Εθνικό Μετσόβιο Πολυτεχνείο

Συστήματα Παράλληλης Επεξεργασίας

Εργαστηριακή άσκηση 4

Παράλληλος προγραμματισμός σε επεξεργαστές γραφικών

Θρασύβουλος Ηλιάδης : 03115761

Πέππας Αθανάσιος : 03115749

1. Πολλαπλασιασμός πίνακα με πίνακα

Ο πολλαπλασιασμός πίνακα με πίνακα (Dense Matrix-Matrix multiplication, DMM) είναι ένας από τους πιο σημαντικούς υπολογιστικούς πυρήνες αλγεβρικών υπολογισμών. Έστω οι πίνακες εισόδου A και B με διαστάσεις M × K και K × N αντίστοιχα. Ο πυρήνας DMM υπολογίζει τον πίνακα εξόδου C = A · B διαστάσεων M × N.

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

Στην άσκηση αυτή υλοποιούμε τον αλγόριθμο DMM για επεξεργαστές γραφικών χρησιμοποιώντας το προγραμματιστικό περιβάλλον CUDA. Ξεκινάμε από μία απλοϊκή αρχική υλοποίηση, και καταλήγουμε σε μία αρκετά αποδοτική υλοποίηση του αλγορίθμου DMM για τους επεξεργαστές γραφικών.

Οι υλοποιήσεις είναι οι εξής:

- *Βασική υλοποίηση (naive)*
- *Συνένωση των προσβάσεων στην κύρια μνήμη (coalesced)*
- *Μείωση των προσβάσεων στην κύρια μνήμη (shmem)*
- *Με χρήση της βιβλιοθήκης cuBLAS*

Βασική υλοποίηση

Στην υλοποίηση αυτή, αναθέτουμε σε κάθε νήμα εκτέλεσης τον υπολογισμό ενός στοιχείου του πίνακα εξόδου.

```
/*
 *  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) {

  /* Compute the row and the column of the current thread */

  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int row = by*blockDim.y + ty;
  int col = bx*blockDim.x + tx;

  value_t sum = 0;

  /* If the thread's position is out of the array, it remains inactive */
  if (row >= M || col >= N) return;

  /* Compute the value of C */
  for (int k = 0; k < K; k++){
    sum += A[row*K+k]*B[col+k*N];
  }

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

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 μπλοκ νημάτων (THREAD_BLOCK_X για 1Δ μπλοκ και THREAD_BLOCK_X/THREAD_BLOCK_Y για 2Δ μπλοκ).

Κάθε νήμα κάνει `2K` προσβάσεις στη μνήμη καθώς `K` φορές προσπελαύνει τη μνήμη για να φέρει τα στοιχεία A_{ik} και B_{kj} . Συνολικά, έχουμε `2KMN` προσβάσεις στη κύρια μνήμη.

2. Υπολογίστε το πηλίκο των πράξεων κινητής υποδιαστολής (floating-point operations) προς τον ριθμό των προσβάσεων στην κύρια μνήμη μίας επανάληψης του εσωτερικού βρόχου της υλοποίησής σας. Η επίδοση της υλοποίησής σας περιορίζεται από το εύρος ζώνης της κύριας μνήμης (memory-bound) ή το ρυθμό εκτέλεσης πράξεων κινητής υποδιαστολής των πολυεπεξεργαστικών στοιχείων (compute-bound); Αιτιολογήστε την απάντησή σας.

Σε μία επανάληψη του εσωτερικού βρόχου έχουμε 2 πράξεις κινητής υποδιαστολής και 2 προσβάσεις στην κύρια μνήμη για float αριθμούς (4 byte). Συνεπώς, έχουμε 1 flop ανά 4 bytes από την κύρια μνήμη.

Η υλοποίησή μας περιορίζεται από το εύρος ζώνης της κύριας μνήμης καθώς το compute-bound είναι πάντα μεγαλύτερο από το bandwidth της κύριας μνήμης και σε αυτήν την περίπτωση οι προσβάσεις στην κύρια μνήμη μας καθυστερούν πολύ.

3. Ποιες από τις προσβάσεις στην κύρια μνήμη συνενώνονται και ποιες όχι με βάση την υλοποίησή σας;

Από τη στιγμή που οι πίνακες ειναι αποθηκευμένοι κατά γραμμές, οι προσβάσεις που γίνονται στον πίνακα A συνενώνονται γιατί διαβάζονται κατά γραμμή και κατ'επέκταση είναι συνεχόμενες θέσεις μνήμης ενώ αυτές του πίνακα B δεν συνενώνονται γιατί διαβάζονται κατα στήλες.

Συνένωση των προσβάσεων στην κύρια μνήμη (coalesced)

Στην υλοποίση αυτή τροποποιήστε την προηγούμενη υλοποίηση ώστε να να επιτύχουμε συνένωση των προσβάσεων στην κύρια μνήμη για τον πίνακα A προφορτώνοντας τμηματικά στοιχεία του στην τοπική μνήμη των πολυεπεξεργαστικών στοιχείων (shared memory).

