ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

Άσκηση 4: Παράλληλος προγραμματισμός σε επεξεργαστές γραφικών

Χειμερινό εξάμηνο 2019-20 - Ροή Υ

Αντωνιάδης, Παναγιώτης
el15009@central.ntua.gr

Μπαζώτης, Νικόλαος
el15739@central.ntua.gr

"We stand at the threshold of a many core world. The hardware community is ready to cross this threshold. The parallel software community is not."

– Tim Mattson, *principal engineer at Intel*

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:

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

2 Υλοποίηση για επεξεργαστές γραφικών (GPUs)

Στην άσκηση αυτή θα υλοποιήσουμε τον αλγόριθμο DMM για επεξεργαστές γραφικών χρησιμοποιώντας το προγραμματιστικό περιβάλλον CUDA. Ξεκινώντας από μία απλοϊκή αρχική υλοποίηση, στο τέλος της άσκησης θα έχουμε πετύχει μία αρκετά αποδοτική υλοποίηση του αλγορίθμου DMM για τους επεξεργαστές γραφικών. Υποθέτουμε ότι οι πίνακες είναι απόθηκευμένοι στη μνήμη κατά γραμμές (row-major format). Οι υλοποιήσεις λειτουργούν για οποιαδήποτε τιμή των διαστάσεων M, N, K, εφόσον τα αντίστοιχα δεδομένα του προβλήματος χωράνε στην κύρια μνήμη της GPU (global memory).

2.1 Βασική υλοποίηση

Υλοποιήστε τον αλγόριθμο DMM, αναθέτοντας σε κάθε νήμα εκτέλεσης τον υπολογισμό ενός στοιχείου του πίνακα εξόδου.

- **Κώδικας**

```
/*
 *  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;
}
```

- **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.**

  Για τον υπολογισμό του στοιχείου $C_{i,j}$, το αντίστοιχο νήμα θα κάνει $2K$ προσβάσεις στη μνήμη καθώς K φορές προσπελαύνει την μνήμη για να φέρει τα στοιχεία A_{ik} και B_{kj}. Συνολικά, έχουμε $2KMN$ προσβάσεις στη κύρια μνήμη. Ο αριθμός των προσβάσεων είναι ανεξάρτητος από τις διαστάσεις του μπλοκ νήματος γιατί δεν κάνουμε καθόλου χρήση της shared memory που διαθέτουν τα thread blocks.

- **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 μία επανάληψη του εσωτερικού βρόχου έχουμε 2 πράξεις κινητής υποδιαστολής (1 πρόσθεση και 1 πολλαπλασιασμό) και 2 προσβάσεις στην κύρια μνήμη για float αριθμούς (4 byte). Συνεπώς, έχουμε 1 flop ανά 4 bytes από την κύρια μνήμη. Είναι προφανές ότι 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. Για παράδειγμα, στην NVIDIA Tesla K40c που έχει το εργαστήριο έχουμε bandwidth ίσο με 288.4 GB/s και θεωρητική floating point performance 5.046 TFLOPS. Εμείς με 1 flop ανά 4 bytes μπορούμε να φτάσουμε μέχρι τα 72.1 GFLOPS.

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

  Έχουμε υποθέσει ότι οι πίνακες είναι απόθηκευμένοι στη μνήμη κατά γραμμές (row-major format). Συνεπώς, οι προσβάσεις στον πίνακα A συνενώνονται στην κύρια μνήμη καθώς κάθε νήμα προσπελαύνει μία γραμμή του πίνακα. Αντίθετα, οι προσβάσεις στον πίνακα B δεν συνενώνονται στην κύρια μνήμη, καθώς κάθε νήμα προσπελαύνει μία στήλη του πίνακα (μη συνεχόμενες θέσεις μνήμης).

- **Πειραματιστείτε με διάφορα μεγέθη μπλοκ νημάτων και καταγράψετε την χρησιμοποίηση των πολυεπεξεργαστικών στοιχείων με χρήση του CUDA Occupancy Calculator και την επίδοση του κώδικά σας.**

  Στο εργαστήριο υπάρχει εγκατεστημένη κάρτα γραφικών γενιάς 3.5. Παρακάτω βλέπουμε τα αντίστοιχα διαγράμματα για αυτήν την γενιά που μας έβγαλε το CUDA Occupancy Calculator και μας δείχνει την χρησιμοποιήση των πολυεπεξεργαστικών στοιχείων:

