην εισαγωγή της αναφοράς.

1. Υπολογίστε τον αριθμό των προσβάσεων στην κύρια μνήμη για τους πίνακες A και B συναρτήσει των διαστάσεων M , N , K του προβλήματος και των διαστάσεων του block νημάτων (THREAD_BLOCK_X για 1Δ block και THREAD_BLOCK_X / THREAD_BLOCK_Y για δισδιάστατο block).

Το κάθε νήμα για τον υπολογισμό του στοιχείου C[i][j] που του έχει ανατεθεί χρειάζεται να κάνει 2K προσβάσεις στην κύρια μνήμη (global memory) καθώς χρειάζεται να φέρει μία φορά όλα τα στοιχεία της γραμμής i του πίνακα A (K στοιχεία) και όλα τα στοιχεία της στήλης j του πίνακα B (K στοιχεία επίσης). Συνολικά λοιπόν γίνονται 2NMK προσβάσεις στην κύρια μνήμη για τους πίνακες A και B.

2. Υπολογίστε το πηλίκο των πράξεων κινητής υποδιαστολής (floating-point operations) προς τον αριθμό των προσβάσεων στην κύρια μνήμη μίας επανάληψης του εσωτερικού βρόγχου της υλοποίησης σας. Η επίδοση της υλοποίησης σας περιορίζεται από το εύρος ζώνης της κύριας μνήμης ή τον ρυθμό εκτέλεσης πράξεων κινητής υποδιαστολής των πολυεπεξεργαστκιών στοιχείων (compute-bound); Αιτιολογείστε την απάντηση σας.

Σε μία επανάληψη του εσωτερικού βρόγχου της υλοποίησης έχουμε 1 πρόσθεση και έναν πολλαπλασιασμό floating point αριθμών για κάθε νήμα. Επομένως 2 flops για κάθε νήμα. Αντίστοιχα έχουμε δύο προσβάσεις στην κύρια μνήμη (μία για το στοιχείο του πίνακα A και μία για το στοιχείο του πίνακα B) φέρνοντας κάθε φορά έναν floating point αριθμό (4 bytes). Σε κάθε επανάληψη λοιπόν του εσωτερικού βρόγχου φέρνουμε 8 bytes. Παρατηρούμε ότι το πηλίκο flops/bytes για κάθε νήμα ισούται με 2/8 = ¼ = 0.25 flops/bytes σε μία επανάληψη του εσωτερικού βρόγχου. Κοιτάζοντας τις πληροφορίες για την κάρτα γραφικών τρίτης γενιάς NVDIA Tesla K40c , αρχιτεκτονική Kepler) πάνω στην οποία τρέχει ο πυρήνας βλέπουμε ότι έχει Bandwith ίσο με 288GB/s και μέγιστη floating point performance 5.046TFLOPS./sec Η floating point performance του κώδικα μας είναι (1flop/4bytes) * 288GB/s = 72.1 GFLOPS/sec.

Καταλαβαίνουμε λοιπόν ότι η floating point επίδοση μας είναι πολύ μικρότερη από την μέγιστη επίδοση που προσφέρει η συγκεκριμένη GPU. Επομένως η επίδοση της υλοποίησης μας είναι memory bound (μοντέλο roofline).

3. Ποιες από τις προσβάσεις στην κύρια μνήμη συνενώνονται και ποιες όχι με βάση την υλοποίηση μας.

Για να έχουμε συνένωση των προσβάσεων στην κύρια μνήμη (global memory) θέλουμε διαδοχικά νήματα που ανήκουν στο ίδιο warp (άρα και στο ίδιο thread_block) να προσπελαύνουν διαδοχικές θέσεις της κύριας μνήμης. Θυμίζουμε ότι τα νήματα που ανήκουν στο ίδιο warp εκτελούν σε κάθε χτύπο του ρολογιού ακριβώς την ίδια εντολή.

Με βάση την υλοποίηση μας λοιπόν καταλαβαίνουμε ότι καμία πρόσβαση του πίνακα A δεν συνενώνεται. Κάτι τέτοιο οφείλεται στο γεγονός ότι διαδοχικά νήματα του ίδιο warp εάν ανήκουν στην ίδια γραμμή i του thread_block προσπελαύνουν σε κάθε iteration το ίδιο ακριβώς στοιχείο του πίνακα A (το A[i][k]). Επομένως δεν προσπελαύνουν ταυτόχρονα διαδοχικά στοιχεία του πίνακα A και άρα δεν υπάρχει συνένωση. Αντίστοιχα στοιχεία του ίδιου warp που ανήκουν σε διαφορετικές γραμμές προσπελαύνουν σε ένα συγκεκριμένο iteration στοιχεία της ίδιας στήλης του πίνακα A. Όμως επειδή οι πίνακες θεωρούμε ότι είναι αποθηκευμένοι κατά γραμμές τα στοιχεία αυτά δεν αποτελούν διαδοχικά στοιχεία της global memory. Επομένως ούτε αυτές οι προσβάσεις συνενώνονται.

Για τον πίνακα B όμως threads ενός warp που ανήκουν στην ίδια γραμμή του thread_block προσπελαύνουν ταυτόχρονα διαδοχικά στοιχεία της ίδιας γραμμής του πίνακα B. Επομένως οι προσβάσεις αυτές στον πίνακα B συνενώνονται.

Καταλαβαίνουμε λοιπόν πως οι προσβάσεις στον πίνακα A δεν συνενώνονται ενώ αντίθετα οι προσβάσεις στον πίνακα B συνενώνονται.

Συνένωση των προσβάσεων στην κύρια μνήμη:

Σκοπός της συγκεκριμένης υλοποίησης είναι να τροποποιήσει την προηγούμενη ώστε να επιτύχουμε συνένωση στις προσβάσεις του πίνακα A προφορτώνοντας τμηματικά στοιχεία του στην τοπική μνήμη των πολυεπεξεργαστικών στοιχείων (shared memory).

Η ιδέα της συγκεκριμένης υλοποίησης βασίζεται στο γεγονός πως κάθε thread_block έχει μία shared memory στην οποία έχουν πρόσβαση μόνο τα νήματα που ανήκουν σε αυτό. Το access time στην συγκεκριμένη μνήμη είναι τάξης μικρότερο από αυτό στην global memory καθιστώντας την μνήμη αυτή σαν ένα είδος cache για τα νήματα του κάθε thread_block. Η μνήμη αυτή όμως έχει σαφώς μικρότερη χωρητικότητα από την global memory και επομένως χρειάζεται προσοχή στην χρήση της. Με την συγκεκριμένη υλοποίηση λοιπόν θέλουμε να μεταφέρουμε τα δεδομένα του πίνακα A στην shared memory πετυχαίνοντας συνένωση αναφορών καθώς κάνουμε access τον πίνακα A για την μεταφορά. Έπειτα θα εκτελέσουμε κανονικά τον αλγόριθμο όπως και στην naive υλοποίηση αλλά με access αυτήν την φορά στην shared memory η οποία όπως είπαμε έχει πολύ μικρότερο latency. Λόγο όμως του μικρού μεγέθους της shared memory εκτελούμε tiled έκδοση του αλγορίθμου χωρίζοντας τον πίνακα A σε Tiles μεγέθους TILE_Y x TILE_X όπου κάθε thread_block μεταφέρει στην shared memory το αντίστοιχο tile του πίνακα A που χρειάζεται σε κάθε χρονικό βήμα.

Ο πυρήνας (kernel) που υλοποιεί τον DMM αλγόριθμο είναι ο ακόλουθος:

```
/*
 *  Coalesced memory acceses of A.
 */
__global__ void dmm_gpu_coalesced_A(const value_t *A, const value_t *B,
                                    value_t *C, const size_t M, const size_t N,
                                    const size_t K) {
  /*
   * FILLME: fill the code.
   */

  __shared__ value_t A_shared[TILE_Y][TILE_X];

  int ty = threadIdx.y;
  int tx = threadIdx.x;

  int row = blockIdx.y*TILE_Y + ty;
  int col = blockIdx.x*TILE_X + tx;


  value_t sum = 0;

  for (int m = 0; m < (K+TILE_X-1)/TILE_X; m++){
        A_shared[ty][tx] = A[row*K + m*TILE_X + tx];

        __syncthreads();

        for (int k =0; k<TILE_X; k++){
                sum += A_shared[ty][k]*B[(m*TILE_X+k)*N + col];
        }

        __syncthreads();
  }

  C[row*N + col] = sum;
}
```

Θεωρούμε όπως υποδεικνύεται και στην εκφώνηση ότι πάντα θα ισχύει THREAD_BLOCK_X = TILE_X και THREAD_BLOCK_Y = TILE_Y για να είναι ορθή η υλοποίηση που παραθέσαμε. Με άλλα λόγια το thread_block και το tile ταυτίζονται στην υλοποίηση μας.

