Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Συστήματα Παράλληλης Επεξεργασίας

4η Εργαστηριακή Άσκηση

9ο Εξάμηνο - Ακαδημαϊκό Έτος 2020-2021

Παγώνης Γεώργιος - 03117030

Γιαννόπουλος Εμμανουήλ - 03117031

18 Ιανουαρίου 2021

1 Naive Kernel

1.1 Κώδικας

```
__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 dot = 0;

    /* If the thread's position is out of the array, it remains inactive */
    while (row < M && col < N){
        /* Compute the value of C */
        for (int k = 0; k < K; k++){
            dot += A[row*K+k]*B[col+k*N];
        }
        C[row*N+col]=dot;
        row += blockDim.y*gridDim.y;
        col += blockDim.x*gridDim.x;
    }

}
```

1.2 Ερωτήσεις

- 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 ενός στοιχείου εξόδου, το κάθε νήμα χρειάζεται $2K$ προσβάσεις στη μνήμη, για να φέρει μια σειρά του και μια στήλη του B. Επομένως έχουμε $2KMN$ προσβάσεις συνολικά.

- 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 naive υλοποίησης έχουμε 2 flops (1 πρόσθεση, 1 πολλαπλασιασμό) ανά 2 προσβάσεις στη μνήμη (ένα στοιχείο του και ένα του B). Επομένως 2 flops ανά 8 bytes ή 0.25 $\frac{flops}{byte}$. Συμφωνά με το επίσημο manual της NVIDIA, η GPU NVIDIA Tesla K40c

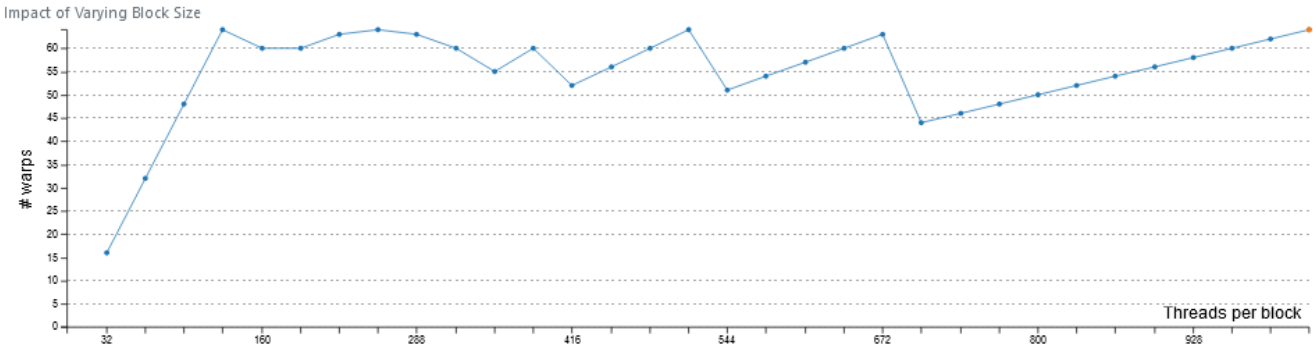
έχει θεωρητική επίδοση 5.046 TFLOPS αλλά bandwidth 288.4 GB/s. Επομένως σε αυτή την υλοποίηση έχουμε άνω όριο τα 72.1 GFLOPS και είναι memory-bound.

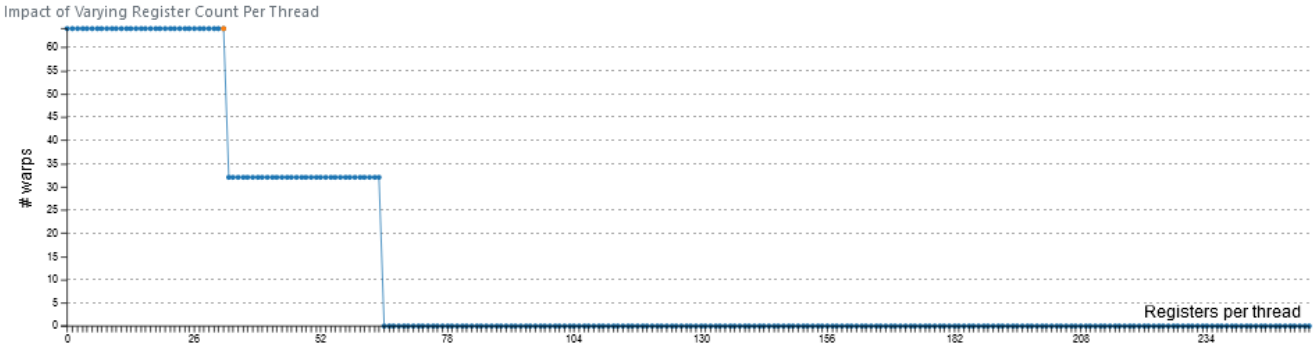
- Ποιες από τις προσβάσεις στην κύρια μνήμη συνενώνονται και ποιες όχι με βάση την υλοποίησή σας;

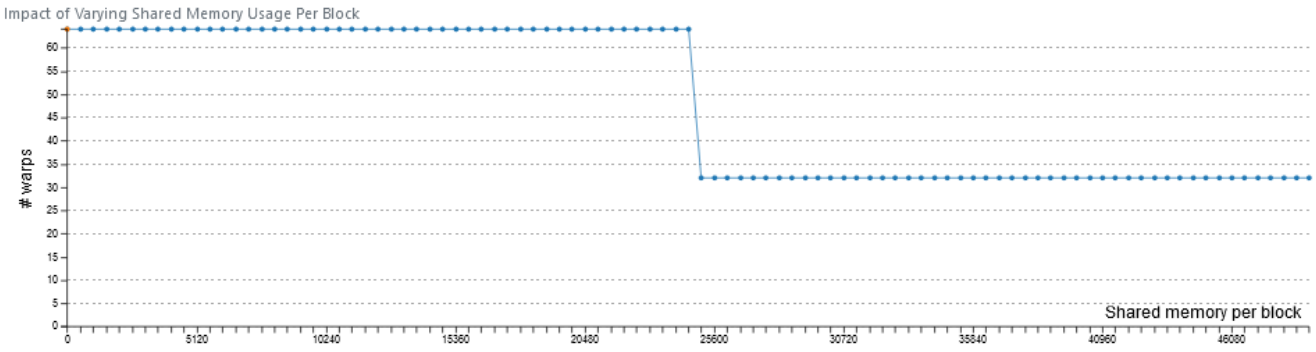
  Λόγω του τρόπου που αποθηκεύονται οι πίνακες στη μνήμη, συνενώνονται οι προσβάσεις στη μνήμη του πίνακα για κάθε νήμα **ξεχωριστά**. Όλες οι υπόλοιπες προσβάσεις δεν συνενώνονται.

- 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.

  Λόγω του μεγέθους των εικόνων, παραθέτουμε μόνο το καλύτερο μέγεθος μπλοκ νημάτων. Όπως βλέπουμε, η επίδοση μεγιστοποιείται για 1024 threads per block, που είναι ο μέγιστος δυνατός αριθμός.







2 Coalesced Kernel

2.1 Κώδικας

```
__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 dot = 0;

    for(int m = 0; m < K; m+=TILE_X){
        /* Load the current tile in the shared memory and synchronize */
        A_shared[ty*TILE_X+tx] = A[row*K + m+tx];

        __syncthreads();

        for(int k = 0; k < TILE_X; k++){
            /* Compute the inner product of current tile and synchronize */
            // This has to be A goes from tile to tile
            // But B traverse the hole block so is the same with
            // the naive but you have to change from B[k*N + col]
            // to B[(m+k)*N+col] because here k is only in the tile
            dot += A_shared[ty*TILE_X+k]*B[(m+k)*N+col];
        }
        __syncthreads();
    }
    /* Save result */
    C[row*N+col] = dot;

}
```

2.2 Ερωτήσεις

- 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 και των διαστάσεων του μπλοκ υπολογισμού. Κατά πόσο μειώνονται οι προσβάσεις σε σχέση με την προηγούμενη υλοποίηση;

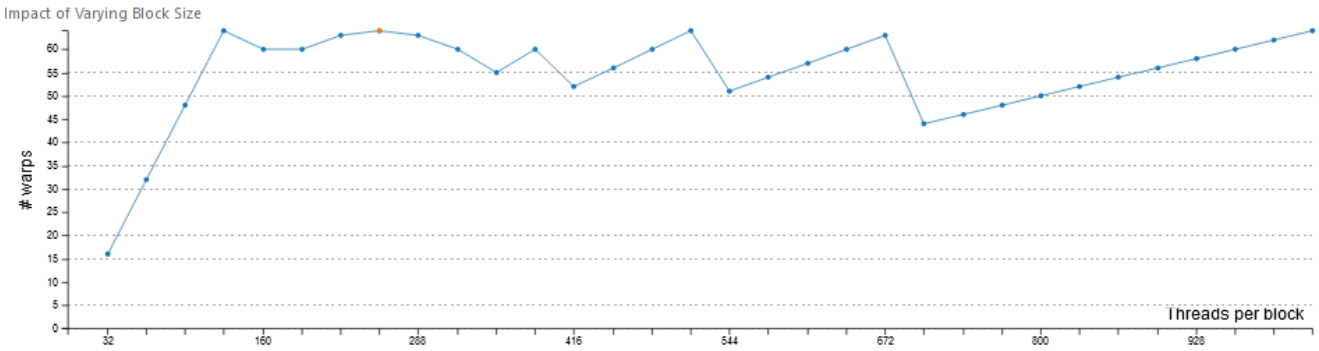
Σε αυτή την υλοποίηση οι προσβάσεις στον πίνακα A ομαδοποιούνται κατά TILE_X×TILE_Y. Επομένως θα έχουμε για τον πίνακα A, $\frac{MNK}{\text{TILE\_X}\times\text{TILE\_Y}}$ προσβάσεις. Οι συνολικές προσβάσεις μειώνονται κατά $MNK - \frac{MNK}{\text{TILE\_X}\times\text{TILE\_Y}}$.