```
/*
 *  Coalesced memory acceses
 */
__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) {

  /* Define the shared memory between the threads of the same thread block */
  __shared__ value_t A_shared[TILE_Y][TILE_X];

  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  /* Compute the tile of the current thread */
  int row = by * TILE_Y + ty;
  int col = bx * TILE_X + tx;

  value_t sum = 0;

  for(int m = 0; m < (K+TILE_X-1)/TILE_X; m++){
```

```
    /* Load the current tile in the shared memory and synchronize */
    A_shared[ty][tx] = A[row*K + m*TILE_X+tx];

    __syncthreads();

    for(int k = 0; k < TILE_X; k++){
      /* Compute the inner product of current tile and synchronize */
      sum += A_shared[ty][k]*B[(m*TILE_X+k)*N+col];
    }
    __syncthreads();
  }
  /* Save result */
  C[row*N+col] = sum;
}
```

1. Υπολογίστε τον αριθμό των προσβάσεων στην κύρια μνήμη για τον πίνακα A συναρτήσει των διαστάσεων M, N, K του προβλήματος, των διαστάσεων του μπλοκ νημάτων (THREAD_BLOCK_X για 1Δ μπλοκ και THREAD_BLOCK_X/THREAD_BLOCK_Y για 2Δ μπλοκ) και των διαστάσεων του μπλοκ υπολογισμού (TILE_X για 1Δ μπλοκ και TILE_X/TILE_Y για 2Δ μπλοκ). Κατά πόσο μειώνονται οι προσβάσεις σε σχέση με την προηγούμενη υλοποίηση;

Οι προσβάσεις στην κύρια μνήμη για τον πίνακα A μειώθηκαν από `2MNK` σε `2MNK / TILE_SIZE`.

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

Σε μια επανάληψη του εσωτερικού βρόχου έχουμε 2 πράξεις κινητής υποδιαστολής (1 πρόσθεση και 1 πολλαπλασιασμό) και μια πρόσβαση την κύρια μνήμη (για τον πίνακα B), δηλαδή 1 flop ανά 2 bytes. Πάλι περιοριζόμαστε από την κύρια μνήμη αλλά τώρα μπορούμε να πετύχουμε περισσοερα Gflops σε σχέση με πριν.

Μείωση των προσβάσεων στην κύρια μνήμη (shmem)

Στην υλοποίηση αυτή αξιοποιούμε την τοπική μνήμη των πολυεπεξεργαστικών στοιχείων (shared memory) για να μειώσουμε περαιτέρω τις προσβάσεις στην κύρια μνήμη προφορτώνοντας τμηματικά και στοιχεία του B στην
τοπική μνήμη των πολυεπεξεργαστικών στοιχείων (shared memory), ενώ ο τρόπος ανάθεσης υπολογισμών σε νήματα παραμένει ως έχει.