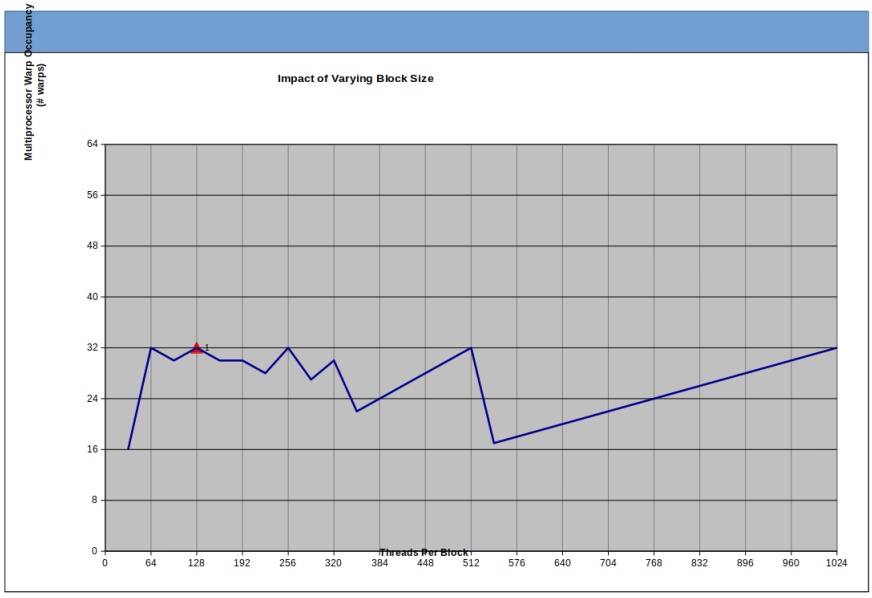


Figure 1: Χρησιμοποιήση με βάση το block size

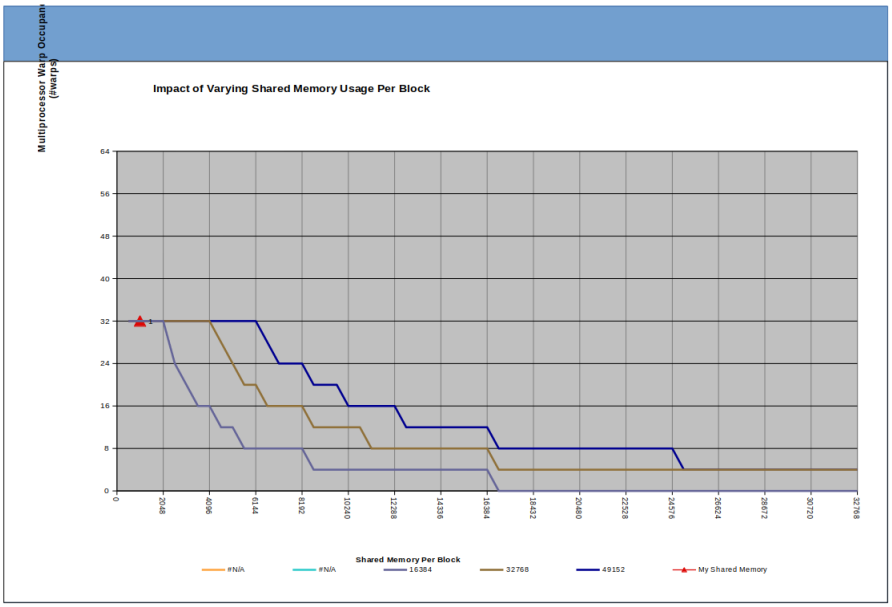


Figure 2: Χρησιμοποιήση με βάση το shared memory size

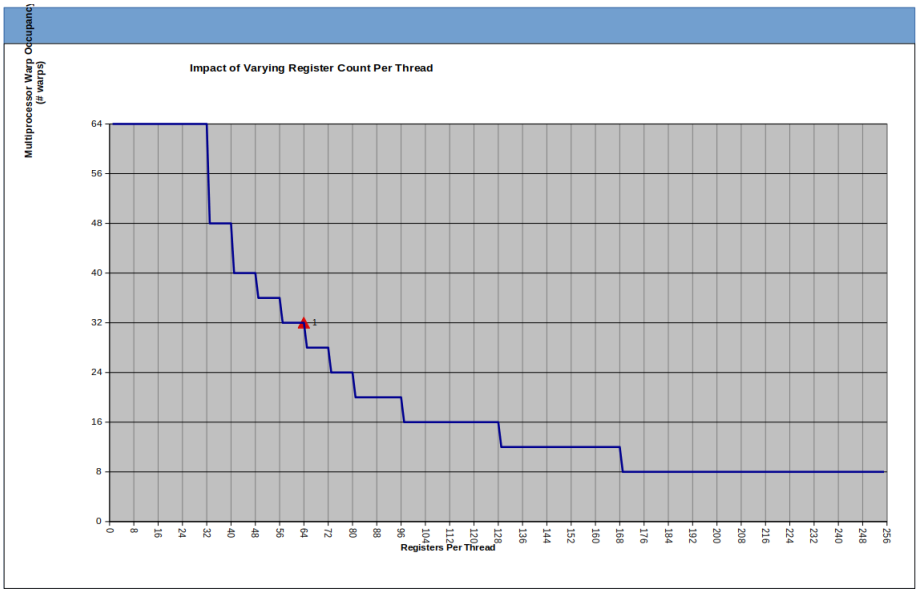


Figure 3: Χρησιμοποιήση με βάση το register count

2.2 Συνένωση των προσβάσεων στην κύρια μνήμη

Τροποποιήστε την προηγούμενη υλοποίηση ώστε να να επιτύχετε συνένωση των προσβάσεων στην κύρια μνήμη για τον πίνακα A προφορτώνοντας τμηματικά στοιχεία του στην τοπική μνήμη των πολυεπεξεργαστικών στοιχείων (shared memory). Μην τροποποιήσετε τον τρόπο ανάθεσης υπολογισμών σε νήματα.

Σημείωση: Για τις επόμενες υλοποιήσεις, υποθέσαμε ότι tile size = thread block size για προγραμματιστική ευκολία.

- **Κώδικας**

```
/*
 *  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) {

  /* 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;
}
```

- **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. Κατά πόσο μειώνονται οι προσβάσεις σε σχέση με την προηγούμενη υλοποίηση;**

  Από εκεί που οι προσβάσεις στην κύρια μνήμη για τον πίνακα A ήταν MNK τώρα μειώθηκαν σε MNK / TILE_SIZE.