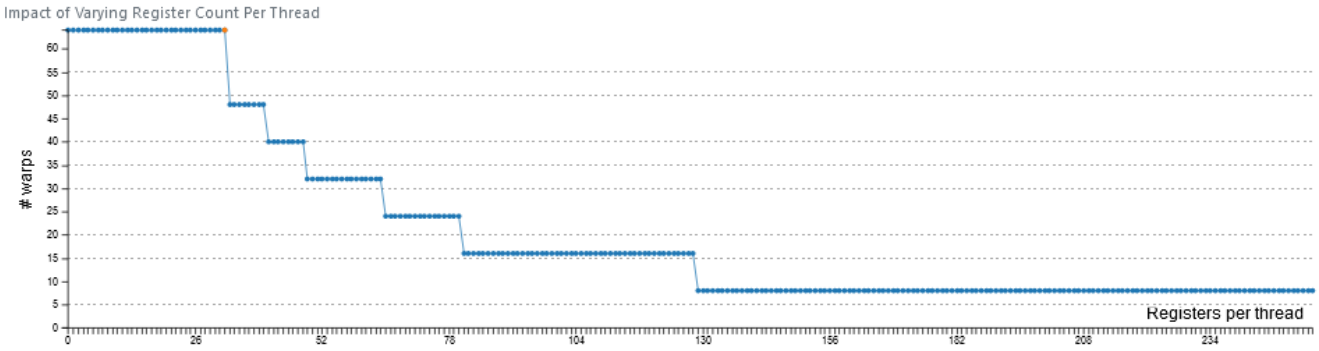
- Υπολογίστε το πηλίκο των πράξεων κινητής υποδιαστολής (floating-point operations) προς τον αριθμό των προσβάσεων στην κύρια μνήμη μίας επανάληψης του εσωτερικού βρόχου της υλοποίησής σας. Η επίδοση της υλοποίησής σας περιορίζεται από το εύρος ζώνης της κύριας μνήμης (memory-bound) ή το ρυθμό εκτέλεσης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

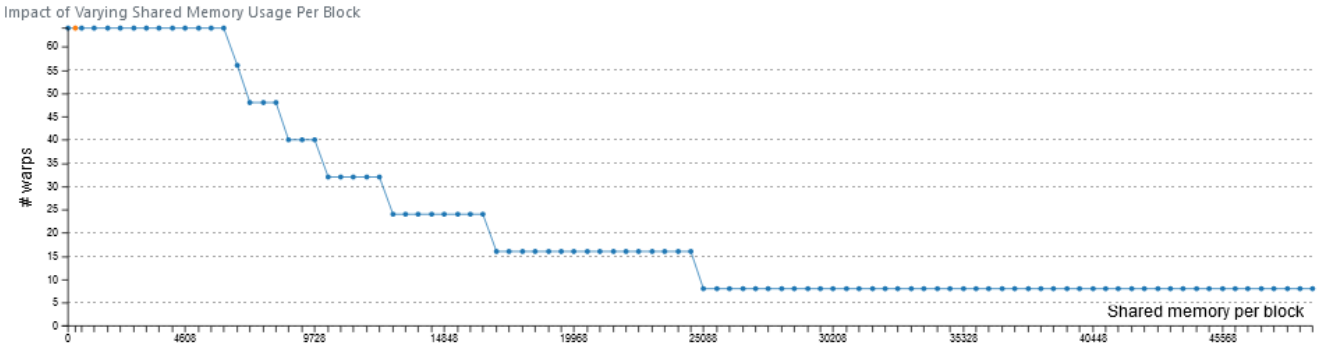
  Στον εσωτερικό βρόχο της naive υλοποίησης έχουμε 2 flops (1 πρόσθεση, 1 πολλαπλασιασμό) ανά 1 προσβάσεις στη μνήμη (ένα στοιχείο του B). Επομένως 2 flops ανά 4 bytes ή 0.5 $\frac{flops}{byte}$. Επομένως σε αυτή την υλοποίηση έχουμε άνω όριο τα 144.2 GFLOPS και η υλοποίηση είναι memory-bound.

- 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.

  Λόγω του μεγέθους των εικόνων, παραθέτουμε μόνο το καλύτερο μέγεθος μπλοκ νημάτων. Όπως βλέπουμε, η επίδοση μεγιστοποιείται για 256 threads per block, δηλαδή για ένα μπλοκ 16×16 όπως θα δούμε και παρακάτω.







3 Shmem Kernel

3.1 Κώδικας

```
__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;
```

```
    int row = by * TILE_Y + ty;
    int col = bx * TILE_X + tx;

    value_t dot = 0;

    for(int m = 0; m < K; m+=TILE_X){
        A_shared[ty*TILE_X+tx] = A[row*K + m+tx];
        // If TILE.x != TILE.y we have to do this differently
        // I dont think this can happen because we add more
        // complexity than nessesary. Either way every element will
        // be excecuted in 32 wraps so no need for extra thinking.
        B_shared[ty*TILE_X+tx] = B[col + (m+ty)*N];

        __syncthreads();

        for(int k = 0; k < TILE_X; k++){
            dot += A_shared[ty*TILE_X+k]*B_shared[k*TILE_X+tx];
        }
        __syncthreads();
    }
    C[row*N+col] = dot;

}
```

3.2 Ερωτήσεις

- 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 και των διαστάσεων του μπλοκ υπολογισμού. Κατά πόσο μειώνονται οι προσβάσεις σε σχέση με την προηγούμενη υλοποίηση;

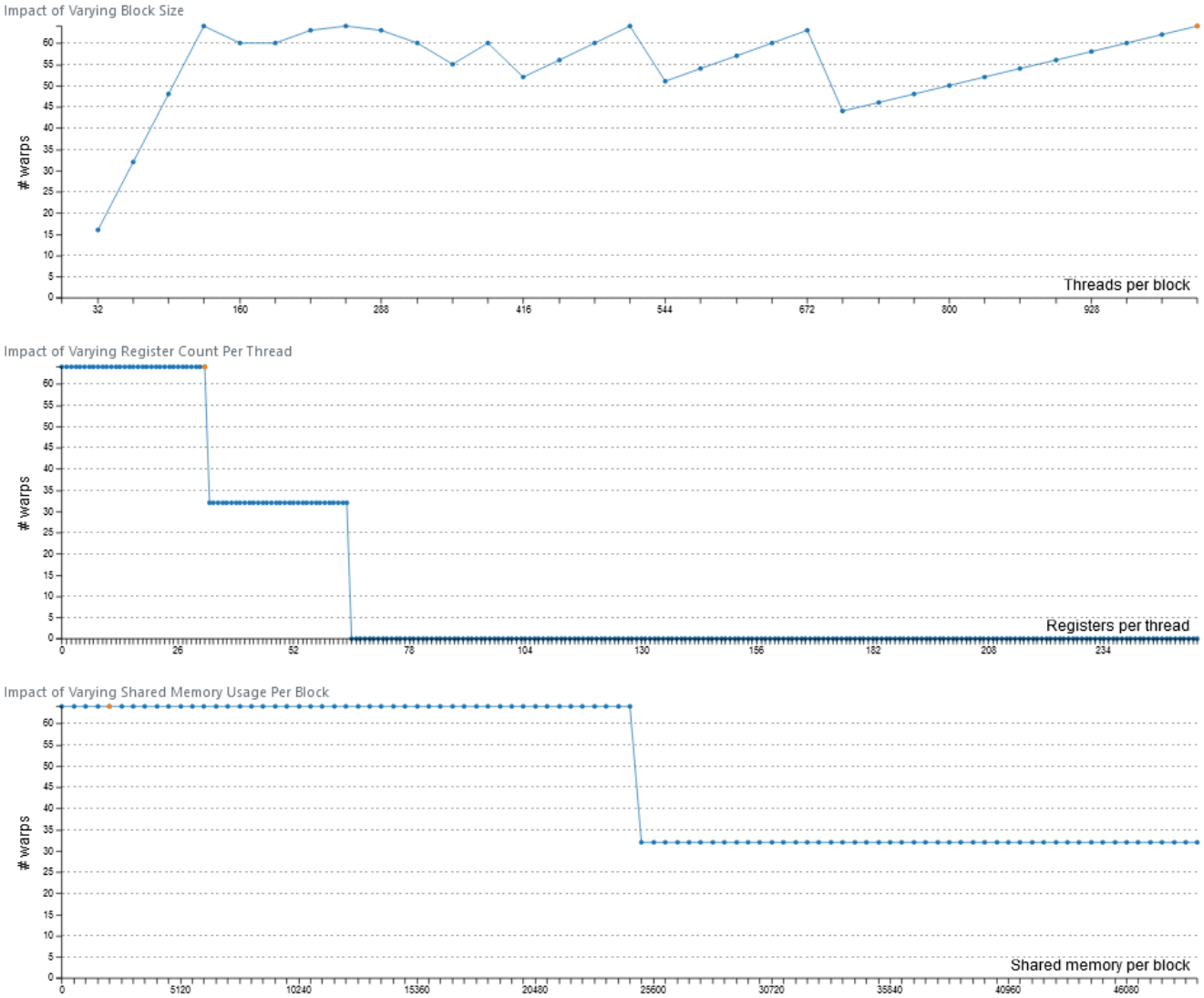
  Σε αυτή την υλοποίηση όλες οι προσβάσεις ομαδοποιούνται κατά TILE_X $\times$ TILE_Y. Επομένως θα έχουμε $\frac{2MNK}{\text{TILE\_X}\times\text{TILE\_Y}}$ προσβάσεις. Οι συνολικές προσβάσεις μειώνονται κατά $MNK - \frac{MNK}{\text{TILE\_X}\times\text{TILE\_Y}}$ σε σχέση με το Coalesced Kernel.

- 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 naive υλοποίησης έχουμε μόνο flops και καμία πρόσβαση στη μνήμη. Επομένως θεωρητικά $\infty \frac{flops}{byte}$. Επομένως σε αυτή την υλοποίηση έχουμε (θεωρητικά) άνω όριο τα 5.046 TFLOPS αν μπορούσαμε κάπως να προφορτώσουμε όλους τους πίνακες στη μνήμη της GPU και η υλοποίηση είναι compute-bound.

- 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.

Λόγω του μεγέθους των εικόνων, παραθέτουμε μόνο το καλύτερο μέγεθος μπλοκ νημάτων. Όπως βλέπουμε, η επίδοση μεγιστοποιείται για 1024 threads per block, που είναι ο μέγιστος δυνατός αριθμός.



4 cuBLAS

4.1 Κώδικας

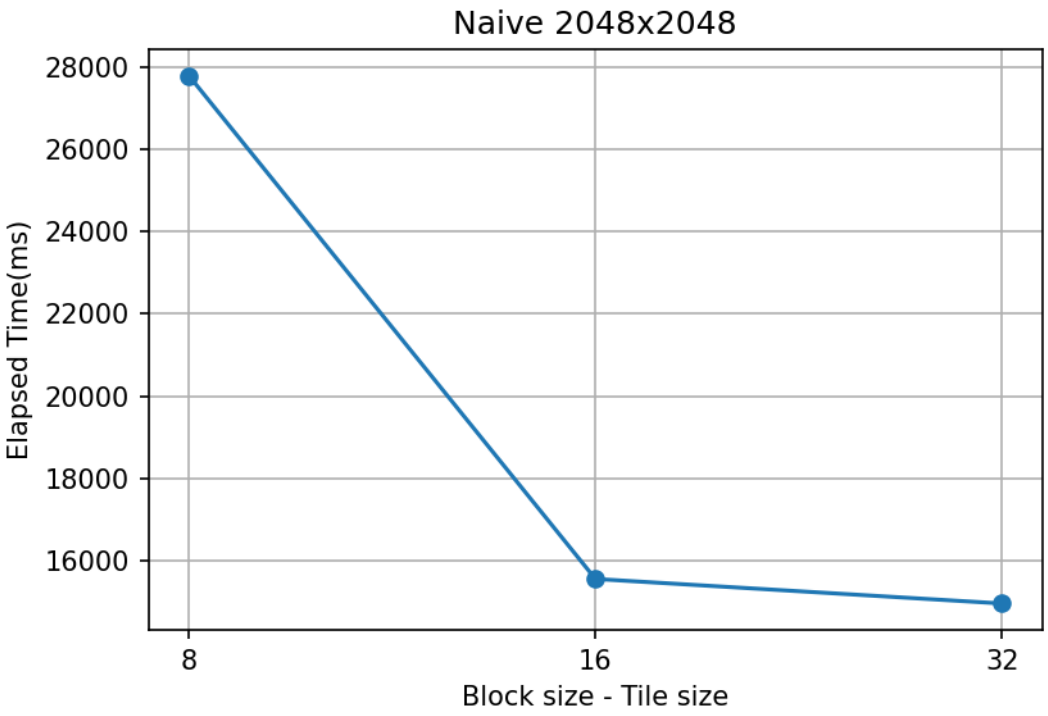
```
void dmm_gpu_cublas(const value_t *A, const value_t *B, value_t *C,
        const size_t M, const size_t N, const size_t K) {

    /* Define parameters for cublasSgemm */

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

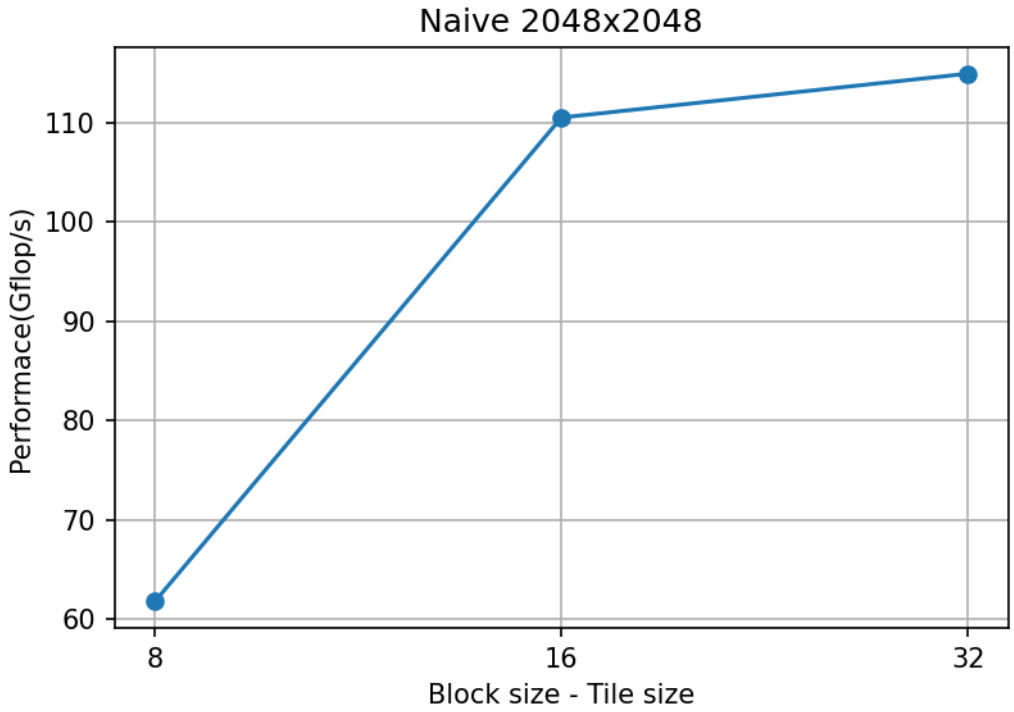
```
    const float alph = 1;
    const float bet = 0;
    const float *alpha = &alph;
    const float *beta = &bet;