```
/*
 *   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) {

  /* Define the shared memory between the threads of the same thread block */
  __shared__ value_t A_shared[TILE_Y][TILE_X];
  __shared__ value_t B_shared[TILE_Y][TILE_X];

  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  /* Compute the tile of the current thread */
  int row = by * TILE_Y + ty;
  int col = bx * TILE_X + tx;

  value_t sum = 0;

  for(int m = 0; m < (K+TILE_X-1)/TILE_X; m++){
    /* Load the currect tile of A and B in the shared memory and synchronize  */
    A_shared[ty][tx] = A[row*K + m*TILE_X+tx];
    B_shared[ty][tx] = B[col + (m*TILE_Y+ty)*N];

    __syncthreads();

    for(int k = 0; k < TILE_X; k++){
      /* Compute the inner product of the current tile and synchronize */
      sum += A_shared[ty][k]*B_shared[k][tx];
    }
    __syncthreads();
  }
  /* Save result */
  C[row*N+col] = sum;
}
```

1. Υπολογίστε τον αριθμό των προσβάσεων στην κύρια μνήμη για τον πίνακα Α συναρτήσει των διαστάσεων M, N, K του προβλήματος, των διαστάσεων του μπλοκ νημάτων (THREAD_BLOCK_X για 1Δ μπλοκ και THREAD_BLOCK_X/THREAD_BLOCK_Y για 2Δ μπλοκ) και των διαστάσεων του μπλοκ υπολογισμού (TILE_X για 1Δ μπλοκ και TILE_X/TILE_Y για 2Δ μπλοκ). Κατά πόσο μειώνονται οι προσβάσεις σε σχέση με την προηγούμενη υλοποίηση;

Από εκεί που οι προσβάσεις στην κύρια μνήμη για τον πίνακα A και B ήταν `2MNK` τώρα μειώθηκαν σε `2MNK / TILE_SIZE` (?).

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

Στον εσωτερικό βρόχο της υλοποίησης μας δεν έχουμε πρόσβαση στην κύρια μνήμη, καθώς έχουμε μταφέρει ότι χρειαζόμαστε την shared memory. Επομένως, η επίδοση της υλοποίησης επηρεάζεται μόνο από τον ρυθμό εκτέλεσης πράξεων κινητής υποδιαστολής των SM (computebound). Προφανώς, και περιορίζει η μνήμη την συνολική επίδοση απλά αυτό που αναφέρουμε είναι ότι ο εσωτερικός βρόχος υπολογισμού είναι compute-bound.

Με χρήση της βιβλιοθήκης cuBLAS

Στην υλοποίηση αυτή χρησιμοποιούμε τη συνάρτηση cublasSgemm() της βιβλιοθήκης cuBLAS για τον πολλαπλασιασμού πινάκων.

```
/*
 *  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) {

  /* Define parameters for cublasSgemm */

  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;

  /* Create a handle for CUBLAS */
  cublasHandle_t handle;
  cublasCreate(&handle);

  /* Compute the matrix multiplication */
  cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K, alpha, A, lda, B, ldb,
beta, C, ldc);

  /* Destroy the handle */
  cublasDestroy(handle);
}
```

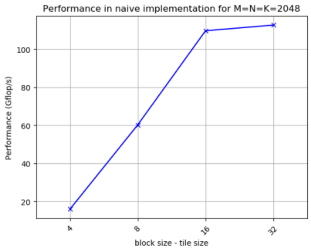
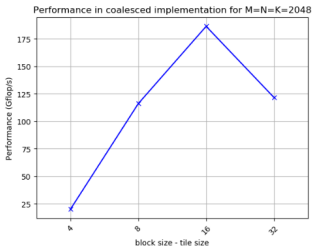
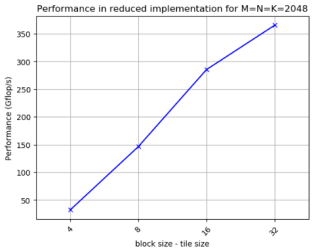
Η βιβλιοθήκη cuBLAS θεωρεί ότι οι πίνακες είναι αποθηκευμένοι στη μνήμη κατά στήλες συνεπώς για την τιμή της παραμέτρου trans κάνουμε χρήση της ιδιότητας πινάκων $C^T = (AB)^T = B^T A^T$

Συνεπώς μέσα στο dmm_gpu.cu καλούμε την συνάρτηση ως εξής