- **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. Αντίστοιχα με παραπάνω, τώρα μπορούμε να φτάσουμε μέχρι 144.2 GFLOPS, αλλά πάλι περιοριζόμαστε από την κύρια μνήμη.

2.3 Μείωση των προσβάσεων στην κύρια μνήμη

Αξιοποιήστε την τοπική μνήμη των πολυεπεξεργαστικών στοιχείων (shared memory) για να μειώσετε περαιτέρω τις προσβάσεις στην κύρια μνήμη προφορτώνοντας τμηματικά και στοιχεία του B στην τοπική μνήμη των πολυεπεξεργαστικών στοιχείων (shared memory). Μην τροποποιήσετε τον τρόπο ανάθεσης υπολογισμών σε νήματα.

- **Κώδικας**

```
/*
 *  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;
}
```

- **Υπολογίστε τον αριθμό των προσβάσεων στην κύρια μνήμη για τους πίνακες A και B συναρτήσει των διαστάσεων M, N, K του προβλήματος, των διαστάσεων του μπλοκ νημάτων. Κατά πόσο μειώνονται οι προσβάσεις στην κύρια μνήμη 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.

- **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 SM (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.

2.4 Χρήση της βιβλιοθήκης cuBLAS

Χρησιμοποιήσετε την συνάρτηση cublasSgemm() της βιβλιοθήκης cuBLAS για την υλοποίηση του πολλαπλασιασμού πινάκων. Η βιβλιοθήκη cuBLAS αποτελεί υλοποίηση της BLAS για τις κάρτες γραφικών της NVIDIA. Χρησιμοποιήστε τις κατάλληλες παραμέτρους για τους βαθμωτούς alpha και beta που απαιτεί η συνάρτηση. Επιπλέον, διαβάστε προσεκτικά πως θεωρεί η βιβλιοθήκη cuBLAS ότι είναι αποθηκευμένοι οι πίνακες στη μνήμη για να καθορίσετε την τιμή της παραμέτρου trans.

- **Κώδικας**

```
/*
 *  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);
}
```

Να σημειωθεί, ότι από την στιγμή που η βιβλιοθήκη cuBLAS θεωρεί ότι οι πίνακες είναι αποθηκευμένοι στην μνήμη κατά στήλες, η συνάρτηση δέχεται τον ανάστροφο των 2 πινάκων που δίνουμε. Συνεπώς, για να έχουμε σωστό αποτέλεσμα εκμεταλλευόμαστε την ιδιότητα $C^T = (AB)^T = B^T A^T$ και καλούμε στην dmm_main.cu 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 και μετρήσεις επιδόσεων

Σκοπός των μετρήσεων είναι (α′) η μελέτη της επίδρασης του μεγέθους του μπλοκ νημάτων/υπολογισμού στην επίδοση των υλοποιήσεών και (β′) η σύγκριση της επίδοσης όλων εκδόσεων του πυρήνα DMM. Συγκεκριμένα, έχουμε τα παρακάτω σύνολα μετρήσεων:

- **Για κάθε έκδοση πυρήνα (naive, coalesced, shmem-reduced) θα καταγράψουμε 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.**

  Θα δοκιμάσουμε μόνο τετραγωνικά blocks-tiles με τιμές 4*4, 8*4, 16*16, 32*32 (παραπάνω δεν γίνεται καθώς πρέπει να έχουμε συνολικά μέχρι 1024 threads ανά thread block). Στα παρακάτω διαγράμματα ο οριζόντιος άξονας ειναι η το μέγεθος του block-tile και ο κατακόρυφος τα Gflops.