Ο αλγόριθμος λειτουργεί ως εξής :

- Αρχικά βρίσκομαι και πάλι το ID (row,col) του κάθε νήματος.
- Για m=0 τα νήματα των thread_blocks που ανήκουν στην i γραμμή του grid θα μεταφέρουν το αντίστοιχο i tile της πρώτης στήλης του grid. Για παράδειγμα παίρνουμε το thread_block (0,0) με βάση το grid (πάνω αριστερά block). Τα στοιχεία της πρώτης γραμμής του thread_block αυτού θα μεταφέρουν την πρώτη γραμμή του tile (0,0) (με βάση το grid) στην shared memory. Πιο συγκεκριμένα το thread (0,0) θα μεταφέρει το στοιχείο (0,0) του tile , το thread (0,1) θα μεταφέρει το στοιχείο (0,1) του tile κ.ο.κ. Τα threads της δεύτερης γραμμής του συγκεκριμένου thread block θα κάνουν την μεταφορά της δεύτερης γραμμής του ίδιου tile με τον ίδιο τρόπο (διαδοχικά threads μεταφέρουν διαδοχικές θέσεις μνήμης). Αντίστοιχα τα threads του thread

block (0,1) με βάση το grid θα μεταφέρουν ακριβώς τα ίδια στοιχεία του πίνακα A (tile (0,0)) όπως και τα threads του thread_block (0,0) με τον ίδιο τρόπο.

Για m=1 εφαρμόζεται η ίδια διαδικασία για τα tiles της δεύτερης στήλη του grid κ.ο.κ.

Παρατηρούμε ότι διαδοχικά νήματα που ανήκουν στην γραμμή του ίδιου thread_block (ανήκουν δηλαδή στο ίδιο warp) κάνουν access σε διαδοχικά στοιχεία του πίνακα A τα οποία μεταφέρουν στην shared memory τους. Έτσι λοιπόν η μεταφορά αυτή παρατηρείται με συνένωση των προσβάσεων στην κύρια μνήμη γεγονός που αποτελεί βασικό στόχο της συγκεκριμένης υλοποίησης.

- Μέσα σε ένα iteration του m λοιπόν αρχικά γίνεται η μεταφορά του κατάλληλου tile στην shared memory. Έπειτα απαιτείται συγχρονισμός μεταξύ των threads του ίδιου thread_block (barrier) ώστε να είμαστε σίγουροι ότι όλα τα δεδομένα του εκάστοτε tile μεταφέρθηκαν στην shared memory πριν αρχίσει η επεξεργασία.
- Στην συνέχεια πραγματοποιείται κανονικά η εκτέλεση του αλγορίθμου όπου όμως υπολογίζεται μερικώς το άθροισμα για κάθε στοιχείο του πίνακα C καθώς διαθέτουμε σε κάθε επανάληψη m ένα μόνο tile του πίνακα A. Γίνεται λοιπόν ο υπολογισμός λαμβάνοντας πλέον τα στοιχεία του πίνακα A από την shared memory που έχουν αποθηκευτεί και τα στοιχεία του πίνακα B από την global memory όπου όμως πραγματοποιείται συνένωση προσβάσεων όπως και στην naive υλοποίηση. Το κάθε νήμα αποθηκεύει το ενδιάμεσο αποτέλεσμα σε μία μεταβλητή sum το οποίο ανανεώνει στο επόμενο βήμα m όπου φορτώνεται στην shared memory το επόμενο tile του πίνακα A. Μετά από κάθε ανανέωση του ενδιάμεσου αποτελέσματος απαιτείται εκ νέου συγχρονισμός των νημάτων του ίδιου thread_block καθώς δεν θέλουμε νήματα του ίδιου thread_block να φορτώνουν στον πίνακα A_shared (shared memory) τα στοιχεία του επόμενου tile του global πίνακα A ενώ άλλα νήματα του ίδιου thread_block να κάνουν ακόμα access τον πίνακα A_shared προκειμένου να υπολογίσουν το ενδιάμεσο αποτέλεσμα sum.
- Μετά το τέλος των βημάτων m τα οποία ισούνται με (K+TILE_X-1)/TILE_X (όσο δηλαδή και το πλήθος των TILES στον άξονα των στηλών K του πίνακα A) το τελικό αποτέλεσμα sum του κάθε thread αποθηκεύεται στην κατάλληλη θέση C[row][col] του πίνακα C. Και σε αυτήν την περίπτωση δεν είμαστε σίγουροι ότι το K αποτελεί ακέραιο πολλαπλάσιο του TILE_X γι αυτό χρησιμοποιείται ο παραπάνω τύπος για το άνω όριο του m.

1. Υπολογίστε τον αριθμό των προσβάσεων στην κύρια μνήμη για τον πίνακα Α και συναρτήσει των διαστάσεων M , N , K του προβλήματος και των διαστάσεων του block νημάτων (THREAD_BLOCK_X για 1Δ block και THREAD_BLOCK_X / THREAD_BLOCK_Y για δισδιάστατο block).

Τώρα πλέον ένα νήμα εκτελεί πρόσβαση στην κύρια μνήμη για τον πίνακα Α K/TILE_X φορές καθώς σε κάθε μεταφορά ενός TILE στην shared memory φέρνει μόνο ένα στοιχείο από την αντίστοιχη γραμμή του πίνακα Α (τα υπόλοιπα τα φέρνουν τα γειτονικά του threads όπως εξηγήσαμε). Επομένως συνολικά οι προσβάσεις στην κύρια μνήμη για τον πίνακα Α είναι MxNxK/TILE_X από εκεί που ήταν MxNxK.

2. Υπολογίστε το πηλίκο των πράξεων κινητής υποδιαστολής (floating-point operations) προς τον αριθμό των προσβάσεων στην κύρια μνήμη μίας επανάληψης του εσωτερικού βρόγχου της υλοποίησης σας. Η επίδοση της υλοποίησης σας περιορίζεται από το εύρος ζώνης της κύριας μνήμης ή τον ρυθμό εκτέλεσης πράξεων κινητής υποδιαστολής των πολυεπεξεργαστκιών στοιχείων (compute-bound); Αιτιολογείστε την απάντηση σας.

Σε μία επανάληψη του εσωτερικού βρόγχου της υλοποίησης έχουμε 1 πρόσθεση και έναν πολλαπλασιασμό floating point αριθμών για κάθε νήμα. Επομένως 2 flops για κάθε νήμα. Αντίστοιχα έχουμε μία πρόσβαση στην κύρια μνήμη (για το στοιχείο του πίνακα Β) φέρνοντας κάθε φορά έναν floating point αριθμό (4 bytes). Παρατηρούμε λοιπόν ότι το πηλίκο flops/bytes για κάθε νήμα σε μία επανάληψη του εσωτερικού βρόγχου ισούται με 2/4 = 1/2 = 0.5 flops/bytes. Με αντίστοιχο υπολογισμό όπως και στην περίπτωση της naive υλοποίησης έχουμε floating point performance 144.2 GFLOPS/sec. Παρατηρούμε λοιπόν ότι παρότι υπάρχει σαφώς βελτίωση σε σχέση με την Naive υλοποίηση το αποτέλεσμα απέχει και πάλι πολύ από το μέγιστο floating point performance που παρέχει η συγκεκριμένη GPU. Επομένως και πάλι η συγκεκριμένη υλοποίηση είναι memory bound.

Παρατήρηση

Οι δύο σημαντικές βελτιώσεις λοιπόν που προσφέρει η συγκεκριμένη υλοποίηση είναι οι ακόλουθες:

- Πλέον οι προσβάσεις στην κύρια μνήμη για τον πίνακα Α συνενόνονται καθώς διαδοχικά thread του ίδιου warp προσπελαύνουν για τον ίδιο χτύπο ρολογιού διαδοχικές θέσεις της global μνήμης (στοιχεία της ίδιας γραμμής του πίνακα Α).
- Οι προσβάσεις στην κύρια μνήμη για τον πίνακα Α μειώνονται ανάλογα με την αύξηση του TILE_Y και αντικαθίστανται σε προσβάσεις στην shared memory του κάθε thread_block όπου εκεί το latency είναι πάρα πολύ μικρότερο.

Μείωση των προσβάσεων στην κύρια μνήμη:

Στην συγκεκριμένη υλοποίηση θέλουμε να αξιοποιήσουμε ακόμα περισσότερο την τοπική μνήμη των πολυεπεξεργαστικών στοιχείων (shared memory) για να μειώσουμε περεταίρω τις προσβάσεις στην κύρια μνήμη προφορτώνοντας και στοιχεία του B στην τοπική μνήμη των πολυεπεξεργαστικών στοιχείων. Η προφόρτωση αυτή έχει παρόμοια λογική όπως και στην περίπτωση του πίνακα A της προηγούμενης υλοποίησης διατηρώντας επίσης την συνένωση των προσβάσεων της μνήμης για τον πίνακα B (η οποία υπήρχε ακόμα και στην naive υλοποίηση) καθώς τον προφορτώνουμε στην shared memory. Ο πυρήνας (kernel) που υλοποιεί τον DMM αλγόριθμο είναι ο ακόλουθος:

```
/*
 *  Reduced memory accesses.
 */
__global__ void dmm_gpu_reduced_global(const value_t *A, const value_t *B, value_t *C,
                                       const size_t M, const size_t N, const size_t K) {
  /*
   * FILLME: fill the code.
   */

  __shared__ value_t A_shared[TILE_Y][TILE_X];
  __shared__ value_t B_shared[TILE_Y][TILE_X];

  int ty = threadIdx.y;
  int tx = threadIdx.x;

  int row = blockIdx.y*TILE_Y + ty;
  int col = blockIdx.x*TILE_X + tx;

  value_t sum = 0;

  for (int m = 0; m < (K+TILE_X-1)/TILE_X; m++){
        A_shared[ty][tx] = A[row*K + m*TILE_X + tx];
        B_shared[ty][tx] = B[(m*TILE_Y + ty) * N + col];

        __syncthreads();

        for (int k =0; k<TILE_X; k++){
                sum += A_shared[ty][k]*B_shared[k][tx];
        }

        __syncthreads();
  }

  C[row*N + col] = sum;
}
```

Θεωρούμε όπως υποδεικνύεται και στην εκφώνηση ότι πάντα θα ισχύει THREAD_BLOCK_X = TILE_X και THREAD_BLOCK_Y = TILE_Y για να είναι ορθή η υλοποίηση που παραθέσαμε. Με άλλα λόγια το thread_block και το tile ταυτίζονται στην υλοποίηση μας.