```
if (kernel==GPU_CUBLAS) {
    for (size_t i=0; i<NR_ITER; ++i)
        gpu_kernels[kernel].fn(gpu_B, gpu_A, gpu_C, M, N, K) ;
}
```

3. Πειράματα, μετρήσεις επιδόσεων και συμπεράσματα

Για κάθε έκδοση πυρήνα που υλοποιήσατε (naive, coalesced, shmem) να καταγράψετε πώς μεταβάλλεται η επίδοση για διαφορετικές διαστάσεις του μπλοκ νημάτων (THREAD_BLOCK_X/Y) και υπολογισμού (TILE_X/Y) για διαστάσεις πινάκων M = N = K = 2048.

naive	coalesced	shmem
		

Συμπέρασμα

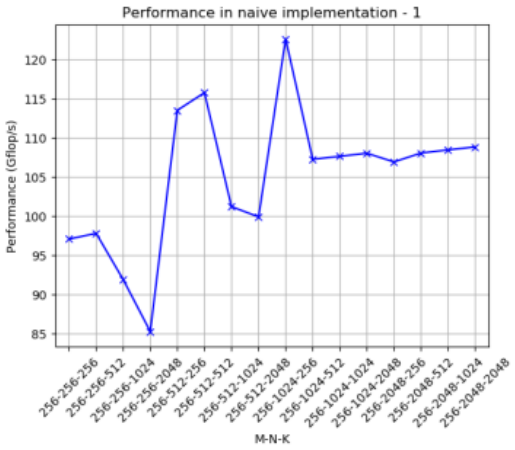
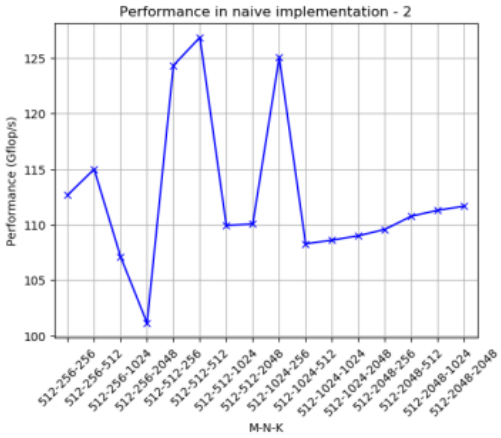
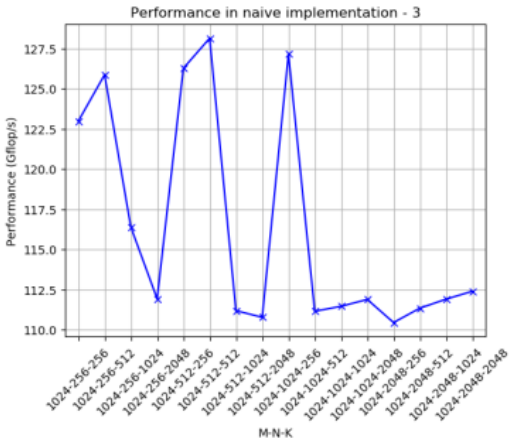
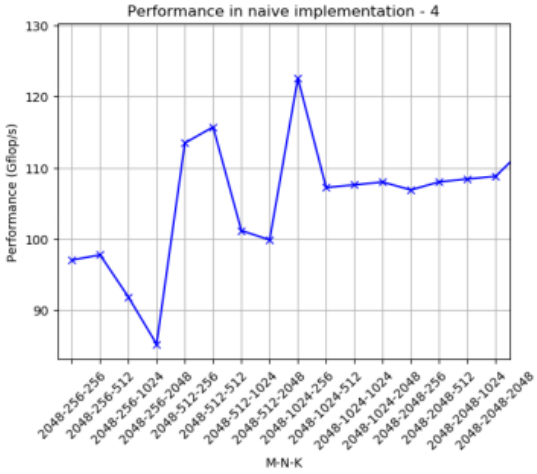
Αρχικά όπως είναι αναμενόμενο, όσο προχωράμε στην βελτιστοποίηση των υλοποιήσεων η επίδοση είναι καλύτερη.