  – **Naive kernel**

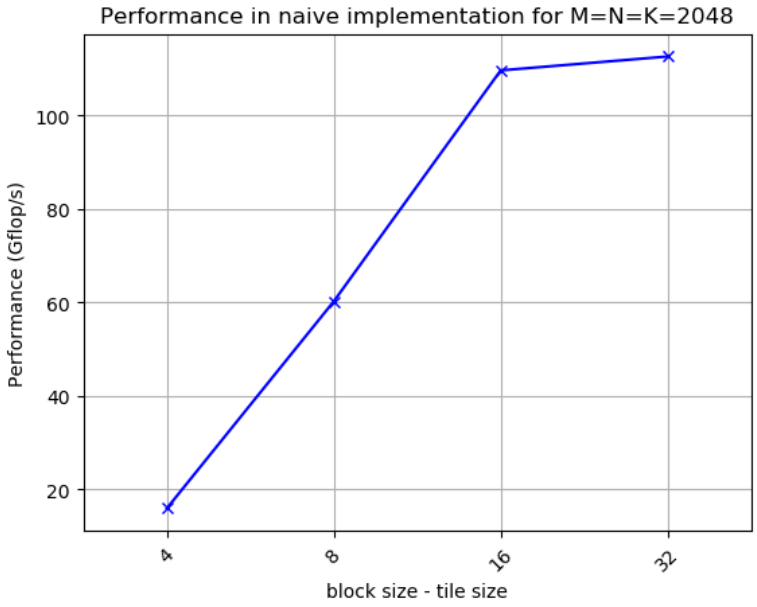


Figure 4: Επίδοση για τον naive πυρήνα καθώς αυξάνεται το block-tile size

  – **Coalesced kernel**

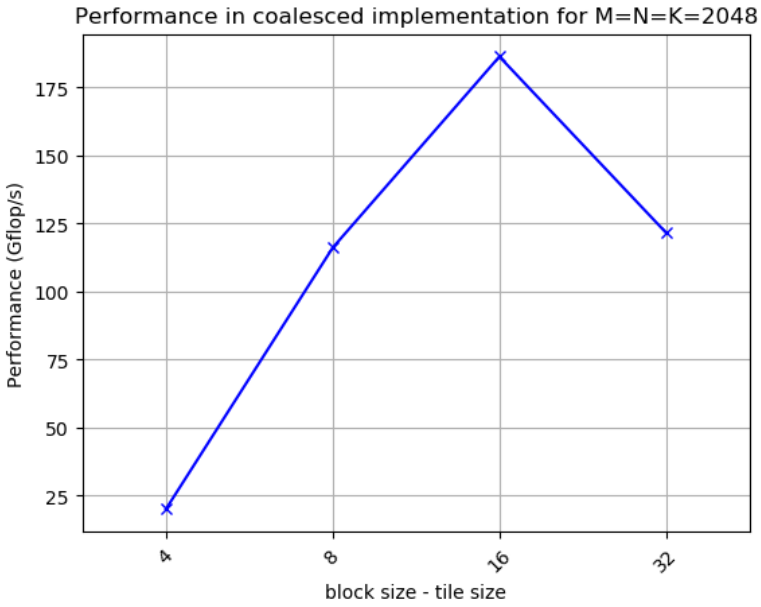


Figure 5: Επίδοση για τον coalesced πυρήνα καθώς αυξάνεται το block-tile size

– **Reduced kernel**

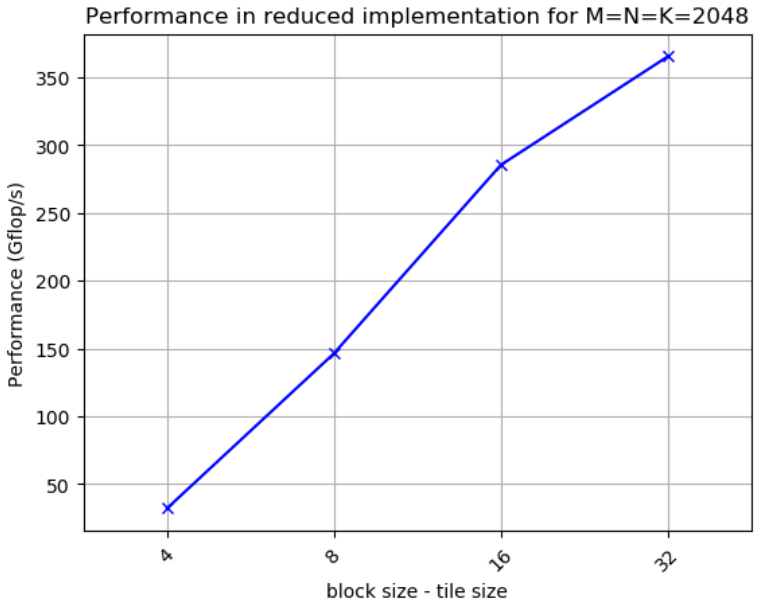
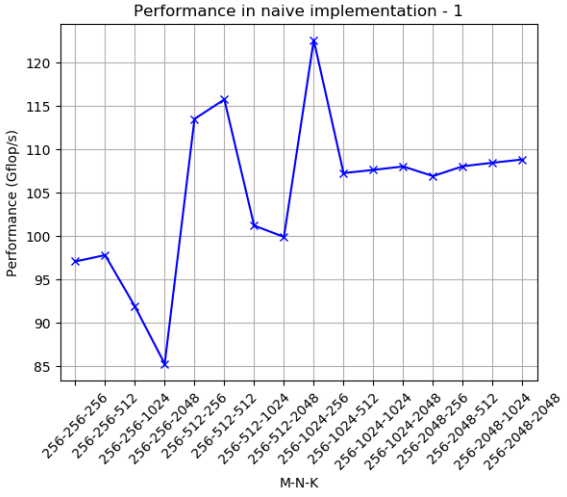


Figure 6: Επίδοση για τον reduced πυρήνα καθώς αυξάνεται το block-tile size

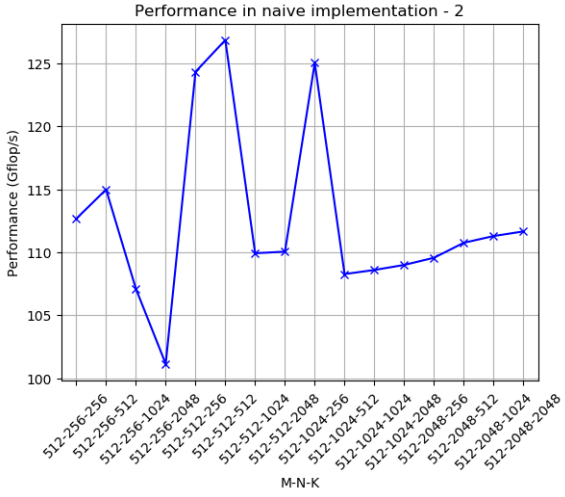
- **Για κάθε μία από τις τέσσερις εκδόσεις πυρήνων (naive, coalesced, shmem και cuBLAS) θα καταγράψουμε την επίδοση για μεγέθη πινάκων $M, N, K \in [256, 512, 1024, 2048]$. Για τις naive, coalesced και shmem υλοποιήσεις επιλέξτε τις βέλτιστες διαστάσεις μπλοκ νημάτων και υπολογισμών.**

  Επειδή όλοι οι πιθανοί συνδυασμοί μεγεθών M, N και K είναι 64 θα χωρίσουμε τα διαγράμματα σε 4 διαγράμματα για κάθε έκδοση πυρήνα, όπως βλέπουμε παρακάτω. Στον οριζόντιο άξονα έχουμε τα μεγέθη M,N,K και στον κάθετο τα Gflops.