Η συγκεκριμένη υλοποίηση δεν διαφέρει καθόλου σε σχέση με την προηγούμενη ως προς τον τρόπο που προφορτώνεται ο πίνακας A στην shared memory. Επιπλέον λοιπόν του A_Shared έχουμε έναν πίνακα B_shared στον οποίο προφορτώνουμε τα στοιχεία του πίνακα B. Ο αλγόριθμος λοιπόν λειτουργεί ως εξής:

- Αρχικά βρίσκομαι και πάλι το ID (row,col) του κάθε νήματος.
- Ο πίνακας A προφορτώνεται όπως στην προηγούμενη υλοποίηση όπου το κάθε νήμα προφορτώνει το κατάλληλο στοιχείο σε κάθε επανάληψη του εξωτερικού loop m.
- Για m=0 τα νήματα των thread_blocks που ανήκουν στην j στήλη του grid θα μεταφέρουν το αντίστοιχο j tile της πρώτης γραμμής του grid. Για παράδειγμα παίρνουμε το thread_block (0,0) με βάση το grid (πάνω αριστερά block). Τα στοιχεία της πρώτης στήλης του thread_block αυτού θα μεταφέρουν την πρώτη στήλη του tile (0,0) (με βάση το grid) στην shared memory. Πιο συγκεκριμένα το thread (0,0) θα μεταφέρει το στοιχείο (0,0) του tile , το thread (1,0) θα μεταφέρει το στοιχείο (1,0) του tile κ.ο.κ. Τα threads της δεύτερης στήλης του συγκεκριμένου thread block θα κάνουν την μεταφορά της δεύτερης στήλης με τον ίδιο τρόπο. Αντίστοιχα τα threads του thread block (1,0) με βάση το grid θα μεταφέρουν ακριβώς τα ίδια στοιχεία του πίνακα A (tile (0,0)) όπως και τα threads του thread_block (0,0) με τον ίδιο τρόπο.
  Για m=1 εφαρμόζεται η ίδια διαδικασία για τα tiles της δεύτερης στήλη του grid κ.ο.κ.
  Παρατηρούμε ότι διαδοχικά νήματα της γραμμής του ίδιου thread_block (ανήκουν δηλαδή στο ίδιο warp) κάνουν access σε διαδοχικά στοιχεία της ίδιας γραμμής του πίνακα B τα οποία μεταφέρουν στην shared memory τους. Έτσι λοιπόν η μεταφορά αυτή πραγματοποιείται με συνένωση των αναφορών στην κύρια μνήμη.
- Μέσα σε ένα iteration του m λοιπόν αρχικά γίνεται η μεταφορά του κατάλληλου tile των πινάκων A και B στην shared memory. Έπειτα απαιτείται συγχρονισμός μεταξύ των threads του ίδιου thread_block (barrier) ώστε να είμαστε σίγουροι ότι όλα τα δεδομένα των εκάστοτε tiles μεταφέρθηκαν στην shared memory πριν αρχίσει η επεξεργασία.
- Στην συνέχεια πραγματοποιείται κανονικά η εκτέλεση του αλγορίθμου όπου όμως υπολογίζεται μερικώς το άθροισμα για κάθε στοιχείο του πίνακα C καθώς διαθέτουμε σε κάθε επανάληψη m ένα μόνο tile του πίνακα A και ένα του πίνακα B. Γίνεται λοιπόν ο υπολογισμός λαμβάνοντας πλέον τα στοιχεία του πίνακα A από την shared memory που έχουν αποθηκευτεί και τα στοιχεία του πίνακα B επίσης από την shared memory. Το κάθε νήμα αποθηκεύει το ενδιάμεσο αποτέλεσμα σε μία μεταβλητή sum την οποία ανανεώνει στο επόμενο βήμα m όπου φορτώνεται στην shared memory το επόμενο tile του πίνακα A και B αντίστοιχα. Μετά από κάθε ανανέωση του ενδιάμεσου αποτελέσματος απαιτείται εκ νέου συγχρονισμός των νημάτων του ίδιου thread_block καθώς δεν θέλουμε νήματα του ίδιου thread_block να φορτώνουν στους πίνακα A_shared και B_shared (shared memory) τα στοιχεία του επόμενου tile των global πινάκων A και B ενώ άλλα νήματα του ίδιου thread_block να κάνουν ακόμα access τον πίνακα A_shared και B_shared προκειμένου να υπολογίσουν το ενδιάμεσο αποτέλεσμα sum.

- Μετά το τέλος των βημάτων m τα οποία ισούνται με (K+TILE_X-1)/TILE_X (όσο δηλαδή και το πλήθος των TILES στον άξονα των στηλών K του πίνακα A. Και σε αυτήν την περίπτωση δεν είμαστε σίγουροι ότι το K αποτελεί ακέραιο πολλαπλάσιο του TILE_X γι αυτό χρησιμοποιείται ο παραπάνω τύπος) το τελικό αποτέλεσμα sum του κάθε thread αποθηκεύεται στην κατάλληλη θέση C[row][col] του πίνακα C.

1. Υπολογίστε τον αριθμό των προσβάσεων στην κύρια μνήμη για τον πίνακα A και συναρτήσει των διαστάσεων M , N , K του προβλήματος και των διαστάσεων του block νημάτων (THREAD_BLOCK_X για 1Δ block και THREAD_BLOCK_X / THREAD_BLOCK_Y για δισδιάστατο block).

Τώρα πλέον ένα νήμα εκτελεί πρόσβαση στην κύρια μνήμη για τον πίνακα A K/TILE_X φορές και για τον πίνακα B επίσης K/TILE_X φορές καθώς σε κάθε μεταφορά ενός TILE του πίνακα A στην shared memory φέρνει μόνο ένα στοιχείο από την αντίστοιχη γραμμή του πίνακα A (τα υπόλοιπα τα φέρνουν τα γειτονικά του threads όπως εξηγήσαμε) ενώ σε κάθε μεταφορά ενός TILE του πίνακα B στην shared memory φέρνει μόνο ένα στοιχείο από την αντίστοιχη στήλη του πίνακα B (τα υπόλοιπα τα φέρνουν τα threads της ίδιας στήλης). Επομένως συνολικά οι προσβάσεις στην κύρια μνήμη για τους πίνακες A και B είναι 2xMxNxK/TILE_X από εκεί που ήταν 2xMxNxK.

2. Υπολογίστε το πηλίκο των πράξεων κινητής υποδιαστολής (floating-point operations) προς τον αριθμό των προσβάσεων στην κύρια μνήμη μίας επανάληψης του εσωτερικού βρόγχου της υλοποίησης σας. Η επίδοση της υλοποίησης σας περιορίζεται από το εύρος ζώνης της κύριας μνήμης ή τον ρυθμό εκτέλεσης πράξεων κινητής υποδιαστολής των πολυεπεξεργαστκιών στοιχείων (compute-bound); Αιτιολογείστε την απάντηση σας.

Στην συγκεκριμένη υλοποίηση στον εσωτερικό βρόγχο δεν πραγματοποιείται καμία πρόσβαση στην κύρια μνήμη καθώς έχουμε προφορτώσει ήδη ότι χρειαζόμαστε στην shared memory τόσο για τον πίνακα A όσο και για τον πίνακα B. Επομένως η επίδοση της υλοποίησης επηρεάζεται μόνο από τον ρυθμό εκτέλεσης πράξεων κινητής υποδιαστολής (computer bound). Προφανώς και η global memory επηρεάζει την επίδοση της υλοποίησης μας απλά ο εσωτερικός βρόγχος είναι computer bound.