    /* 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);
}
```

5 Πειράματα

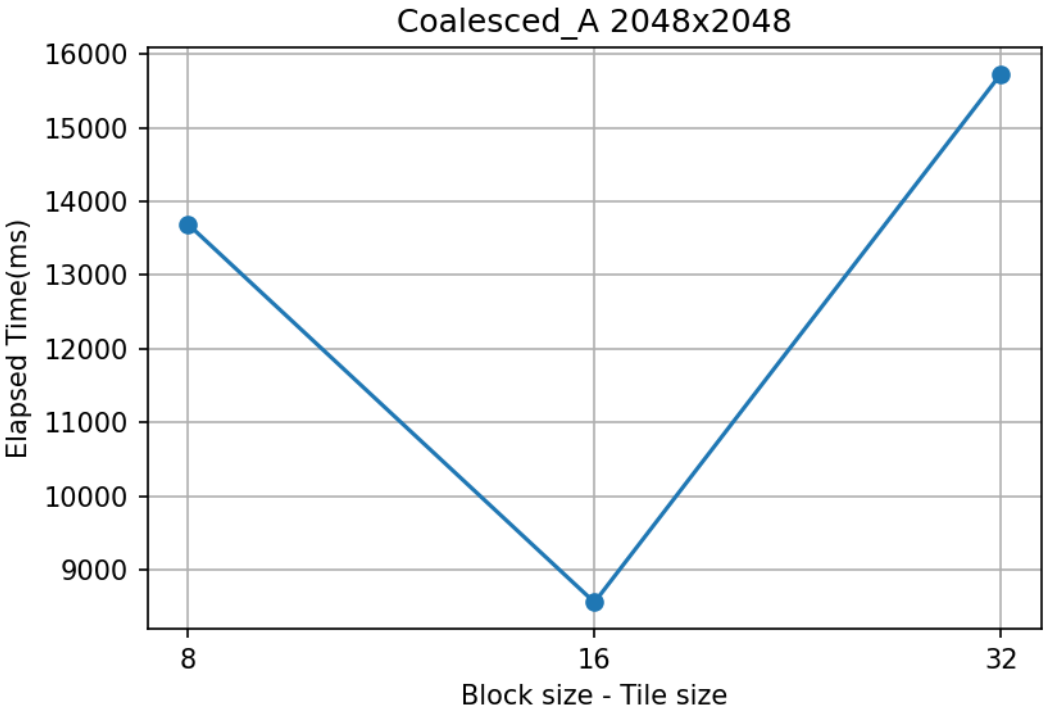
5.1 Μετρήσεις για διαφορετικά block sizes

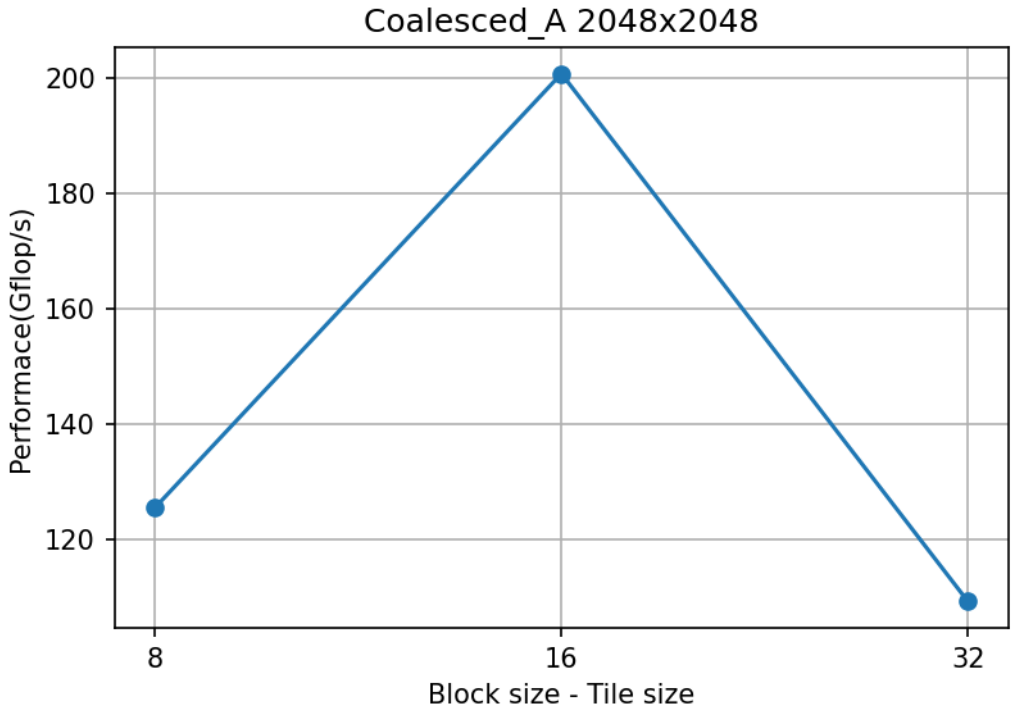
5.1.1 Naive kernel



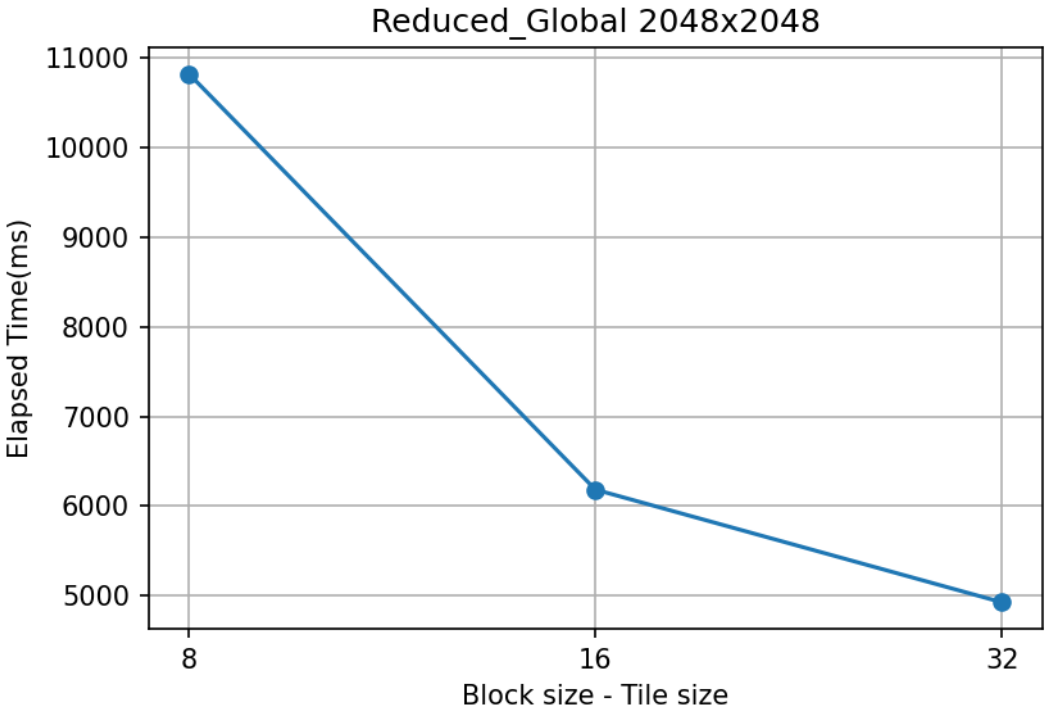


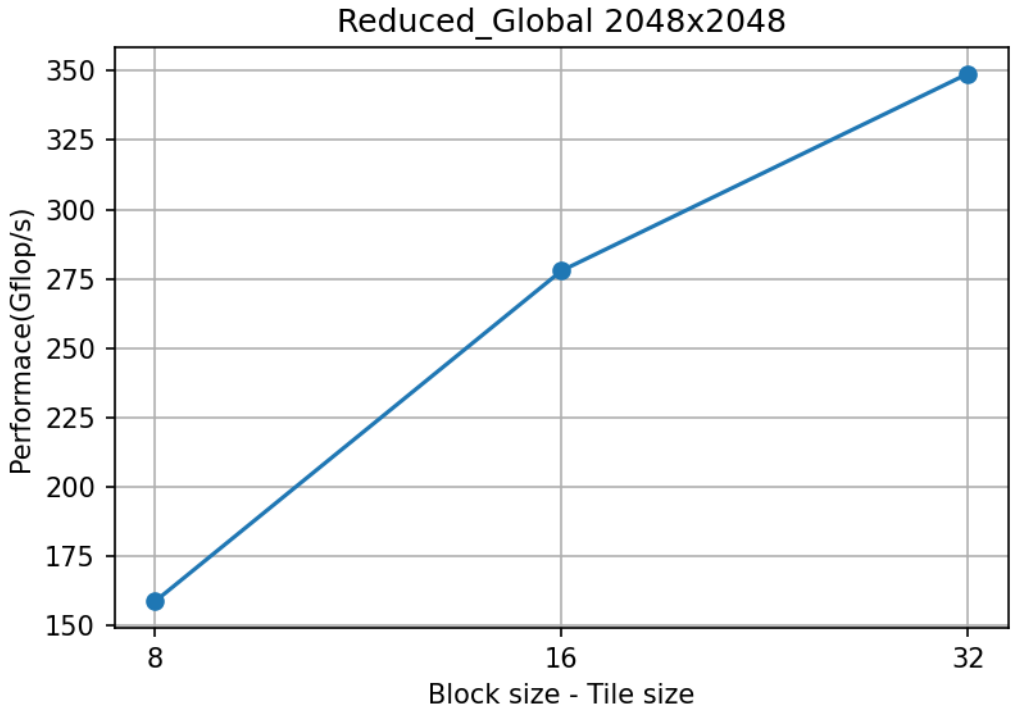
5.1.2 Coalesced kernel





5.1.3 Shmem kernel





Παρατηρήσεις:

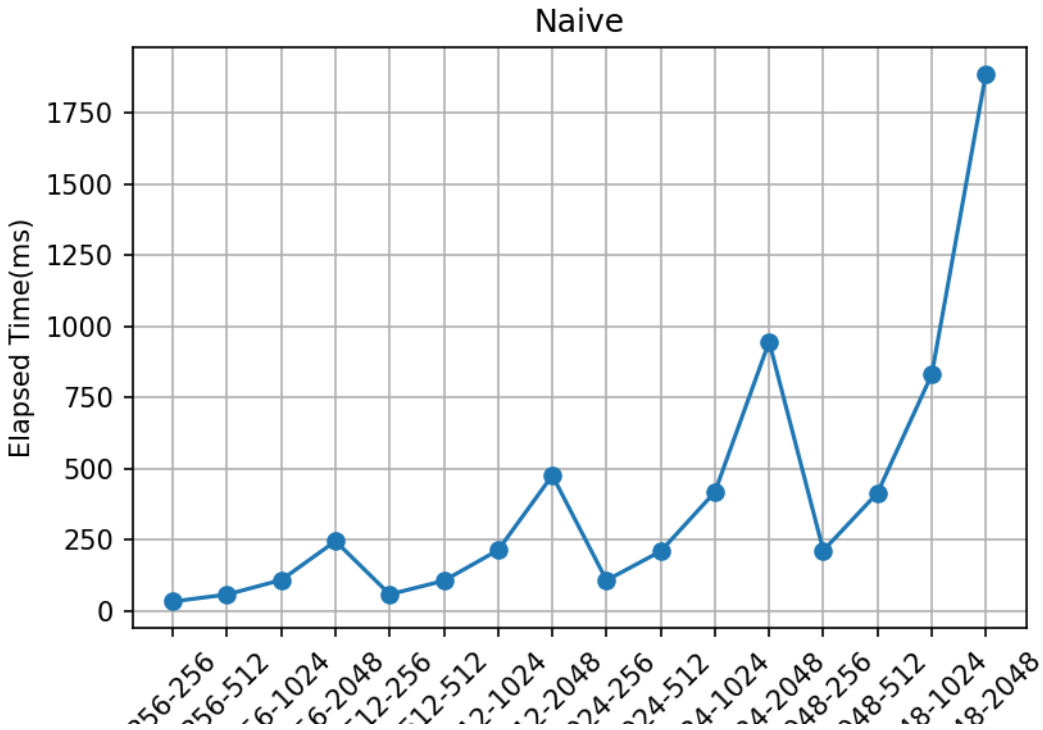
Στις υλοποιήσεις naive και shared memory, η επίδοση αυξάνεται όσο αυξάνεται