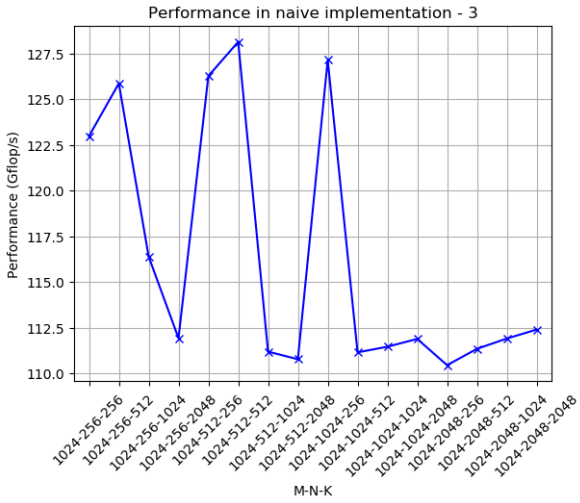
  – **Naive kernel**



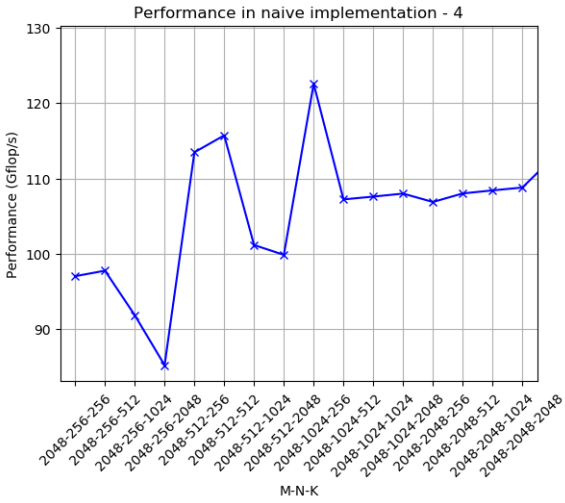
(a) Επίδοση με σταθερό M=256



(b) Επίδοση με σταθερό M=512

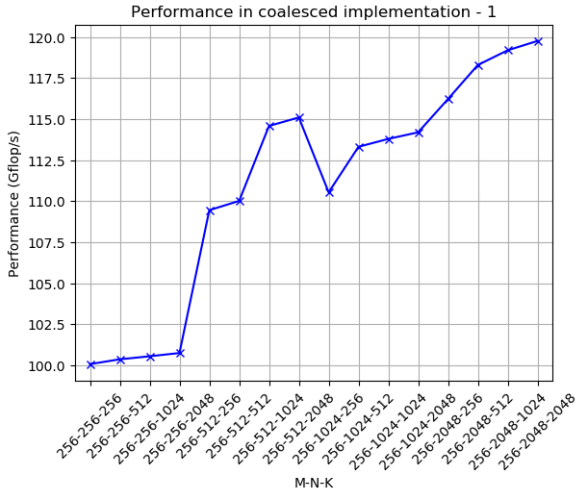


(a) Επίδοση με σταθερό M=1024

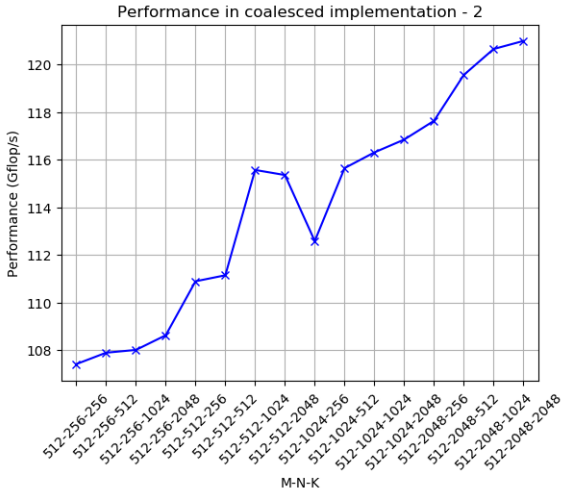


(b) Επίδοση με σταθερό M=2048

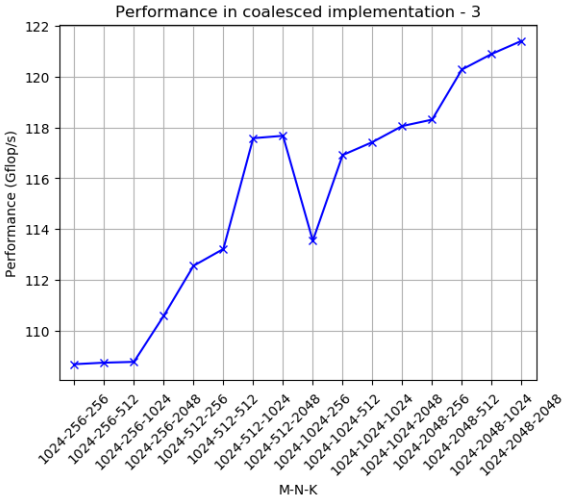
– **Coalesced kernel**



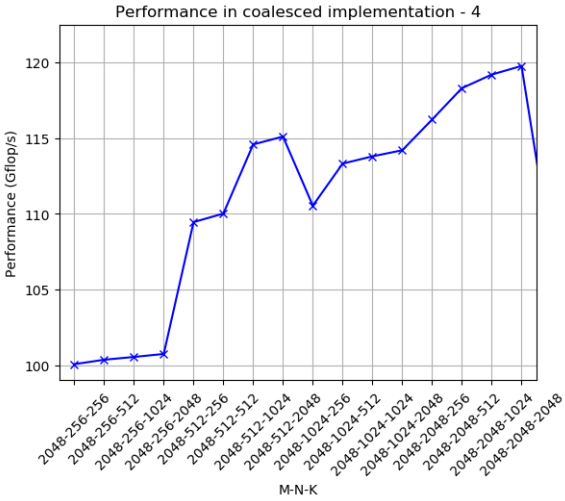
(a) Επίδοση με σταθερό M=256



(b) Επίδοση με σταθερό M=512

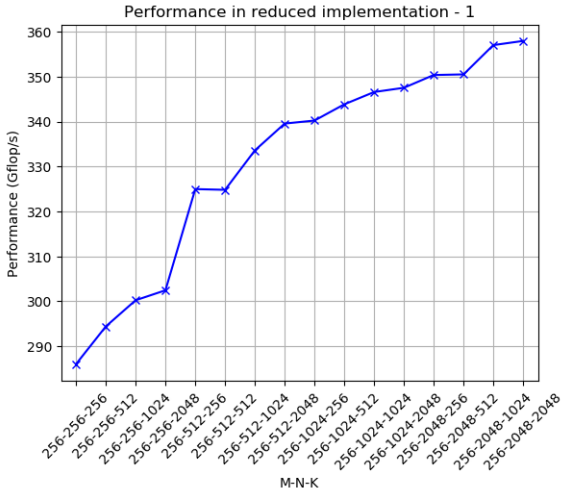


(a) Επίδοση με σταθερό M=1024

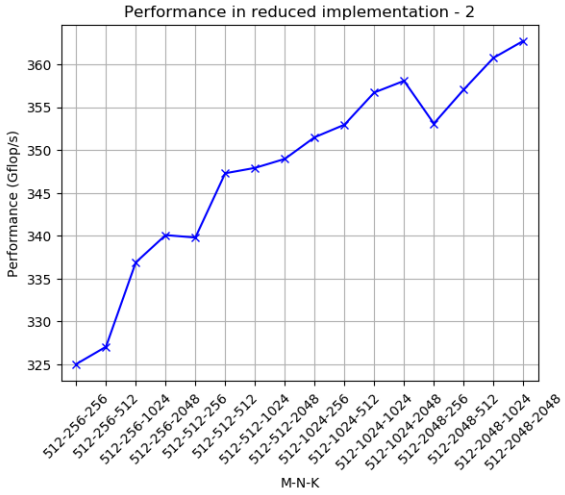


(b) Επίδοση με σταθερό M=2048

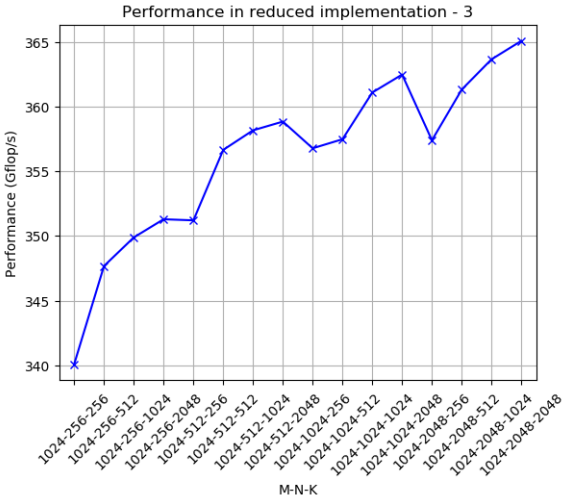
– **Reduced kernel**



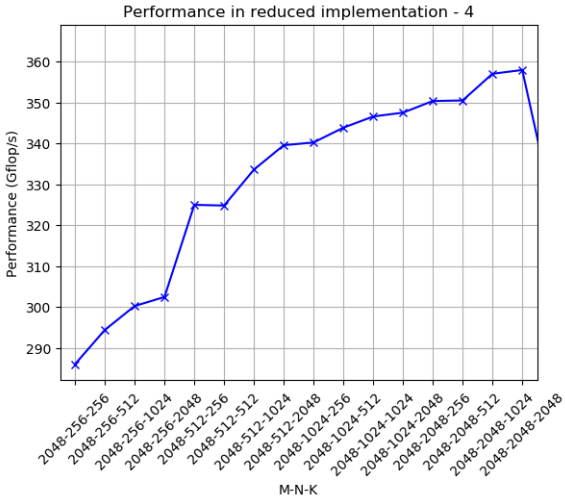
(a) Επίδοση με σταθερό M=256