Χρήση της βιβλιοθήκης cuBLAS

Στην συγκεκριμένη υλοποίηση χρησιμοποιούμε την βιβλιοθήκη cuBLAS και πιο συγκεκριμένα την συνάρτηση cublasSgemm() για να υλοποιήσουμε τον πολλαπλασιασμό πινάκων. Ο πυρήνας (kernel) που υλοποιεί τον DMM αλγόριθμο είναι ο ακόλουθος:

```
/*
 *  Use of cuBLAS
 */
void dmm_gpu_cublas(const value_t *A, const value_t *B, value_t *C,
                    const size_t M, const size_t N, const size_t K) {
  /*
   * FILLME: fill the code.
   */

   int lda = N;
   int ldb = K;
   int ldc = N;

   const float alf = 1;
   const float bet = 0;
   const float *alpha = &alf;
   const float *beta = &bet;

   cublasHandle_t handle;
   cublasCreate(&handle);

   cublasSgemm(handle, CUBLAS_OP_N , CUBLAS_OP_N , N , M , K , alpha , A , lda , B , ldb , beta, C , ldc);

   cublasDestroy(handle);
}
```

Παρατήρηση

Η βιβλιοθήκη cuBLAS θεωρεί ότι οι πίνακες είναι αποθηκευμένοι στην μνήμη κατά στήλες. Έτσι η συνάρτηση cublasSgemm() δέχεται τους ανάστροφους πίνακες των δύο πινάκων που της δίνουμε σαν ορίσματα. Για να πάρουμε το σωστό αποτέλεσμα όμως εκμεταλλευόμαστε την ιδιότητα ότι $(C)^T = (AB)^T = B^T A^T$. Έτσι μετατρέπουμε κατάλληλα τον κώδικα στο αρχείο dmm_main.cu ως εξής:

```
  /* Execute and time the kernel */
  cudaEventRecord(start);
  if (kernel == GPU_CUBLAS) {
    for (size_t i = 0; i < NR_ITER; ++i)
      /* FILLME: you might need to change the arguments */
      gpu_kernels[kernel].fn(gpu_B,
                             gpu_A,
                             gpu_C,
                             M, N, K);
```

4. Πειράματα και μετρήσεις επιδόσεων

Σκοπός των μετρήσεων είναι (α') η μελέτη της επίδρασης του μεγέθους του block νημάτων/υπολογισμού στην επίδοση των υλοποιήσεων και (β') η σύγκριση της επίδοσης όλων των εκδόσεων του πυρήνα DMM. Έχουμε λοιπόν τα ακόλουθα σύνολα μετρήσεων:

- Για κάθε μία από τις παραπάνω εκδόσεις πυρήνα που υλοποιήσαμε (naive , coalesced , shmem) καταγράφουμε πως μεταβάλλεται η επίδοση για διαφορετικές διαστάσεις του block νημάτων (THREAD_BLOCK_X/Y) και υπολογισμού (TILE_X/Y) για διαστάσεις πινάκων M=N=K=2048.

  Σημείωση1: Θεωρούμε όπως επισημάναμε και προηγουμένως ότι THREAD_BLOCK_X=THREAD_BLOCK_Y=TILE_X=TILE_Y.

  Σημείωση2: Δοκιμάζουμε μόνο τετραγωνικά μεγέθη blocks-tiles με τιμές 4*4 , 8*8 , 16*16 , 32*32 (δυνάμεις του 2). Δεν μπορούμε να προχωρήσουμε σε μεγαλύτερα μεγέθη block-tile καθώς ανά block πρέπει να έχουμε μέχρι 1024 νήματα σύμφωνα με τα χαρακτηριστικά της GPU που χρησιμοποιούμε στο εργαστήριο. Αν προχωρούσαμε σε υψηλότερες τιμές το kernel μας δεν θα ξεκίναγε ποτέ όταν το καλούσε η CPU και το πρόγραμμα μας θα "έσκαγε" λόγο έλλειψης πόρων της GPU.

Έχομε λοιπόν τις ακόλουθες μετρήσεις:

Naive kernel:

THREAD_BLOCK_X = THREAD_BLOCK_Y	Elapsed Time (ms)	Performance (GFLOPS/s)
4x4	107740.578125	15.945588
8x8	28750.400391	59.755235
16x16	15649.118164	109.781708
32x32	15231.567383	112.791210

Coalesced kernel:

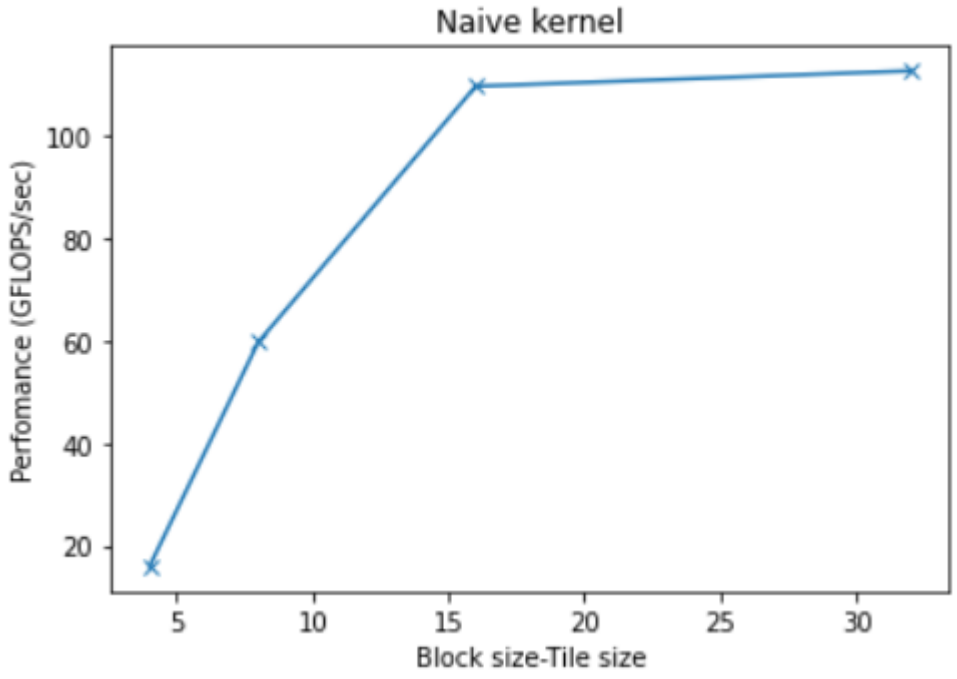
TILE_X = TILE_Y	Elapsed Time (ms)	Performance (GFLOPS/s)
4x4	85641.593750	20.060193
8x8	14914.971680	115.185396

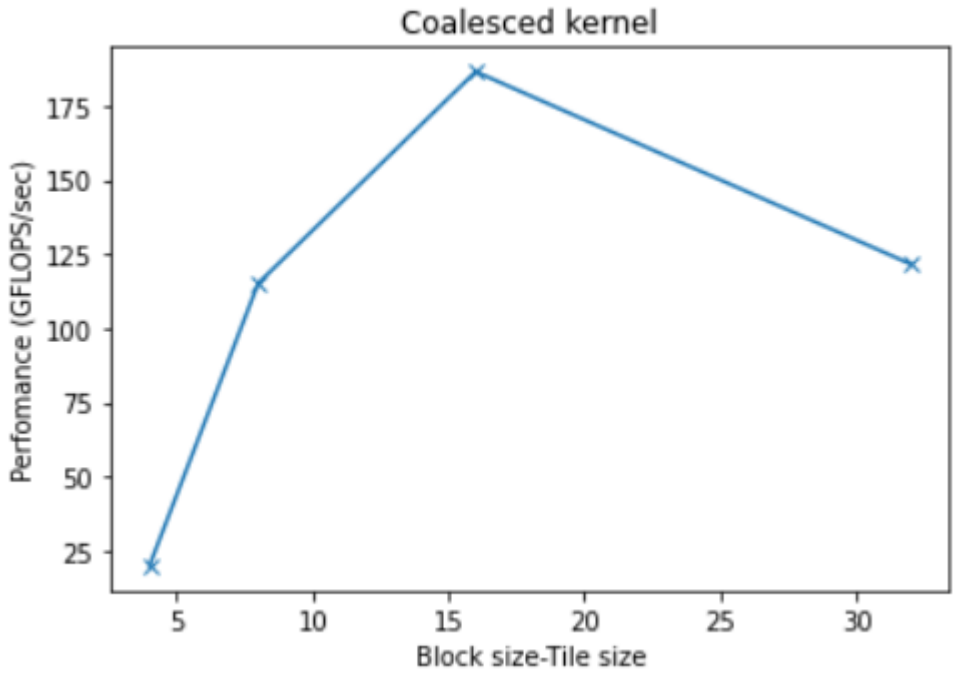
16x16	9213.229492	186.469567
32x32	14121.960938	121.653667