και το block size. Αντίθετα στην coalesced το βέλτιστο block size είναι το 16×16 αποτέλεσμα που συμπίπτει και με το output του CUDA Occupancy Calculator. Αυτό μπορεί να οφείλεται στην υλοποίησή μας και στα στοιχεία της αρχιτεκτονικής της GPU.

5.2 Μετρήσεις για διαφορετικά μεγέθη πινάκων

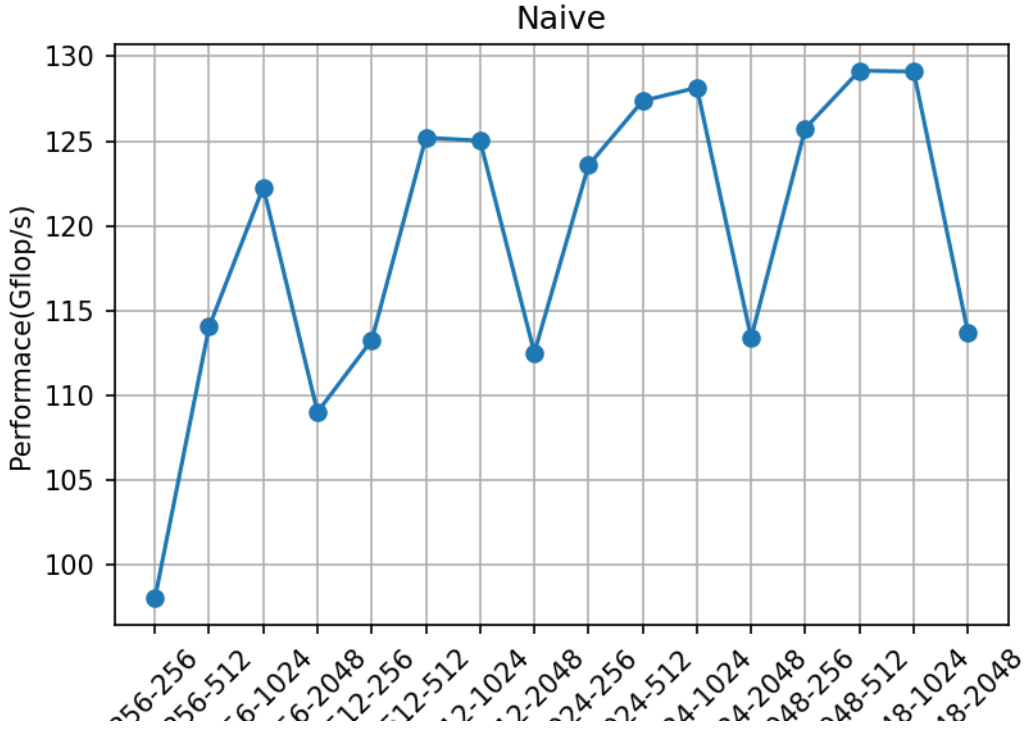
Για αυτές τις μετρήσεις, στα διαγράμματα κρατάμε σταθερή την εσωτερική διάσταση K και παρουσιάζουμε το scaling με βάση τις διαστάσεις του πίνακα εξόδου C.

5.2.1 Naive kernel