Όπως φαίνεται και από τα διαγράμματα, όσο αυξάνουμε το μπλοκ νημάτων τόσο αυξάνεται η επίδοση σε όλες τις εκδόσεις. Αυτό συμβαίνει διότι όσα περισσότερα νήματα έχουμε τόσο κερδίζουμε με την παραλληλία τις όποιες καθυστερήσεις έχουμε λόγω των προσβάσεων στην μνήμη. Για τις υλοποιήσεις naive και shmem η βέλτιστη επίδοση είναι για μπλοκ νημάτων `32*32` ενώ για την coalesced έκδοση, είνια αυτή του `16*16`. Αυτό συμβαίνει γιατι εδώ ο συγχρονισμός των νημάτων επιδρά αρνητικά στους χρόνους εκτέλεσης.

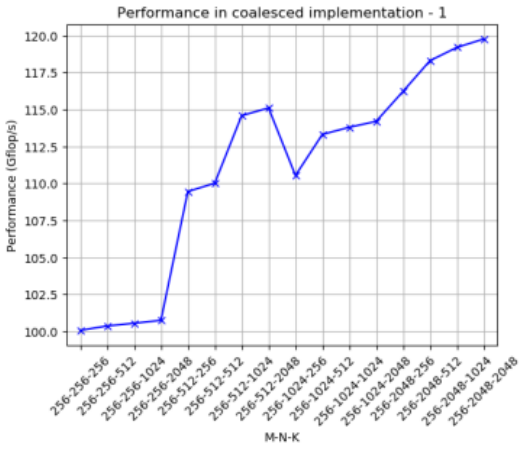
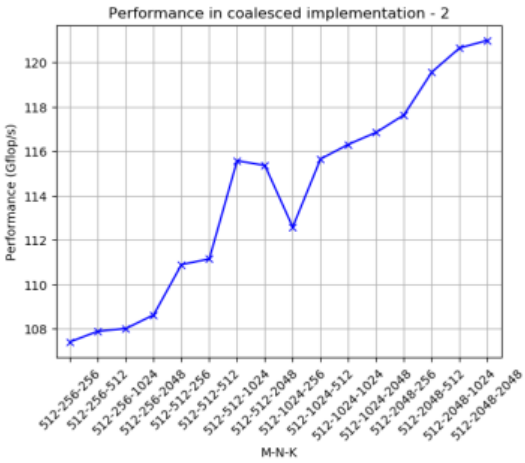
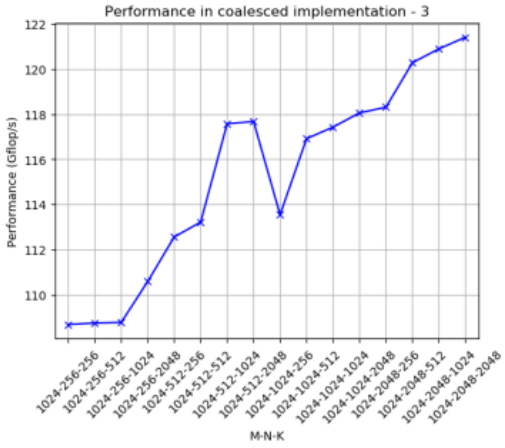
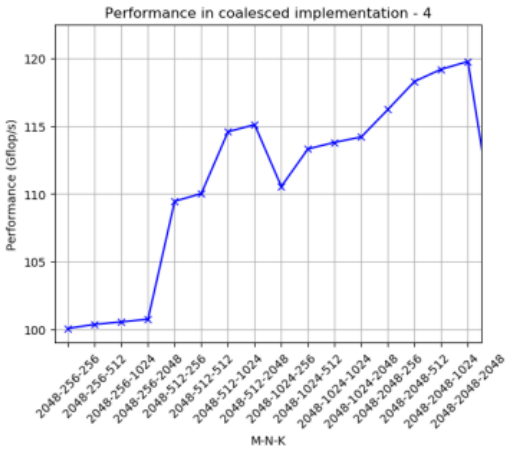
Για κάθε μία από τις τέσσερις εκδόσεις πυρήνων (naive, coalesced, shmem και cuBLAS) θα καταγράψουμε την επίδοση για μεγέθη πινάκων M, N, K ∈[256, 512, 1024, 2048]. Για τις naive, coalesced και shmem υλοποιήσεις επιλέξτε τις βέλτιστες διαστάσεις μπλοκ νημάτων και υπολογισμών.