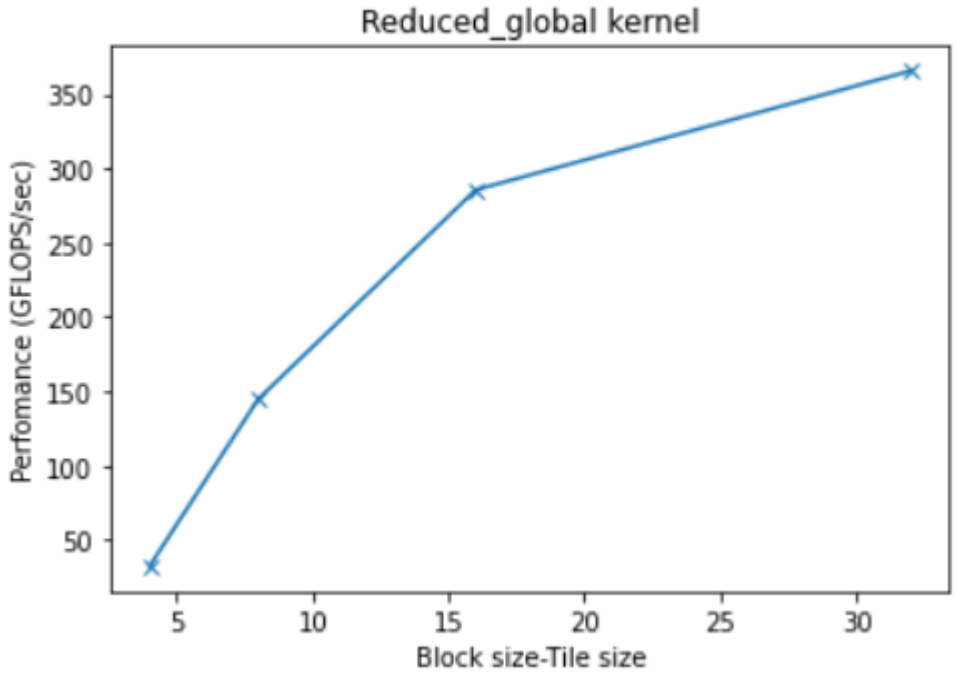
Reduced_global kernel:

TILE_X = TILE_Y	Elapsed Time (ms)	Performance (GFLOPS/s)
4x4	53561.628906	32.074957
8x8	11877.852539	144.637839
16x16	6016.200195	285.560131
32x32	4698.353027	365.657265

Στην συνέχεια παραθέτουμε και γραφικά τις παραπάνω μετρήσεις ώστε να έχουμε μία καλύτερη εικόνα:




Coalesced kernel
Perfomance (GFLOPS/sec)
175
150
125
100
75
50
25
5
10
15
20
25
30
Block size-Tile size


Reduced_global kernel
Perfomance (GFLOPS/sec)
350
300
250
200
150
100
50
5
10
15
20
25
30
Block size-Tile size

- Έπειτα για κάθε μία από τις εκδόσεις των πυρήνων (naive , coalesced , reduced_global , cuBLAS) καταγράφουμε την επίδοση για όλους τους δυνατούς συνδυασμούς μεγεθών των δύο πινάκων A , B που προκύπτουν για τιμές [256,512,1024,2048] των N,M,K. Για όλες τις μετρήσεις επιλέχθηκε THREAD_BLOCK_X=THREAD_BLOCK_Y=TILE_X=TILE_Y = 32 το οποίο αποτελεί το βέλτιστο από τα μεγέθη για τα οποία κάναμε τις προηγούμενες μετρήσεις (θα σχολιαστεί αναλυτικά στις παρατηρήσεις). Έτσι έχουμε τις ακόλουθες μετρήσεις:

Naive Kernel

Σταθερό M = 256

M	N	K	Performance(GFLOS/s)
256	256	256	95.556600
256	256	512	96.655381
256	256	1024	91.212151
256	256	2048	85.321526
256	512	256	112.256361
256	512	512	114.386856
256	512	1024	100.528719
256	512	2048	99.215399
256	1024	256	122.621069
256	1024	512	108.083972
256	1024	1024	106.957450
256	1024	2048	107.332393
256	2048	256	106.092825
256	2048	512	108.088627
256	2048	1024	107.719992
256	2048	2048	108.057708

Σταθερό M = 512

M	N	K	Performance(GFLOPS/sec)
512	256	256	111.380135
512	256	512	113.744146
512	256	1024	107.088943
512	256	2048	101.162599
512	512	256	124.505647
512	512	512	126.915039
512	512	1024	110.043950
512	512	2048	110.193126
512	1024	256	125.126817
512	1024	512	108.232518

512	1024	1024	108.733679
512	1024	2048	109.018624
512	2048	256	109.670906
512	2048	512	110.661040
512	2048	1024	111.357779
512	2048	2048	111.664281

Σταθερό M = 1024

M	N	K	Performance(GFLOPS/sec)
1024	256	256	123.043275
1024	256	512	126.040997
1024	256	1024	116.456827
1024	256	2048	112.012986
1024	512	256	126.488380
1024	512	512	128.547776
1024	512	1024	111.059214
1024	512	2048	110.856647
1024	1024	256	125.999201
1024	1024	512	111.096933
1024	1024	1024	111.624236
1024	1024	2048	111.902063
1024	2048	256	110.298029
1024	2048	512	111.393521
1024	2048	1024	112.045154
1024	2048	2048	112.423992

Σταθερό M = 2048

M	N	K	Performance(GFLOPS/sec)
2048	256	256	124.878312
2048	256	512	127.681809
2048	256	1024	115.405402
2048	256	2048	112.687324
2048	512	256	128.093147
2048	512	512	131.197303
2048	512	1024	113.731498
2048	512	2048	113.594691
2048	1024	256	128.169813
2048	1024	512	111.781480
2048	1024	1024	112.315832
2048	1024	2048	112.623536

2048	2048	256	110.861537
2048	2048	512	111.808604
2048	2048	1024	112.466454
2048	2048	2048	112.712218

Coalesced Kernel

Σταθερό M = 256

M	N	K	Performance(GFLOPS/sec)
256	256	256	100.086386
256	256	512	100.365252
256	256	1024	100.461839
256	256	2048	100.867051
256	512	256	109.594146
256	512	512	110.020386
256	512	1024	114.757187
256	512	2048	115.033238
256	1024	256	110.441523
256	1024	512	113.487578
256	1024	1024	113.885302
256	1024	2048	114.289358
256	2048	256	116.291045
256	2048	512	118.361844
256	2048	1024	119.263600
256	2048	2048	119.879378

Σταθερό M = 512

M	N	K	Performance(GFLOPS/sec)
512	256	256	107.502977
512	256	512	108.085685
512	256	1024	108.001135
512	256	2048	108.674496
512	512	256	111.049764
512	512	512	111.219011
512	512	1024	115.744478
512	512	2048	115.469486
512	1024	256	112.710976

512	1024	512	115.723380
512	1024	1024	116.405237
512	1024	2048	116.915754
512	2048	256	117.807912
512	2048	512	119.683487
512	2048	1024	120.740181
512	2048	2048	121.140352

Σταθερό M = 1024

M	N	K	Performance(GFLOPS/sec)
1024	256	256	108.683277
1024	256	512	108.821251
1024	256	1024	108.750012
1024	256	2048	110.692206
1024	512	256	112.570402
1024	512	512	113.210879
1024	512	1024	117.708822
1024	512	2048	117.797342
1024	1024	256	113.632882
1024	1024	512	115.743077
1024	1024	1024	117.501140
1024	1024	2048	118.151116
1024	2048	256	118.403787
1024	2048	512	120.301753
1024	2048	1024	121.011750
1024	2048	2048	121.554682

Σταθερό M = 2048

M	N	K	Performance(GFLOPS/sec)
2048	256	256	110.418717
2048	256	512	110.541267
2048	256	1024	110.674218
2048	256	2048	111.415348
2048	512	256	113.824681
2048	512	512	114.133884
2048	512	1024	118.674943
2048	512	2048	117.360192
2048	1024	256	113.961452
2048	1024	512	117.124153
2048	1024	1024	117.924353

2048	1024	2048	118.454812
2048	2048	256	119.331450
2048	2048	512	120.408045
2048	2048	1024	121.233342
2048	2048	2048	121.663722

Reduced_global kernel

Σταθερό M = 256

M	N	K	Performance(GFLOPS/sec)
256	256	256	286.275613
256	256	512	295.513375
256	256	1024	300.478691
256	256	2048	302.831711
256	512	256	323.629060
256	512	512	326.490857
256	512	1024	333.915751
256	512	2048	334.166481
256	1024	256	340.550949
256	1024	512	344.112174
256	1024	1024	347.154257
256	1024	2048	347.910041
256	2048	256	350.304483
256	2048	512	351.635447
256	2048	1024	357.269864
256	2048	2048	358.165594

Σταθερό M = 512

M	N	K	Performance(GFLOPS/sec)
512	256	256	325.534510
512	256	512	328.297104
512	256	1024	337.658874
512	256	2048	340.355860
512	512	256	340.074096
512	512	512	347.778194
512	512	1024	348.304322
512	512	2048	349.400052
512	1024	256	351.360600
512	1024	512	353.469078
512	1024	1024	357.261557

512	1024	2048	358.247721
512	2048	256	354.680026
512	2048	512	357.310428
512	2048	1024	361.414000
512	2048	2048	363.146961

Σταθερό M = 1024

M	N	K	Performance(GFLOPS/sec)
1024	256	256	340.377394
1024	256	512	347.798374
1024	256	1024	350.103078
1024	256	2048	351.609479
1024	512	256	351.730991
1024	512	512	357.416415
1024	512	1024	358.363500
1024	512	2048	359.128719
1024	1024	256	357.382543
1024	1024	512	357.738064
1024	1024	1024	361.498589
1024	1024	2048	363.020093
1024	2048	256	357.957674
1024	2048	512	361.513112
1024	2048	1024	364.131823
1024	2048	2048	364.983075