(b) Επίδοση με σταθερό M=512

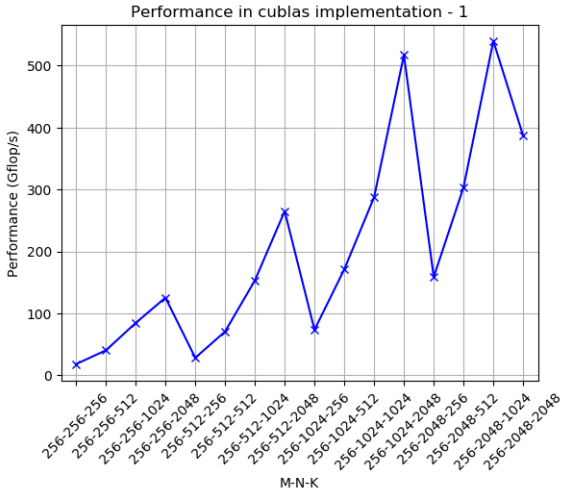


(a) Επίδοση με σταθερό M=1024

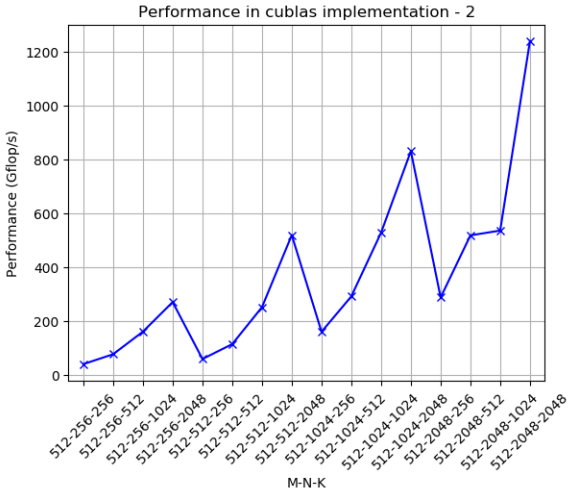


(b) Επίδοση με σταθερό M=2048

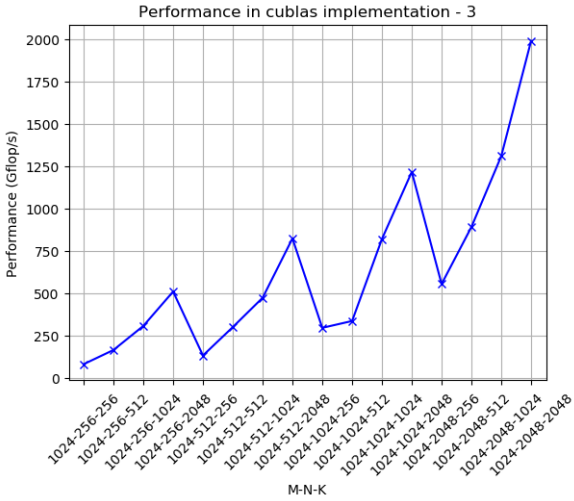
– **cuBLAS kernel**



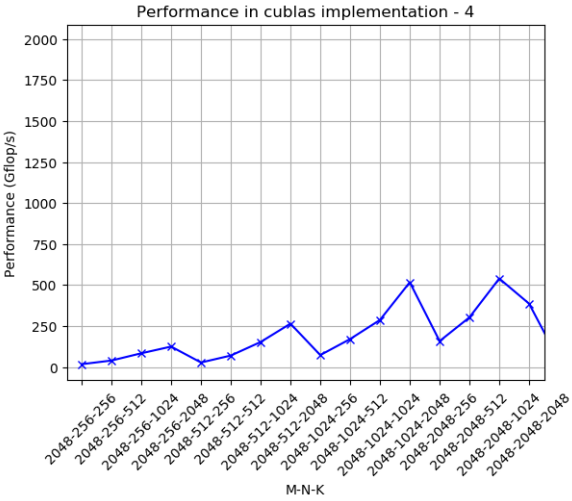
(a) Επίδοση με σταθερό M=256



(b) Επίδοση με σταθερό M=512



(a) Επίδοση με σταθερό M=1024



(b) Επίδοση με σταθερό M=2048

4 Ερμηνεία της συμπεριφοράς και συμπέρασμα

- **Επίδοση για διαφορετικές διαστάσεις του μπλοκ νημάτων**

  Παρατηρούμε ότι για όλες τις εκδόσεις πυρήνες, η επίδοση αυξάνεται καθώς αυξάνεται το μέγεθος του μπλοκ νημάτων - υπολογισμού. Αυτό συμβαίνει γιατί χρησιμοποιώντας όλο και μεγαλύτερα block, έχουμε περισσότερα νήματα που τρέχουν παράλληλα με αποτέλεσμα να κρύβουμε την όποια καθυστέρηση έχουμε για προσβάσεις στη μνήμη. Όσο περισσότερα νήματα έχουμε, τόσο πιο πολύ κρύβουμε το latency στη μνήμη, καθώς κάνουμε παράλληλα κάποια άλλη χρήσιμη λειτουργία. Προφανώς, υπάρχει και ένα όριο στο πόσο μπορούμε να μεγαλώσουμε τα μπλοκ νημάτων (1024 threads), συνεπώς κρατάμε ως βέλτιστο μέγεθος block το 32*32. Επίσης, παρατηρούμε ότι στην coalesced υλοποίηση η αύξηση του μπλοκ νημάτων από 16*16 σε 32*32 μειώνει τη επίδοση του προγράμματος. Αυτό ίσως συμβαίνει διότι η καθυστέρηση που επιβάλλει συγχρονισμός ανάμεσα στα νήματα του thread block δεν καλύπτεται από την επιτάχυνση που παίρνουμε λόγω της χρήσης της shared memory. Τέλος, όπως ήταν αναμενόμενη, καθώς πηγαίνουμε σε όλο και καλύτερη υλοποίηση η αύξηση της επίδοσης είναι μεγαλύτερη λόγω της χρήσης της shared memory.