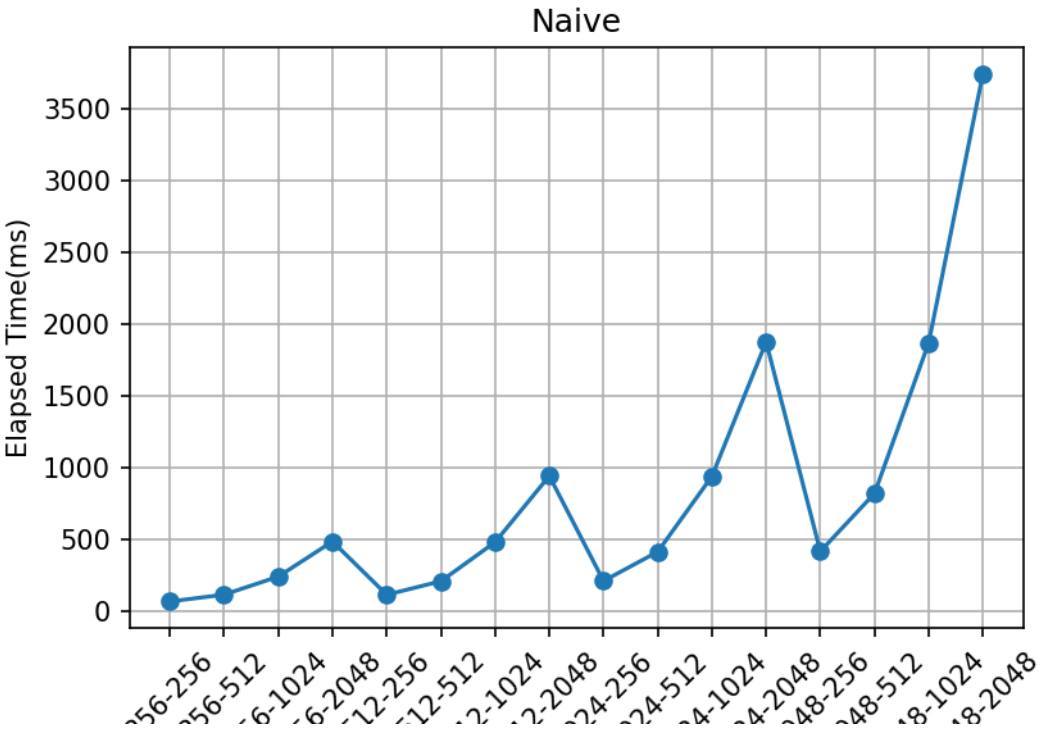
Σχήμα 1: $K = 256$



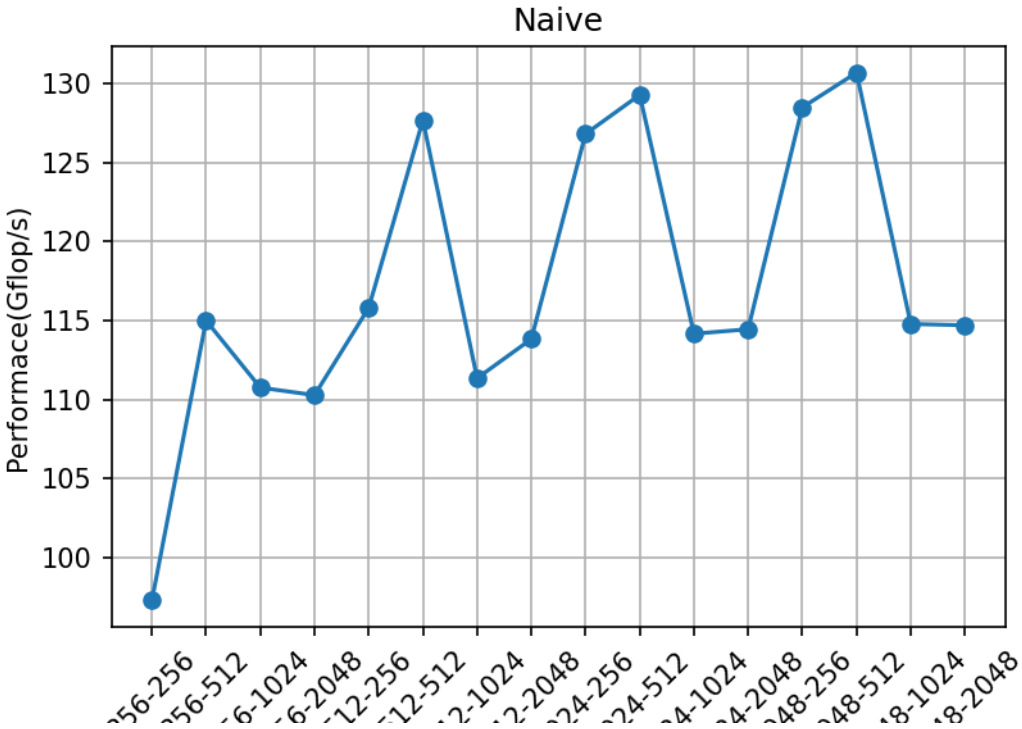
Σχήμα 2: $K = 256$



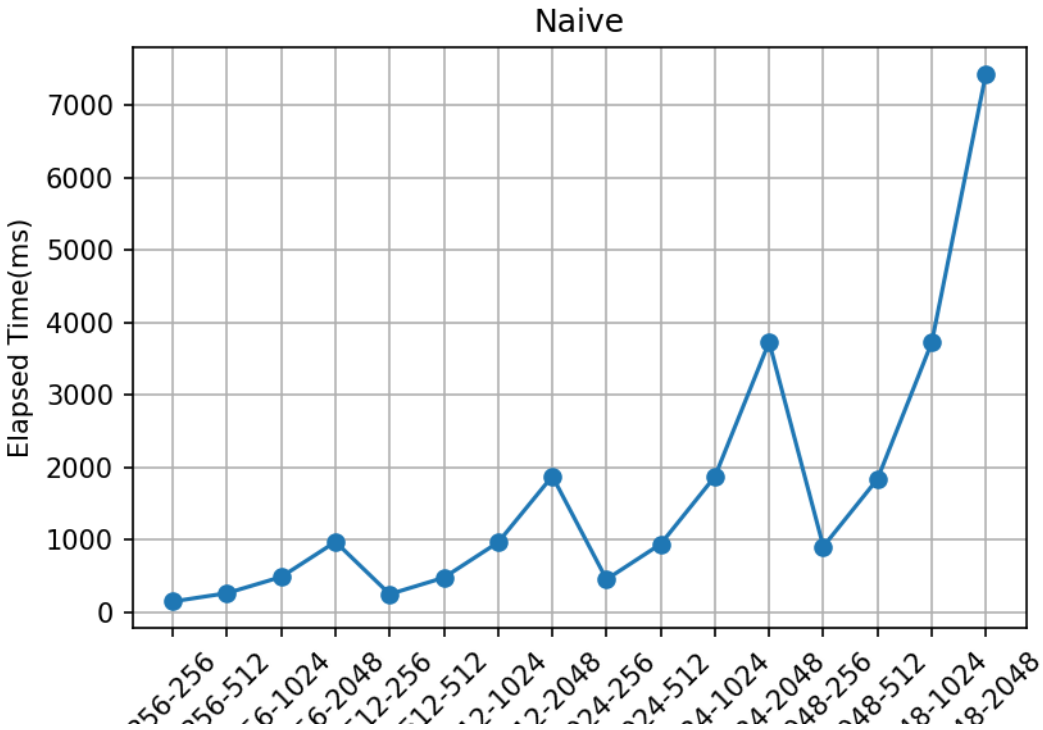
Σχήμα 3: $K = 512$



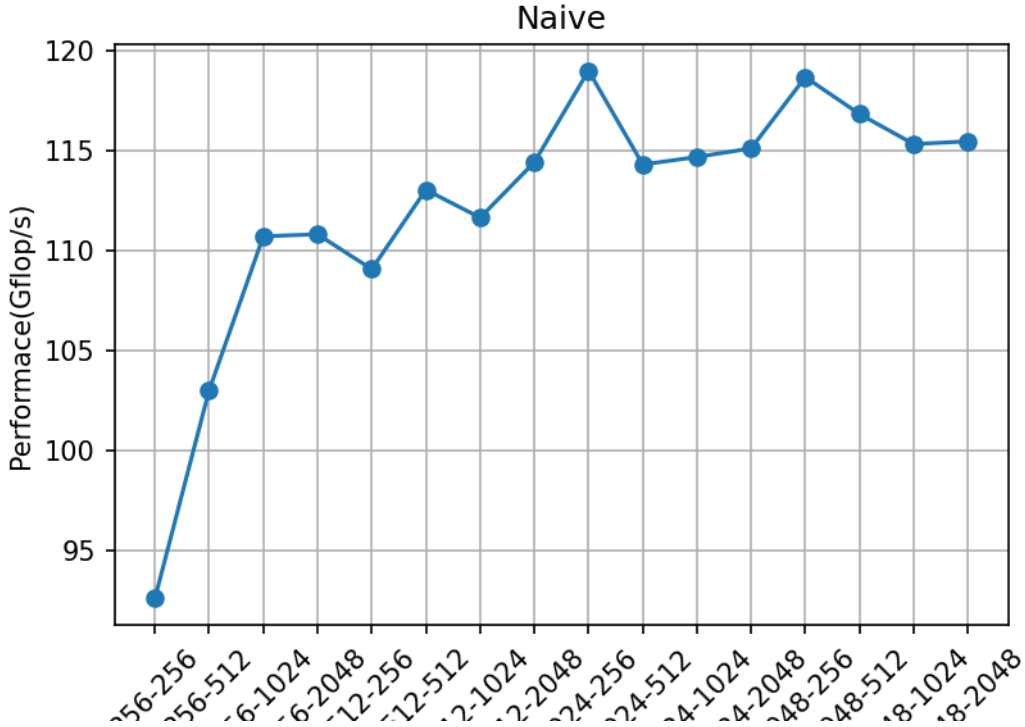
Σχήμα 4: $K = 512$



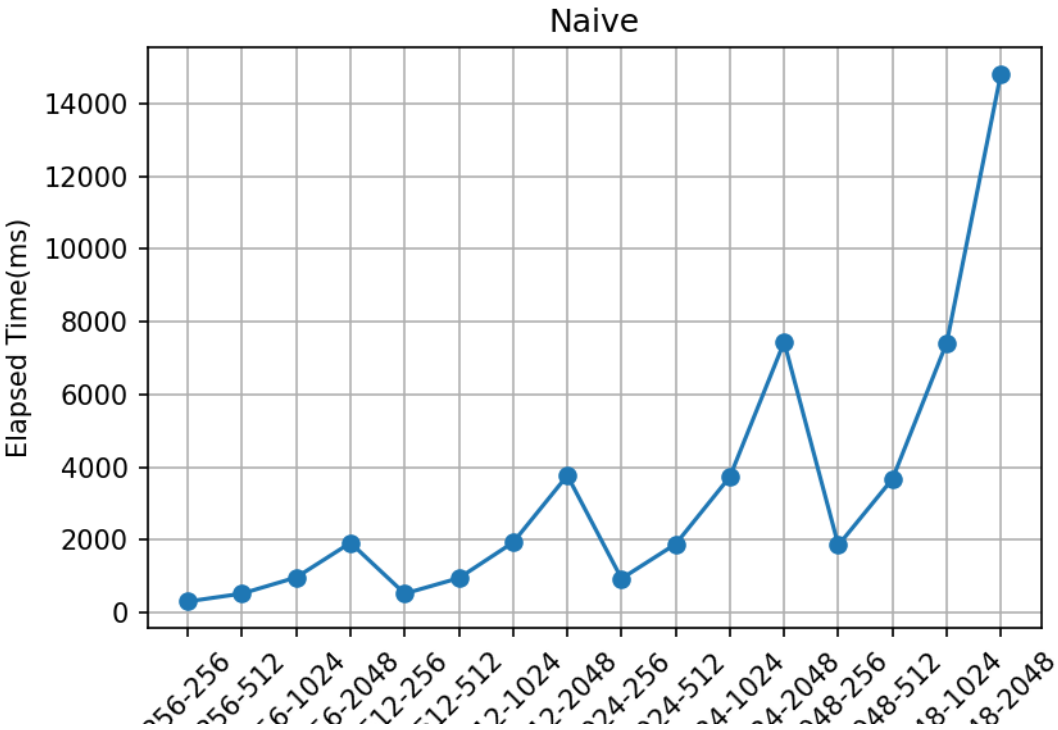
Σχήμα 5: $K = 1024$



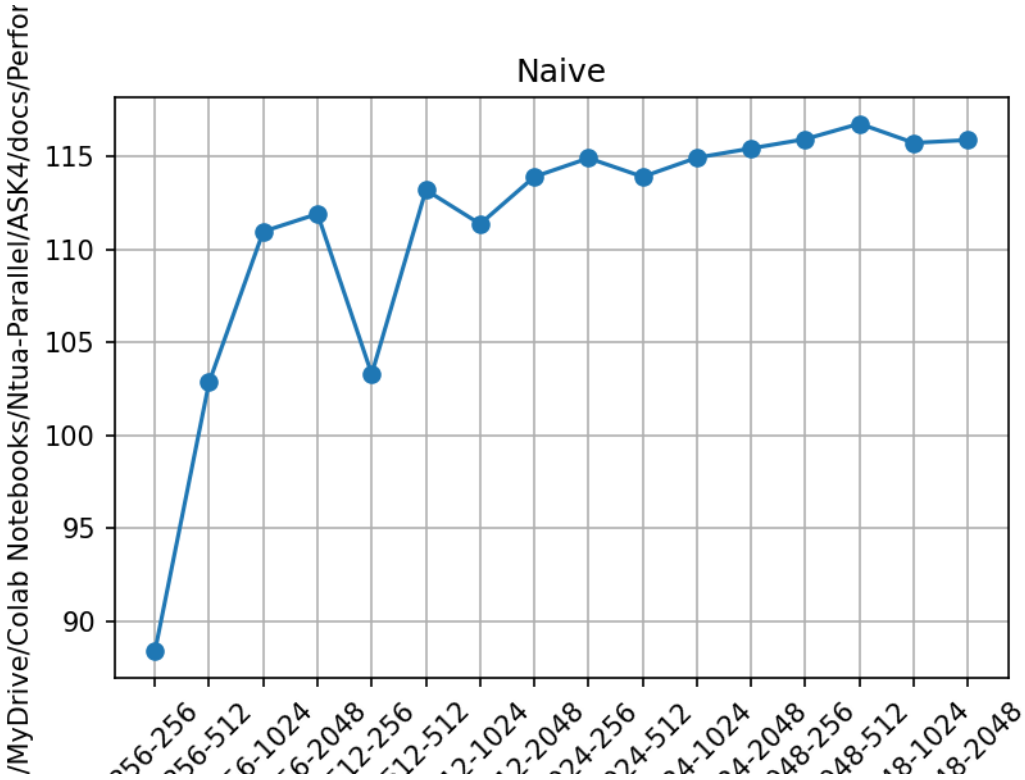
Σχήμα 6: $K = 1024$



Σχήμα 7: $K = 2048$



Σχήμα 8: $K = 2048$

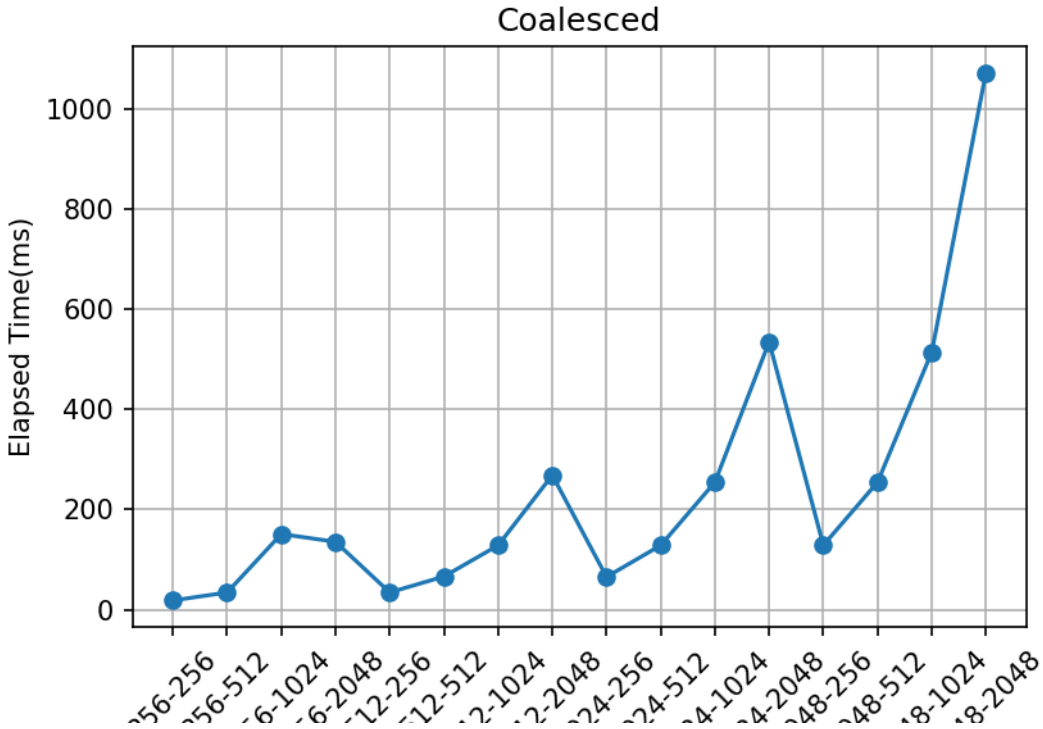


Παρατηρήσεις