Σταθερό M = 2048

M	N	K	Performance(GFLOPS/sec)
2048	256	256	351.486330
2048	256	512	358.022946
2048	256	1024	359.877924
2048	256	2048	360.532592
2048	512	256	357.153232
2048	512	512	361.681642
2048	512	1024	363.959308
2048	512	2048	363.811594
2048	1024	256	358.569700
2048	1024	512	361.713434
2048	1024	1024	363.944512
2048	1024	2048	364.873462
2048	2048	256	358.827655

2048	2048	512	362.487514
2048	2048	1024	364.949836
2048	2048	2048	365.711007

cuBLAS kernel

Σταθερό M = 256

M	N	K	Performance(GFLOPS/sec)
256	256	256	7.191301
256	256	512	45.103224
256	256	1024	82.059858
256	256	2048	157.577083
256	512	256	45.966556
256	512	512	84.735466
256	512	1024	163.084425
256	512	2048	285.598262
256	1024	256	88.051438
256	1024	512	161.588745
256	1024	1024	301.716303
256	1024	2048	537.382252
256	2048	256	164.049286
256	2048	512	325.242525
256	2048	1024	546.760166
256	2048	2048	851.605738

Σταθερό M = 512

M	N	K	Performance(GFLOPS/sec)
512	256	256	47.095094
512	256	512	88.594398
512	256	1024	167.140674
512	256	2048	288.516582
512	512	256	78.847047
512	512	512	161.891054
512	512	1024	303.110912
512	512	2048	519.926724
512	1024	256	172.290764
512	1024	512	313.040169
512	1024	1024	535.406053
512	1024	2048	817.804827
512	2048	256	309.845933
512	2048	512	560.531730
512	2048	1024	875.194275

512	2048	2048	1256.857972

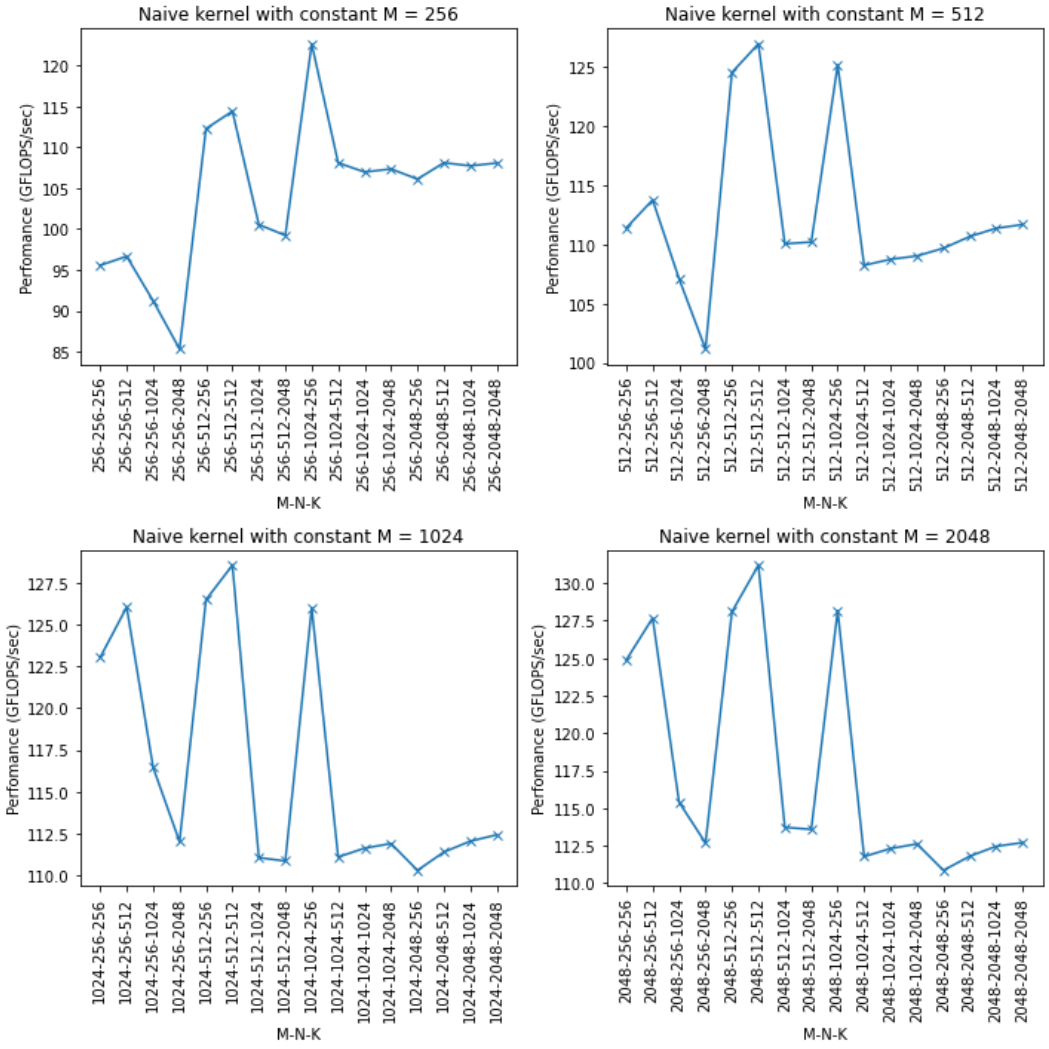
Σταθερό M = 1024

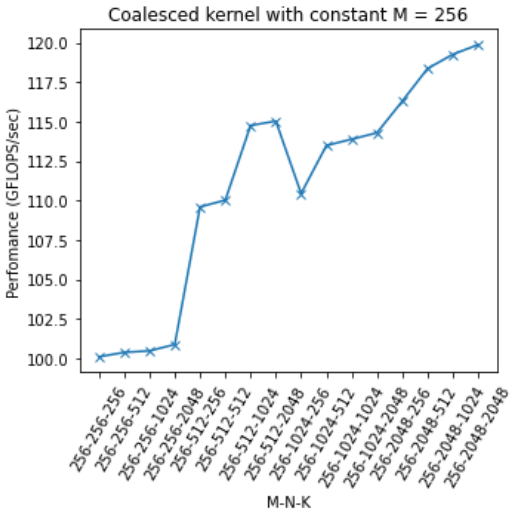
M	N	K	Performance(GFLOPS/sec)
1024	256	256	83.695586
1024	256	512	168.252556
1024	256	1024	293.307396
1024	256	2048	517.295064
1024	512	256	163.141396
1024	512	512	290.968776
1024	512	1024	526.538864
1024	512	2048	859.403333
1024	1024	256	323.437232
1024	1024	512	524.669285
1024	1024	1024	916.169325
1024	1024	2048	1232.593927
1024	2048	256	533.585991
1024	2048	512	935.236963
1024	2048	1024	1439.140161
1024	2048	2048	1954.038163

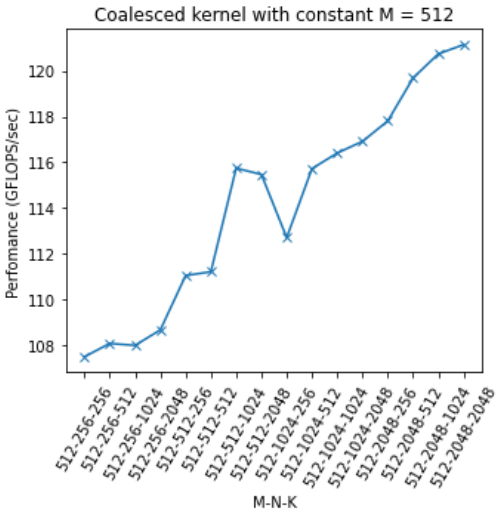
Σταθερό M = 2048

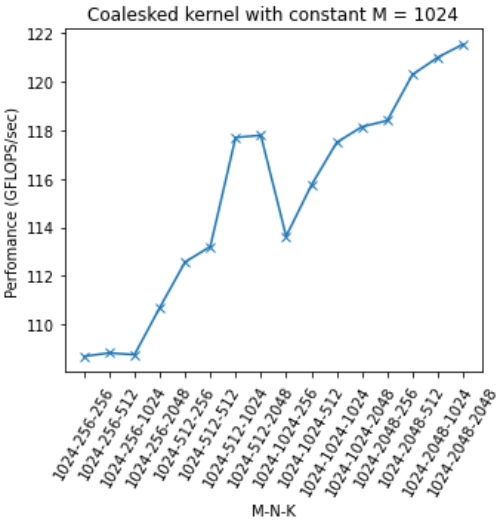
M	N	K	Performance(GFLOPS/sec)
2048	256	256	160.601593
2048	256	512	319.164905
2048	256	1024	562.548519
2048	256	2048	843.024060
2048	512	256	299.951587
2048	512	512	524.907473
2048	512	1024	819.843720
2048	512	2048	1254.215706
2048	1024	256	551.283602
2048	1024	512	965.188527
2048	1024	1024	1480.169159
2048	1024	2048	1992.237839
2048	2048	256	868.965346
2048	2048	512	1434.760004
2048	2048	1024	1916.353351
2048	2048	2048	2363.756195