Επειδή οι συνδιασμοί μεγέθους πινάκων είναι πολλοί για να χωρέσουν σε ένα μονο διάγραμμα, επιλέξαμε να κάνουμε 4 διαγράμματα όπου στο καθένα κρατάμε σταθερό το M.

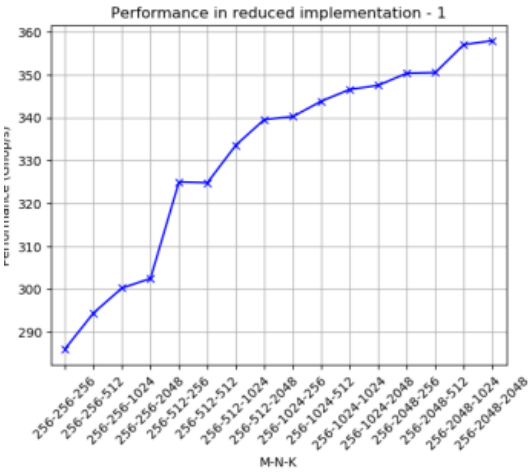
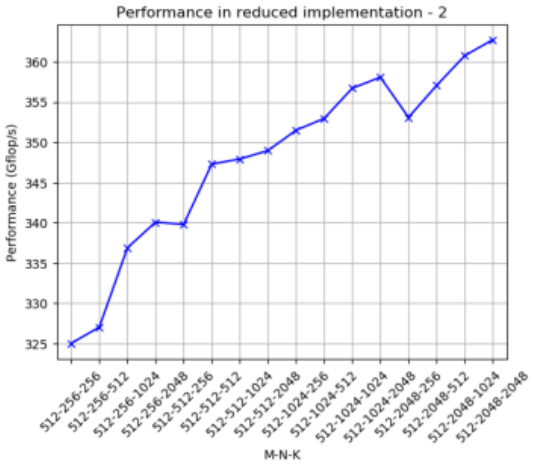
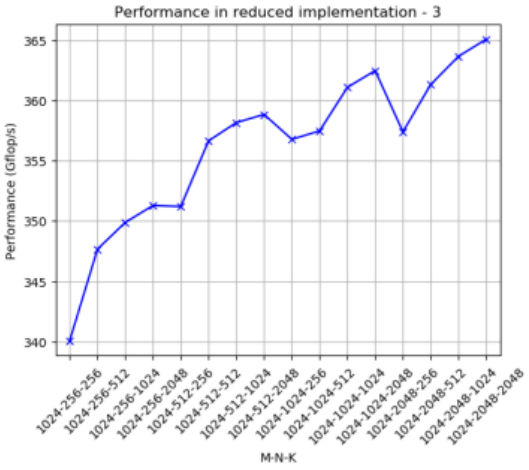
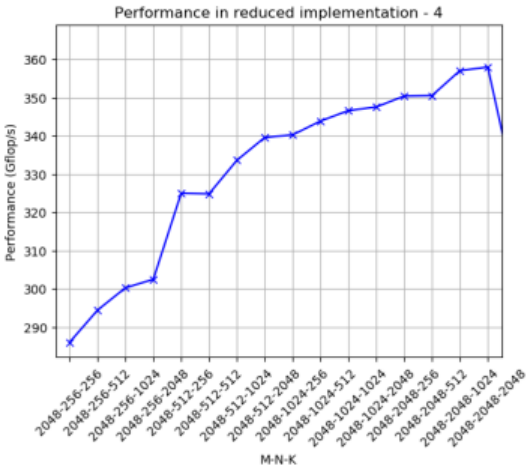
- **Naive**

M = 256	M = 512
	
M = 1024	**M = 2048**
	

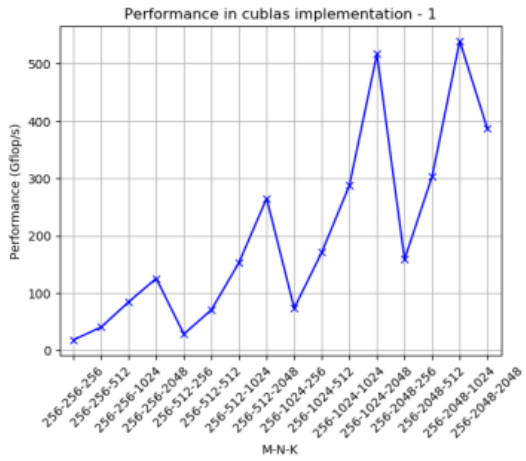
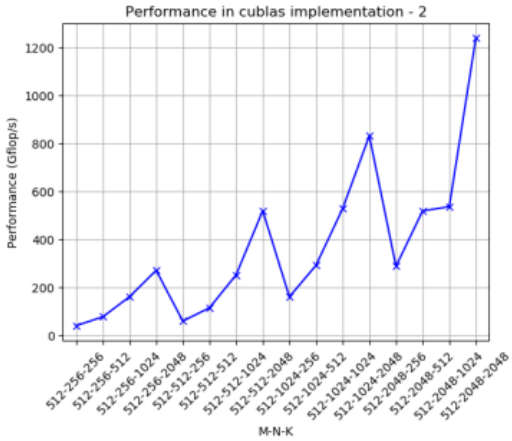
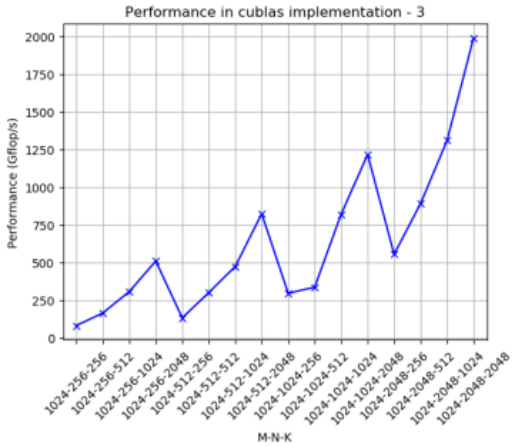
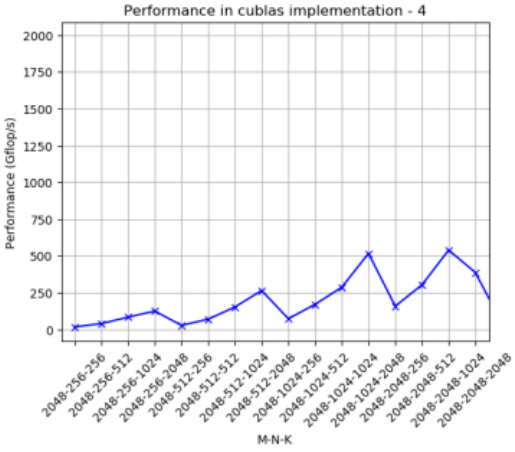
- **coaleshed**

M = 256	M = 512
	
M = 1024	**M = 2048**
	

- **shmem**

M = 256	M = 512
	
M = 1024	**M = 2048**
	

- **cuBLAS**

M = 256	M = 512
	
M = 1024	**M = 2048**
	

Συμπέρασμα