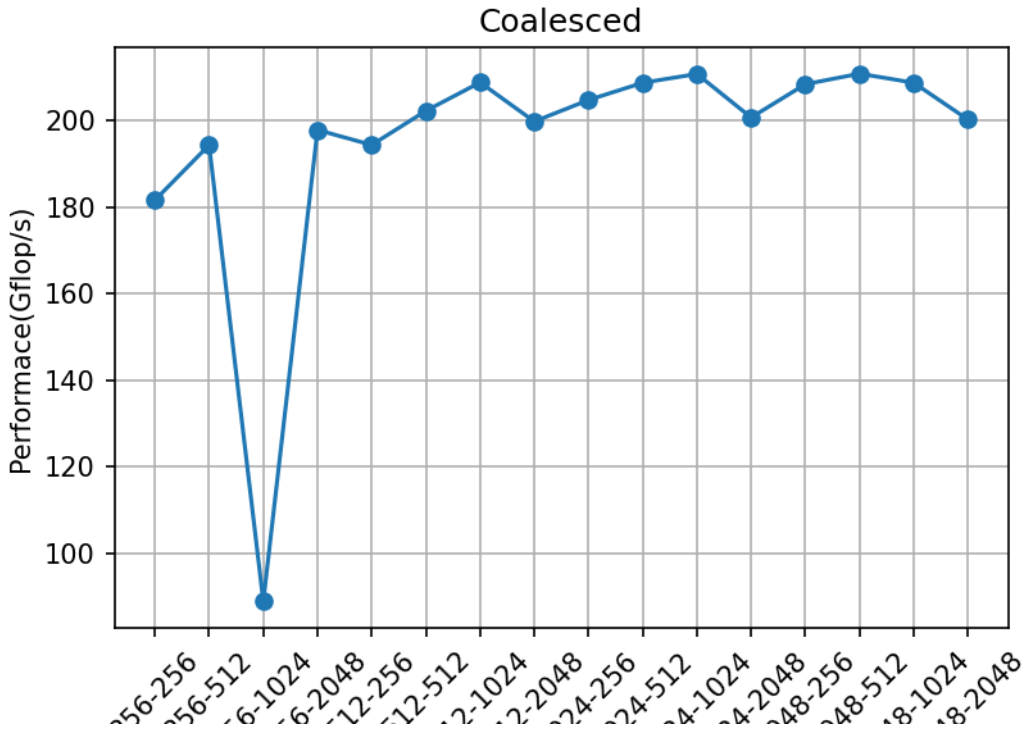
Στην υλοποίηση naive kernel η επίδοση παρουσιάζει πολλές μη προβλέψιμες αυξομοιώσεις. Αυτό συμβαίνει πιθανότατα επειδή οι προσβάσεις στη μνήμη δεν είναι καθόλου βελτιστοποιημένες και μας περιορίζει πολύ το transfer bandwidth της κάρτας γραφικών. Το μεγαλύτερο ρόλο στην επίδοση σε αυτή την υλοποίηση παίζει ο χρόνος μεταφοράς των δεδομένων.

5.2.2 Coalesced kernel

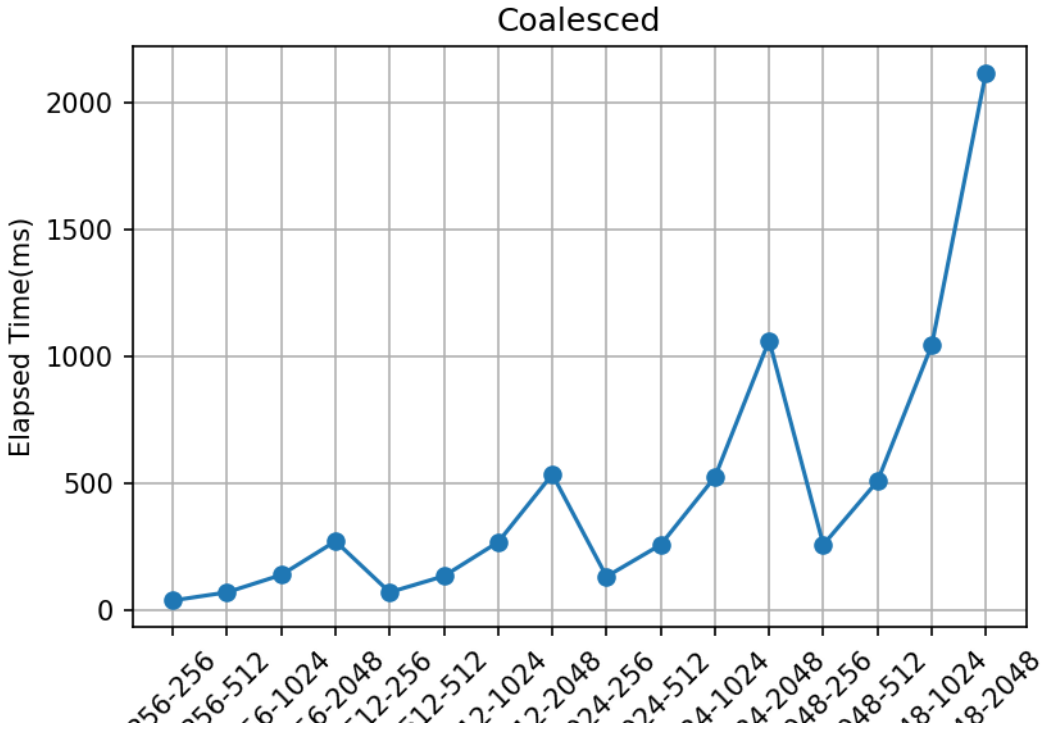
Σχήμα 9: $K = 256$



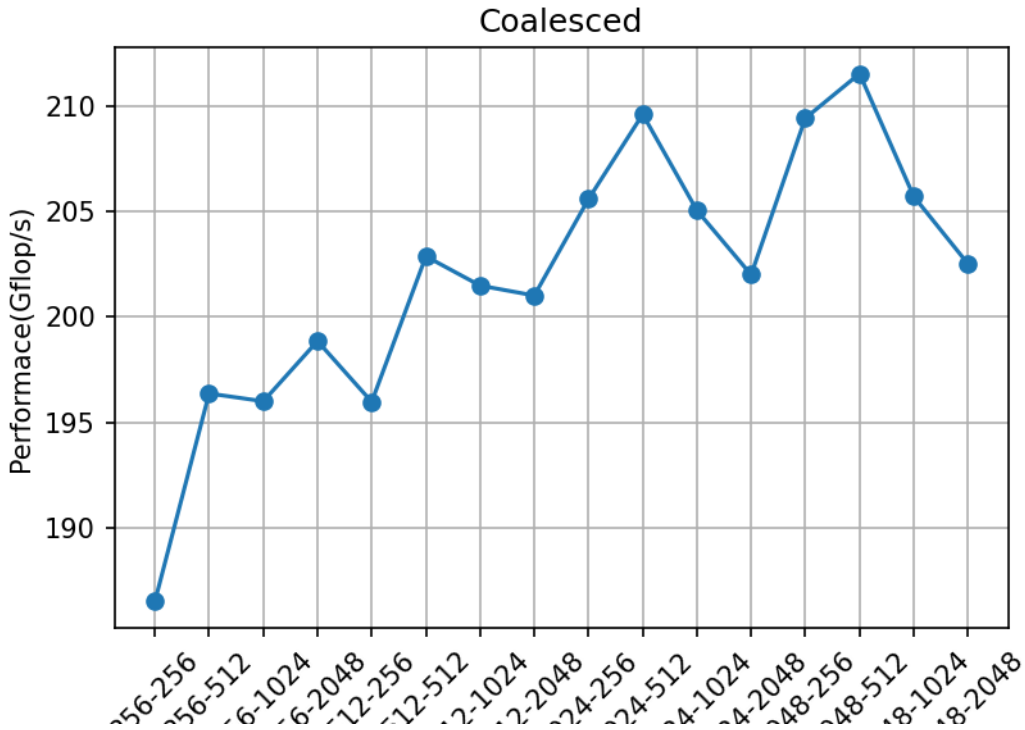