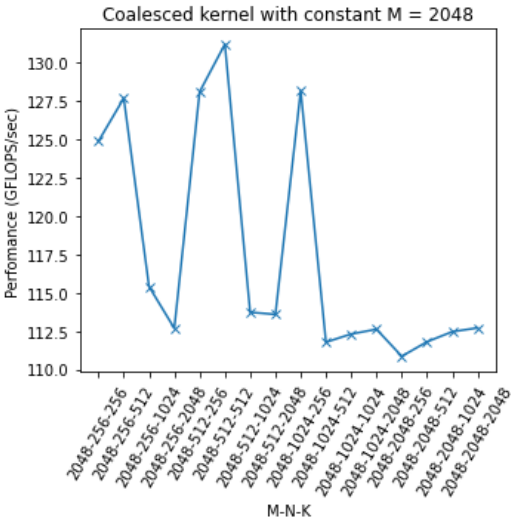
Στην συνέχεια παραθέτουμε γραφικά τις παραπάνω μετρήσεις για να έχουμε μία καλύτερη εικόνα. Για κάθε kernel έχουμε 4 subplots όπου στο καθένα έχουμε σταθερή τιμή του M = [256,512,1024,2048]. Έτσι προκύπτουν οι ακόλουθες γραφικές αναπαραστάσεις:

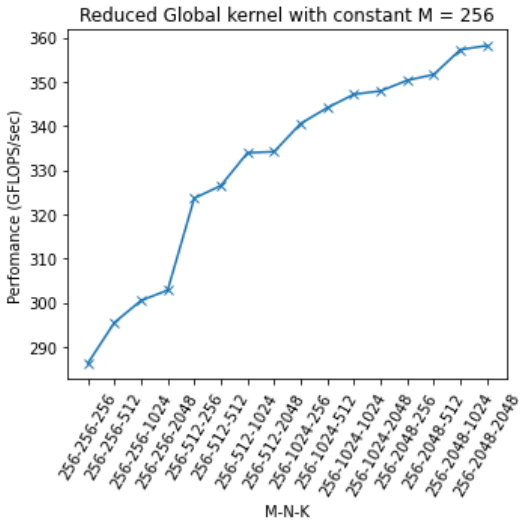


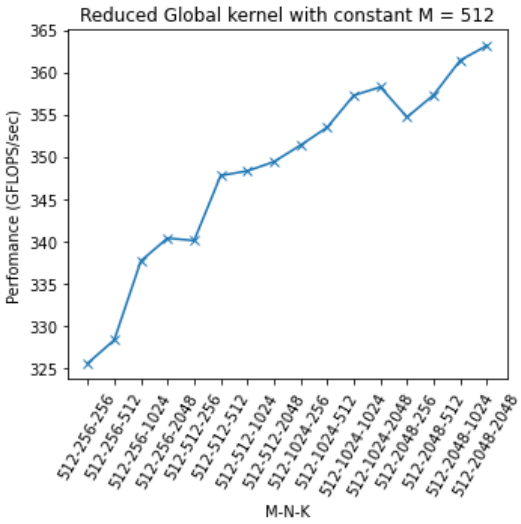

Coalesced kernel with constant M = 256
Perfomance (GFLOPS/sec)
M-N-K

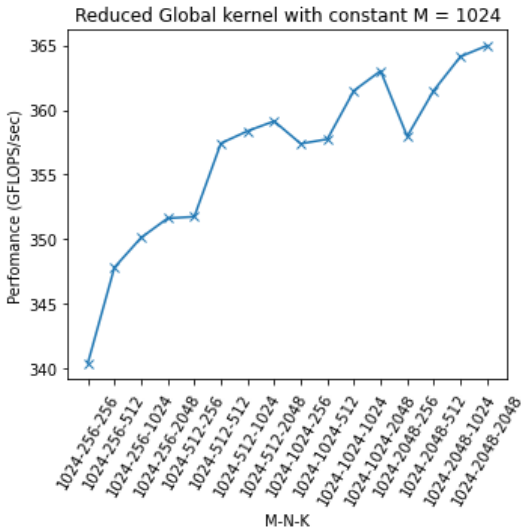

Coalesced kernel with constant M = 512
Perfomance (GFLOPS/sec)
M-N-K

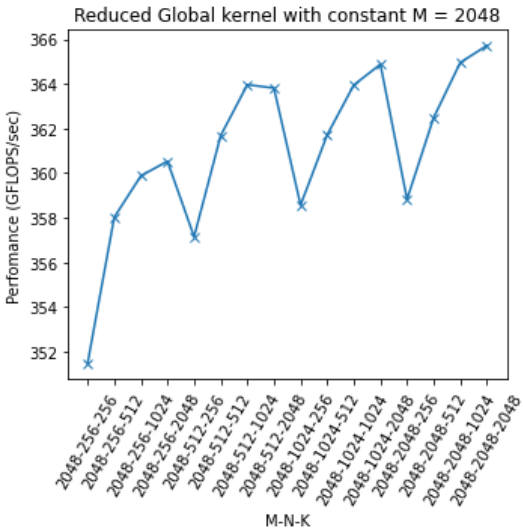

Coalesked kernel with constant M = 1024
Perfomance (GFLOPS/sec)
M-N-K

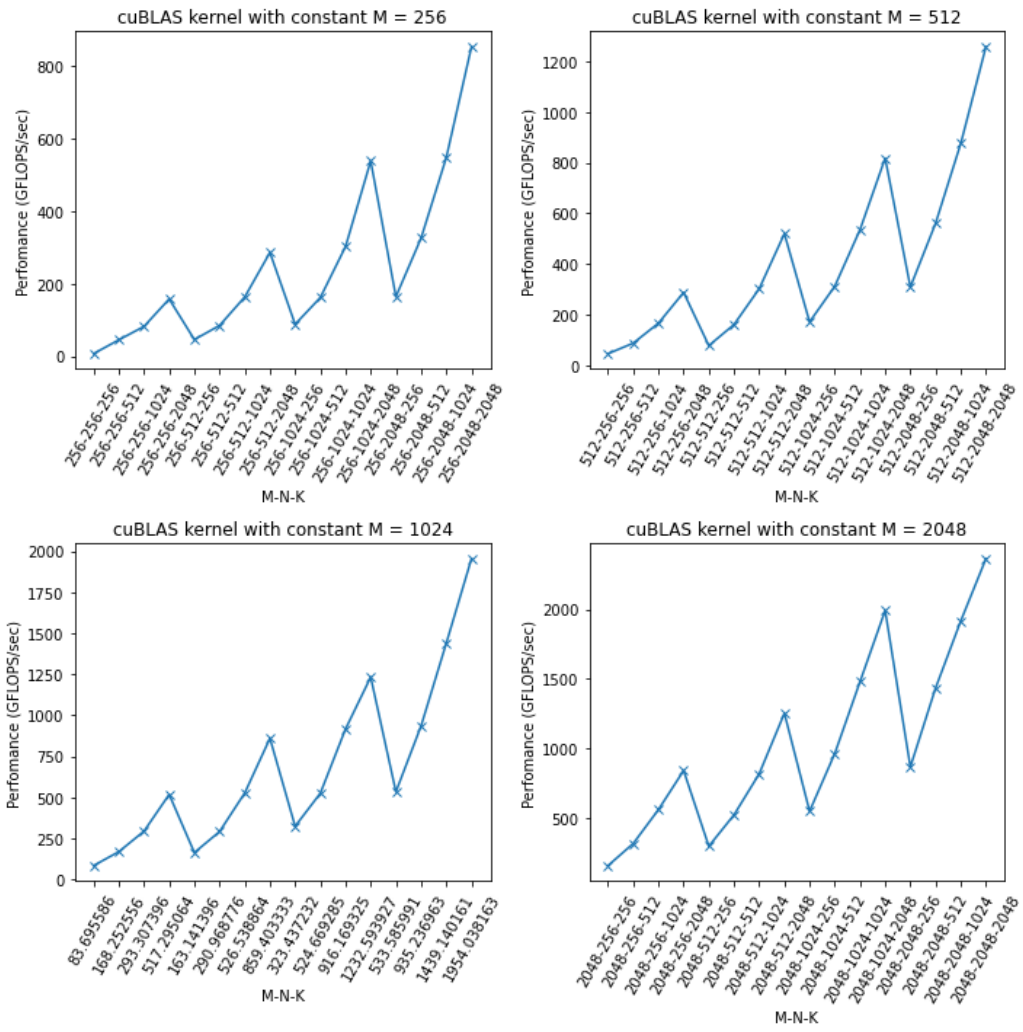

Coalesced kernel with constant M = 2048
Perfomance (GFLOPS/sec)
M-N-K


Reduced Global kernel with constant M = 256
Perfomance (GFLOPS/sec)
M-N-K


Reduced Global kernel with constant M = 512
Perfomance (GFLOPS/sec)
M-N-K


Reduced Global kernel with constant M = 1024
Perfomance (GFLOPS/sec)
M-N-K


Reduced Global kernel with constant M = 2048
Perfomance (GFLOPS/sec)
M-N-K



Ερμηνεία της συμπεριφοράς και συμπεράσματα:

- Επίδοση για διαφορετικές διαστάσεις thread_block και M=N=K=2048

  Παρατηρούμε ότι και για τις τρεις εκδόσεις πυρήνων (naive , coalesced , reduced_global) η αύξηση του size του thread_block οδηγεί και στην αύξηση του performance των τριών πυρήνων. Κάτι τέτοιο είναι λογικό καθώς αυξάνοντας το size του thread_block έχουμε όλο και περισσότερα νήματα που τρέχουν παράλληλα καθώς αυξάνεται ο αριθμός των warps ενός thread block. Έτσι καταφέρνουμε σε έναν βαθμό να κρύψουμε την καθυστέρηση που υπάρχει στην πρόσβαση στην μνήμη καθώς κάνουμε παράλληλα κάποια άλλη χρήσιμη λειτουργία ενώ στις εκδόσεις coalesced και reduced_global πετυχαίνουμε όλο και περισσότερη συνένωση αναφορών στην κύρια μνήμη. Βέβαια όπως επισημάναμε το block_size δεν μπορεί να αυξηθεί πάνω από 32x32 καθώς σε μία τέτοια περίπτωση ο πυρήνας μας δεν θα μπορούσε να εκτελεστεί λόγο έλλειψης πόρων. Μία σημαντική παρατήρηση που κάνουμε

βέβαια είναι ότι στην περίπτωση του coalesced πυρήνα για thread_block 32x32 παρατηρούμε μείωση του performance σε σχέση με προηγούμενες τιμές. Αυτό ίσως να οφείλεται στο γεγονός πως το overhead του συγχρονισμού που απαιτείται μεγαλώνει (μεγαλύτερο μπλοκ size σημαίνει περισσότερα warps εντός του thread_block και άρα περισσότερα threads περιμένουν). Το overhead αυτό ίσως να μην μπορεί να καλυφθεί από την χρήση της shared memory για τον πίνακα A. Τέλος να επισημάνουμε ότι όσο πηγαίνουμε σε καλύτερη υλοποίηση η αύξηση της επίδοσης με την αύξηση του thread_block είναι πολύ μεγαλύτερη. Κάτι τέτοιο είναι αναμενόμενο λόγο και φυσικά της καλύτερης εκμετάλλευσης της shared memory και της συνένωσης προσβάσεων στην κύρια μνήμη που αναφέραμε και προηγουμένως.

- Επίδοση των τεσσάρων εκδόσεων (naive , coalesced , reduced_global , cuBLAS) για διαφορετικές τιμές N , M , K.

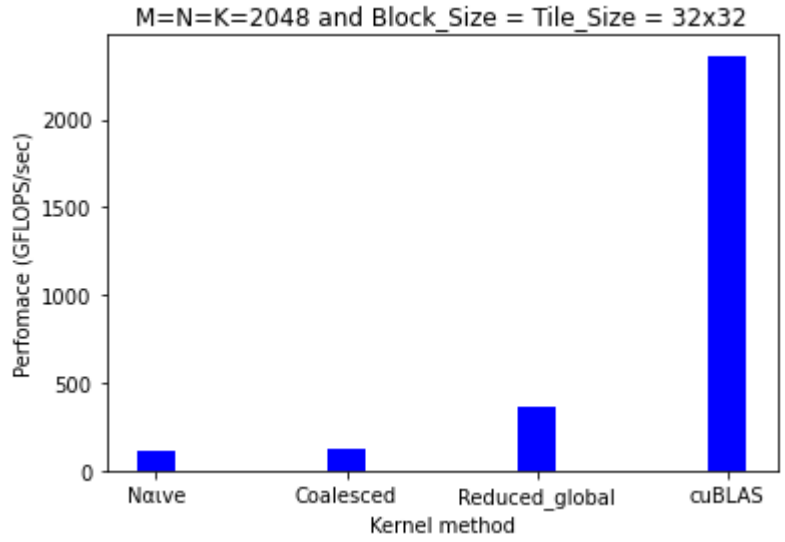
  Παρατηρούμε αρχικά ότι στην naive υλοποίηση η αύξηση της τιμής του K (ειδικά για 1024 και 2048) οδηγεί σε μείωση του performance του πυρήνα. Κάτι τέτοιο οφείλεται στο γεγονός πως αυξάνονται οι προσβάσεις που γίνονται στην κύρια μνήμη καθώς όπως είπαμε κάθε νήμα εκτελεί 2K προσβάσεις στην κύρια μνήμη (οι προσβάσεις μάλιστα αυτές όπως αναφέραμε δεν γίνονται με συνένωση των αναφορών). Η μείωση αυτή της επίδοσης με την αύξηση του K βλέπουμε πως δεν παρατηρείται στις επόμενες δύο υλοποιήσεις όπου εκεί ακριβώς μας είναι να εκμεταλλευτούμε την shared memory και να πετύχουμε συνένωση αναφορών κατά την πρόσβαση στην κύρια μνήμη.

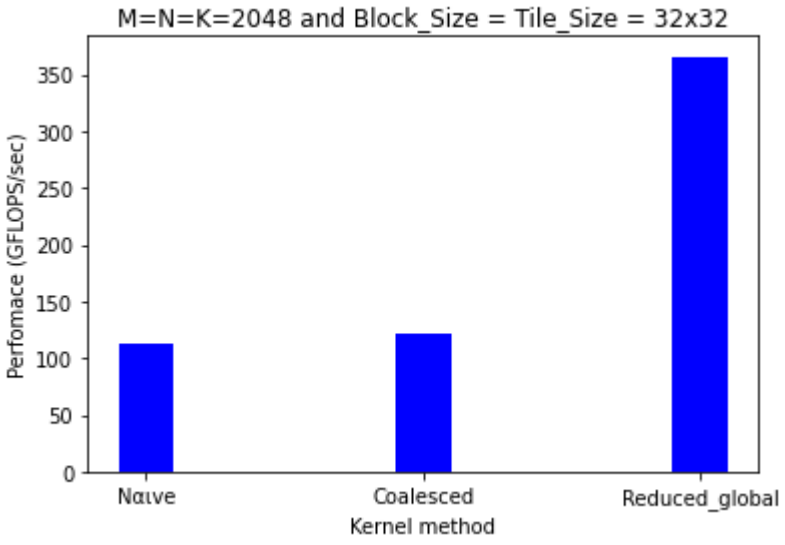
  Παρατηρούμε βέβαια ότι η αύξηση του μεγέθους των πινάκων (M,N,K) οδηγεί σε καλύτερο performance και για τις τέσσεσρις υλοποιήσεις. Κάτι τέτοιο είναι λογικό καθώς όπως καταλαβαίνουμε η GPU για να επιτύχει υψηλό performance χρειάζεται πολλά δεδομένα. Σε διαφορετική περίπτωση το transfer time που απαιτείται για να μεταφερθούν οι πίνακες A και B από την CPU στην GPU και να επιστραφεί ο πίνακας C στην CPU από την GPU αποτελεί bottleneck ενώ δεν γίνεται και fully utilized η GPU καθώς δεν μπορούν να αξιοποιηθούν τα χιλιάδες threads που αυτή υποστηρίζει. Παρόλο λοιπόν που θα μπορούσαμε να πούμε ότι ο αλγόριθμος DMM είναι στην φύση του computer bound τα λίγα δεδομένα οδηγούν σε πολύ κακή επίδοση της GPU.

  Όσον αφορά την σύγκριση των τριών υλοποιήσεων βλέπουμε την βελτίωση στην απόδοση που παρατηρείται για τα ίδια μεγέθη M , N , K καθώς μεταβαίνουμε από την πρώτη πιο naive υλοποίηση μέχρι και στην cuBLAS υλοποίηση στην οποία χρησιμοποιήσαμε απλά μία έτοιμη συνάρτηση μίας βιβλιοθήκης.

Τέλος παρατηρούμε ότι η υλοποίηση του cuBLAS παρουσιάζει τα καλύτερα αποτελέσματα όσον αφορά την επίδοση. Κάτι τέτοιο είναι λογικό καθώς η βιβλιοθήκη χρησιμοποιεί πιο προηγμένες μεθόδους υλοποίησης.

Η σύγκριση των $4^{ων}$ υλοποιήσεων φαίνεται καλύτερα και με τα ακόλουθα δύο διαγράμματα:





Υπολογισμός Transfer overhead

Ο υπολογισμός δεν πρόλαβε δυστυχώς να πραγματοποιηθεί λόγο χρόνου. Για την πραγματοποίηση του υπολογισμού χρειάζονται timestamps στην αρχή και στο τέλος της μεταφοράς των πινάκων A , B , C (έχουν αρχικοποιηθεί με τυχαίες τιμές) από την CPU στην GPU καθώς και αντίστοιχα timestamps στην επιστροφή του πίνακα C από την GPU στην CPU μετά τους υπολογισμούς. Εικάζουμε ότι το overhead που προκύπτει από την μεταφορά αυτή θα παίζει σημαντικό ρόλο για μικρότερα μεγέθη των πινάκων A , B , C όπου το Performance της GPU δεν είναι τόσο υψηλό (όπως είπαμε για λίγα δεδομένα το data transfer αποτελεί bottlneck). Καταλαβαίνουμε λοιπόν ότι για να φανούν τα οφέλη της GPU απαιτείται να έχουμε αρκετά δεδομένα.