- **Επίδοση για διαφορετικές διαστάσεις πινάκων εισόδου**

  Παρατηρούμε ότι στην naive υλοποίηση έχουμε κάποιες μεγάλες διακυμάνσης οι οποίες επαναλαμβάνονται. Αυτό συμβαίνει καθώς, όσο αυξάνεται το μέγεθος K που ισούται με το πόσο στοιχεία φορτώνουμε σε κάθε επανάληψη, η επίδοση μειώνεται καθώς ο χρόνος προσπέλασης στη κύρια μνήμη γίνεται όλο και περισσότερος. Σκοπός μας με τις 2 επόμενες υλοποιήσεις (coalesced και reduced) ήταν να καλύψουμε αυτό τον χρόνο προφορτώνοντας στοιχεία των πινάκων στην shared memory. Αν δούμε τις αντίστοιχες γραφικές, παρατηρούμε ότι το πετυχαίνουμε αυτό καθώς η επίδοση δεν επηρεάζεται από την αύξηση του K. 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. Ενδιαφέρον είναι το γεγονός ότι στο cuBlas πάλι έχουμε μία περιοδικότητα όπου (αντίθετα με την naive) καθώς αυξάνεται το K με σταθερά τα M, N η επίδοση αυξάνεται.

- **Συμπέρασμα**

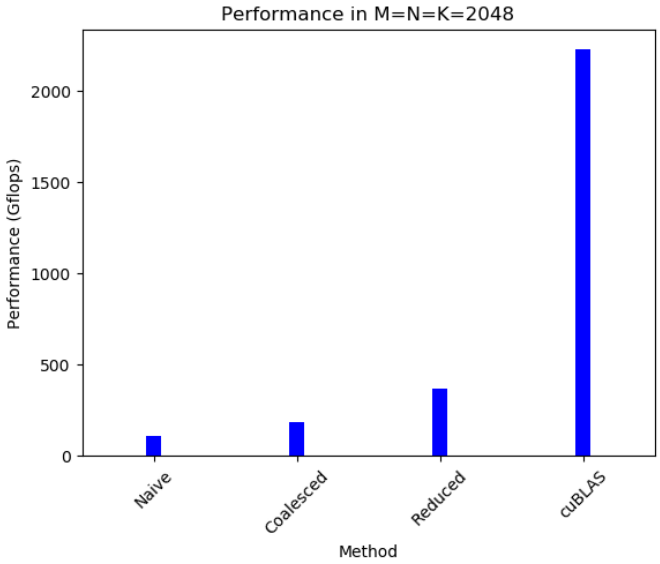


Figure 15: Επίδοση για όλες τις μεθόδους σε πίνακες M=N=K=2048

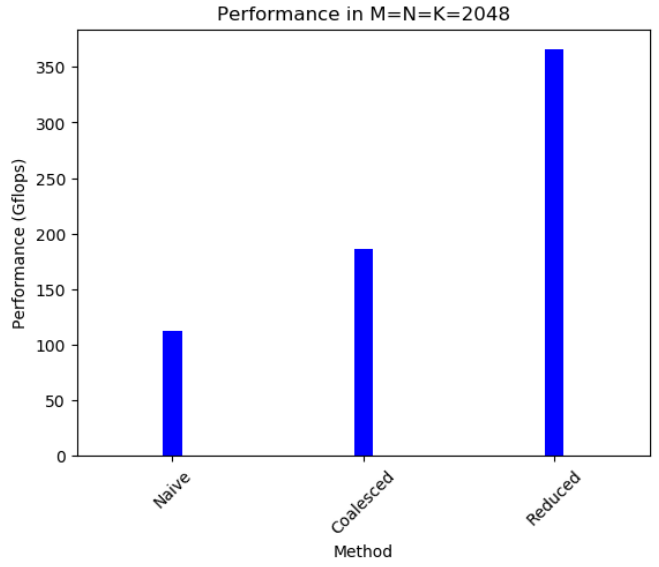


Figure 16: Επίδοση για τις δικές μας υλοποιήσεις σε πίνακες M=N=K=2048

Συμπερασματικά, όπως βλέπουμε και στα παραπάνω διαγράμματα, με τις 2 υλοποιήσεις που εκμεταλλεύονται

την shared memory καταφέραμε να βελτιώσουμε την επίδοση του υπολογιστικού πυρήνα. Ωστόσο, η βιβλιοθήκη cuBLAS έχει πολύ καλύτερη επίδοση, γιατί οι τεχνικές που χρησιμοποιεί είναι πιο προηγμένες.

Αναφορές

[1] "Σημειώσεις του μαθήματος" http://www.cslab.ntua.gr/courses/pps/notes.go

[2] "Cuda Programming-Guide" https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[3] "Nvidia developers" https://developer.nvidia.com/