Σχήμα 10: $K = 256$



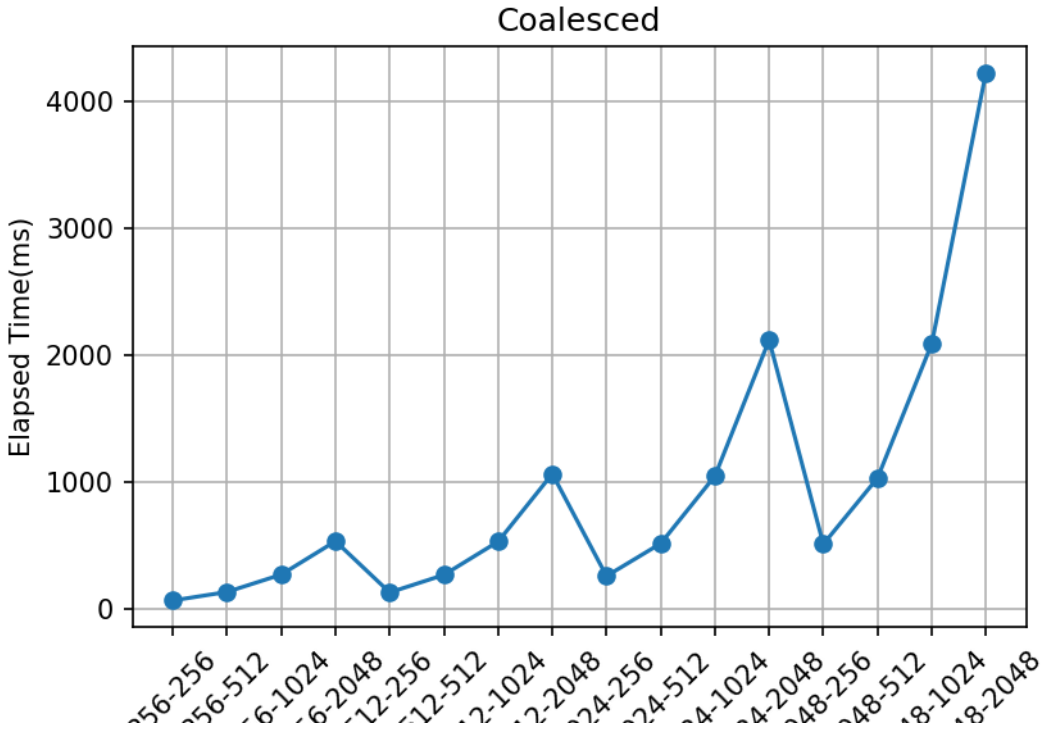
Σχήμα 11: $K = 512$



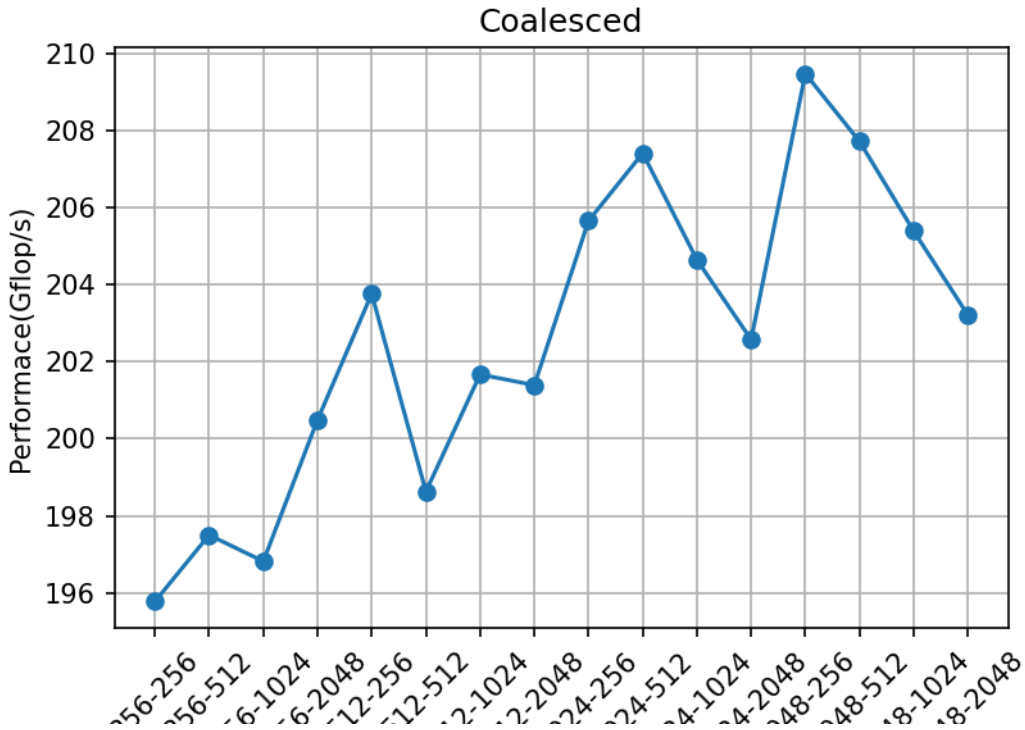
Σχήμα 12: $K = 512$



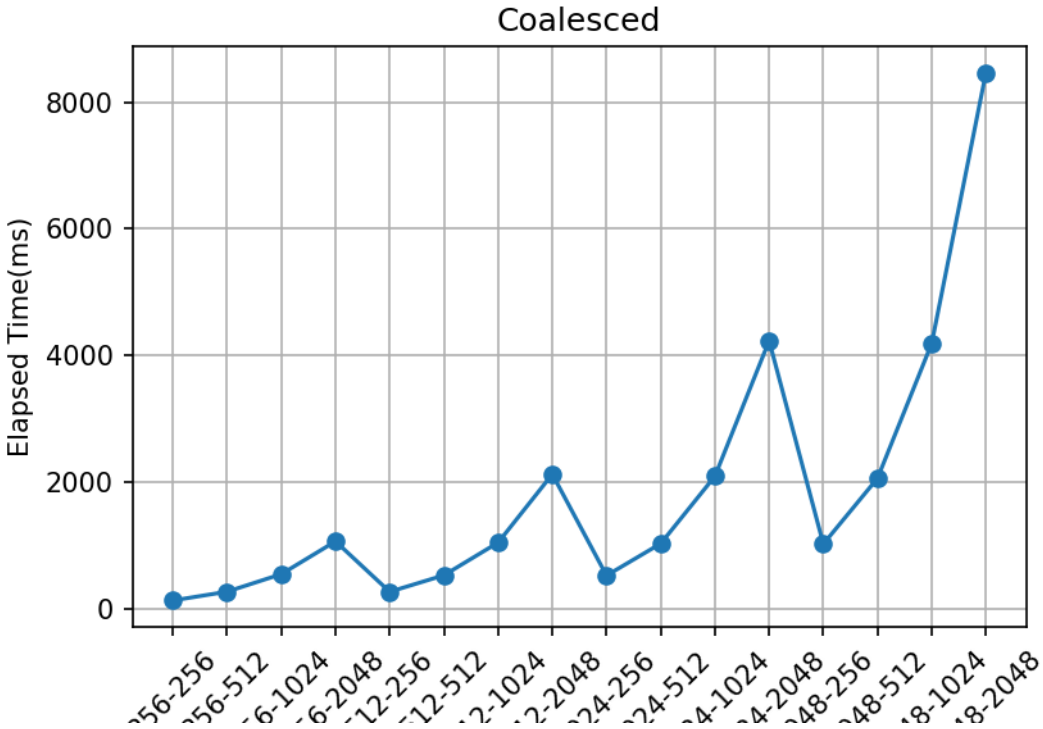
Σχήμα 13: $K = 1024$



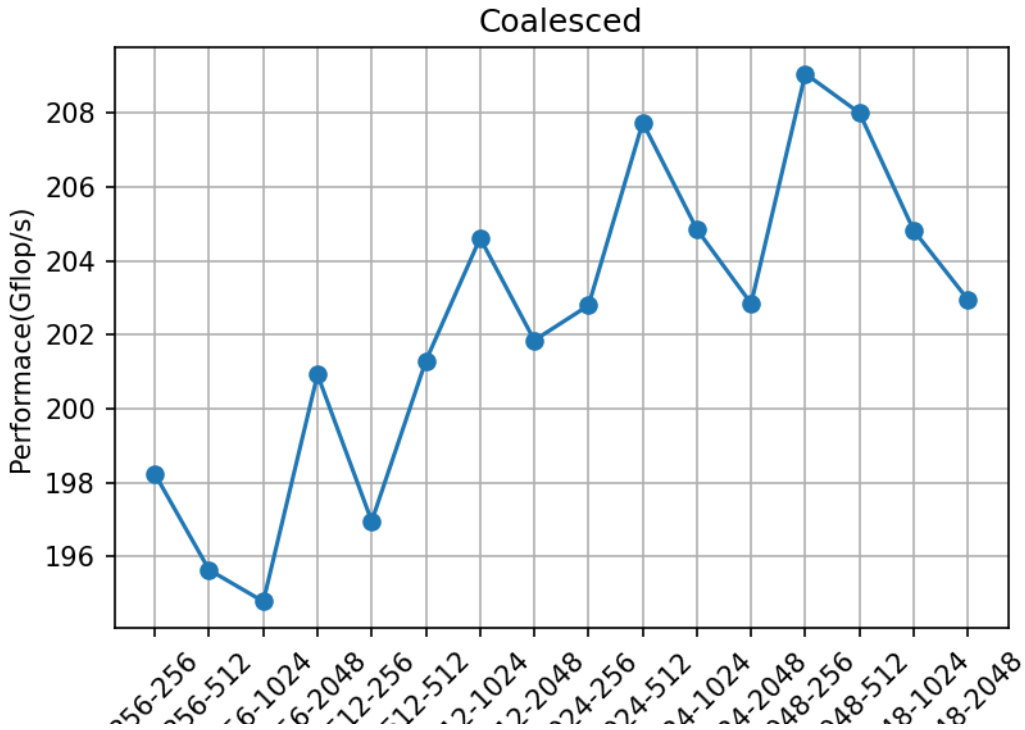
Σχήμα 14: $K = 1024$



Σχήμα 15: $K = 2048$



Σχήμα 16: $K = 2048$

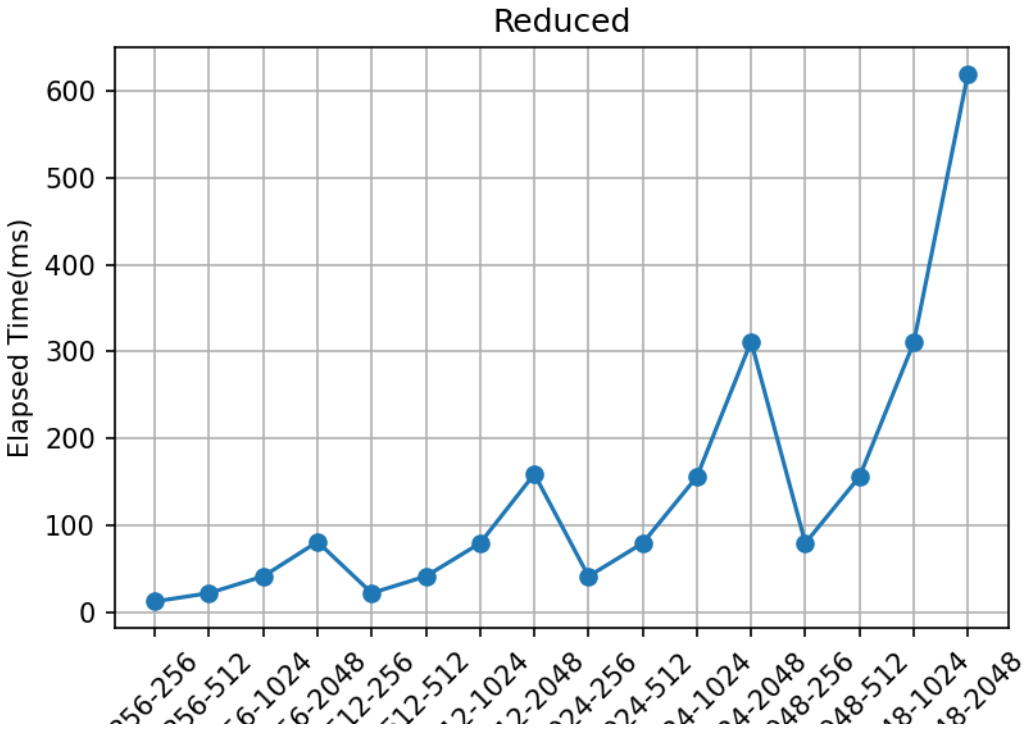


Παρατηρήσεις

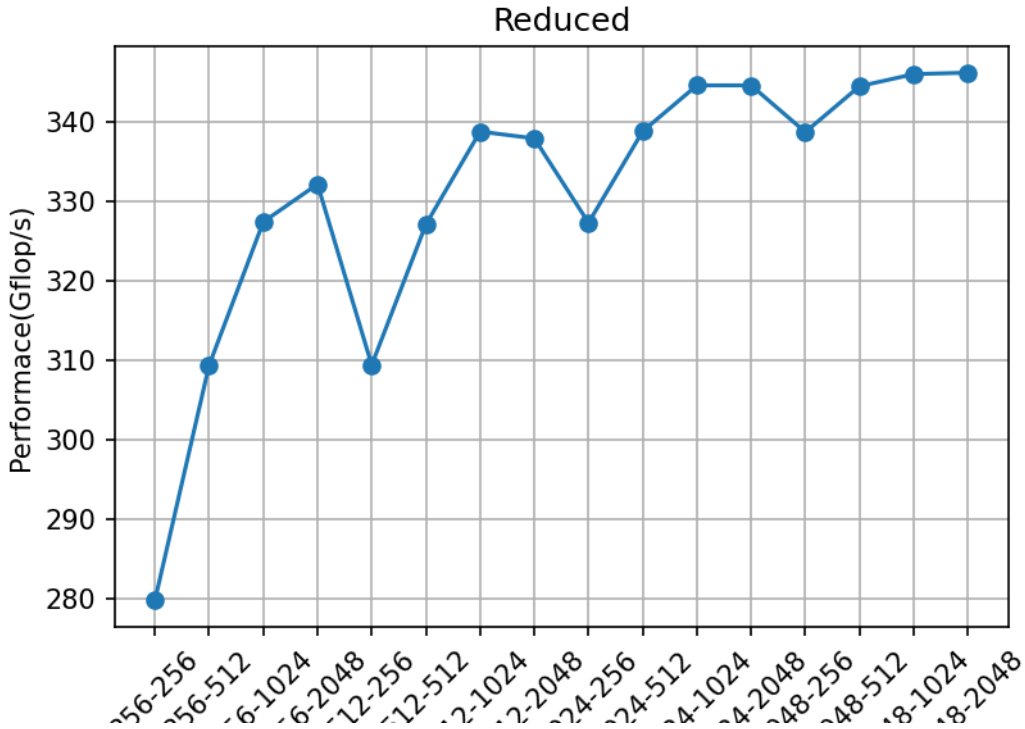