Παρατηρούμε ότι στην naive υλοποίηση έχουμε κάποιες μεγάλες διακυμάνσεις οι οποίες επαναλαμβάνονται. Αυτό οφείλεται στο μέγεθος του K που όσο αυξάνεται αυξάνεται και ο χρόνος προσπέλασης στη κύρια μνήμη. Με τις 2 επόμενες υλοποιήσεις καλύπτουμε αυτό τον χρόνο προφορτώνοντας στοιχεία των πινάκων στην shared memory. Επίσης, όσο αυξάνονται τα μεγέθη M, N, K η επίδοση αυξάνεται καθώς εκμεταλλευόμαστε όλο και περισσότερο την shared memory. Τέλος, όπως ήταν αναμενόμενο το cuBLAS εμφανίζει την καλύτερη επίδοση καθώς θα χρησιμοποιεί πιο προηγμένες τεχνικές.

Ένα τελικό συμπέρασμα είναι πως η υλοποίηση με τη βιβλιοθήκη cuBLAS είναι η βέλτιστη υλοποίηση καθώς ακολουθεί προηγμένες τεχνικές, στη συνέχεια ακολουθεί η shmem, μετά η coaleshed ενώ τελευταία έρχεται η naive, γεγονός που το περιμέναμε εξ' αρχής.