Στην υλοποίηση coalesced Kernel η επίδοση αυξάνεται όσο μεγαλώνει η διάσταση του πίνακα A και όσο μικραίνει η διάσταση του πίνακα B. Αυτό είναι λογικό εφόσον έχουμε βελτιστοποιήσει τις προσβάσεις μνήμης στον πίνακα A (άρα αυτός δεν μας περιορίζει λόγω μνήμης) αλλά όχι αυτές στον πίνακα B.

5.2.3 Shmem kernel

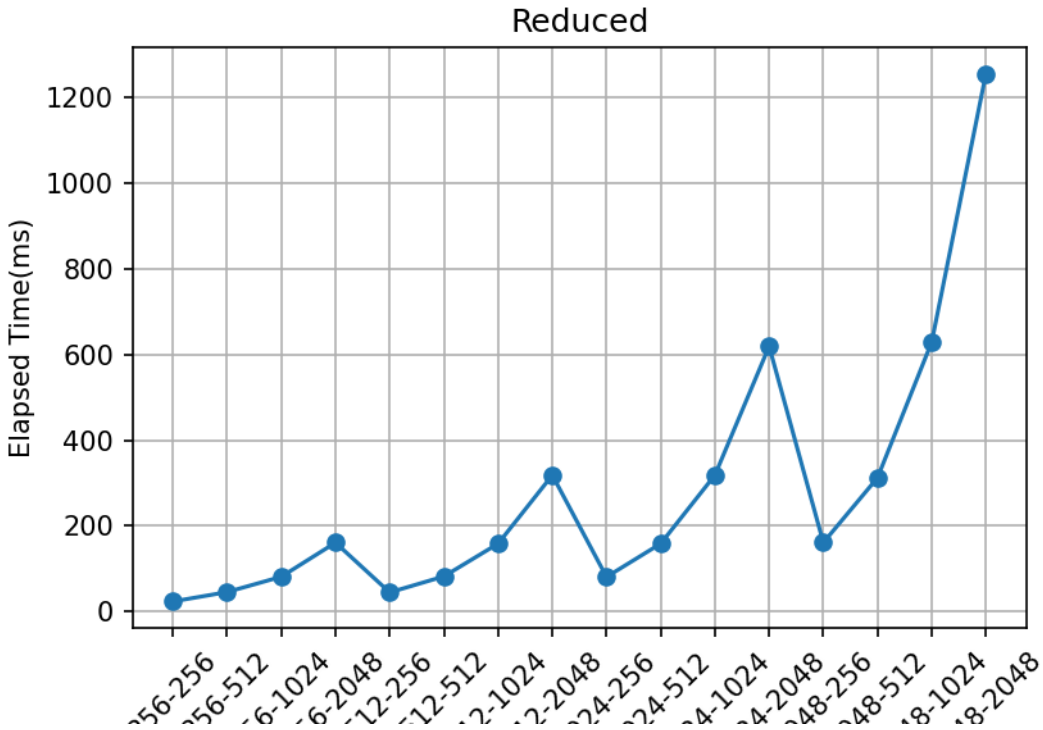
Σχήμα 17: $K = 256$



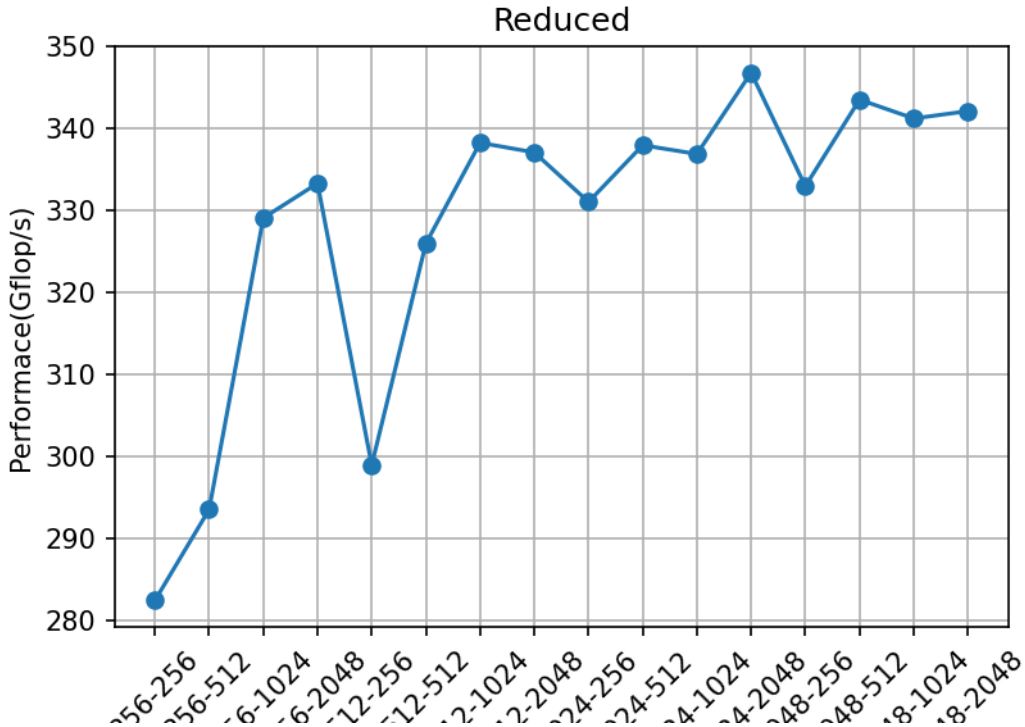
Σχήμα 18: $K = 256$



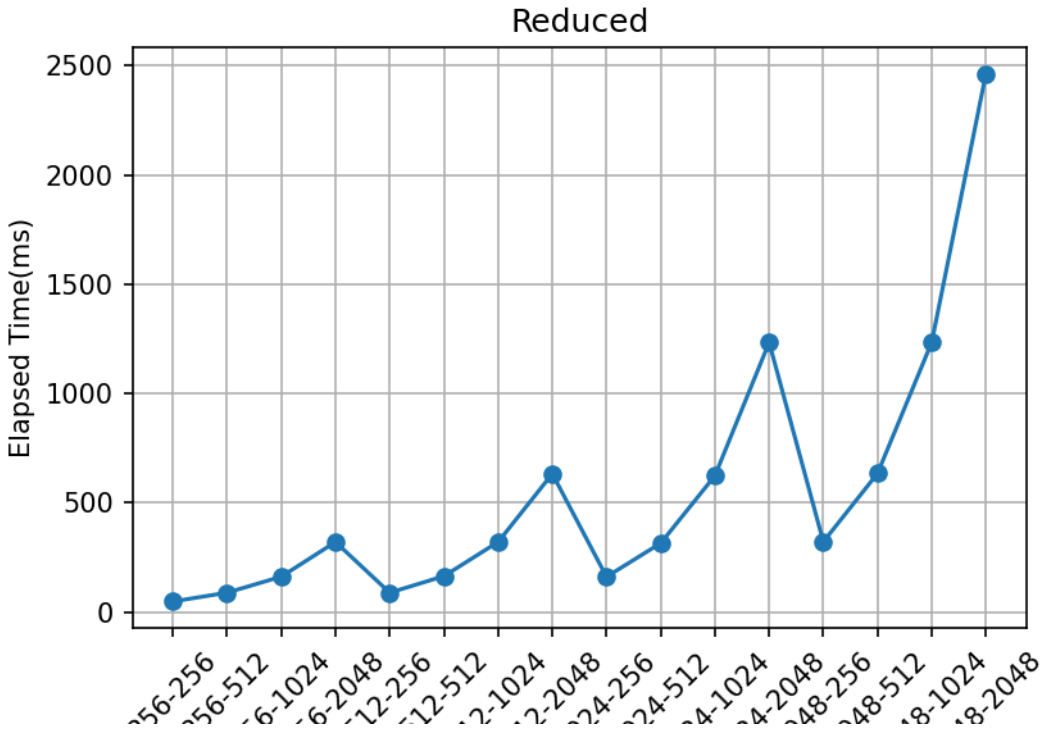
Σχήμα 19: $K = 512$



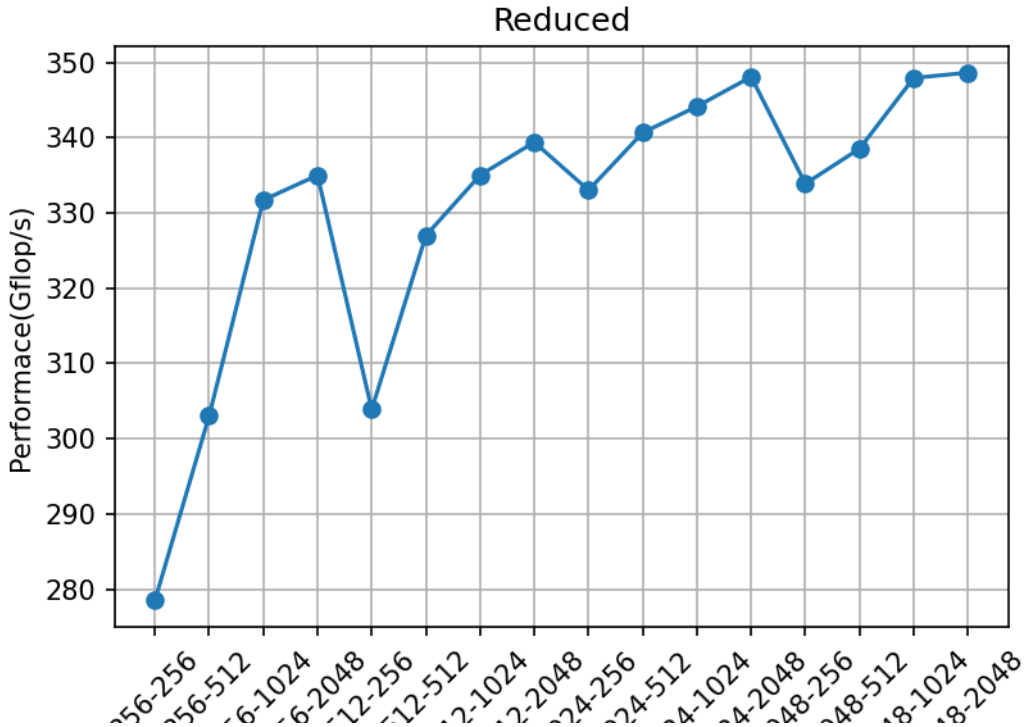
Σχήμα 20: $K = 512$



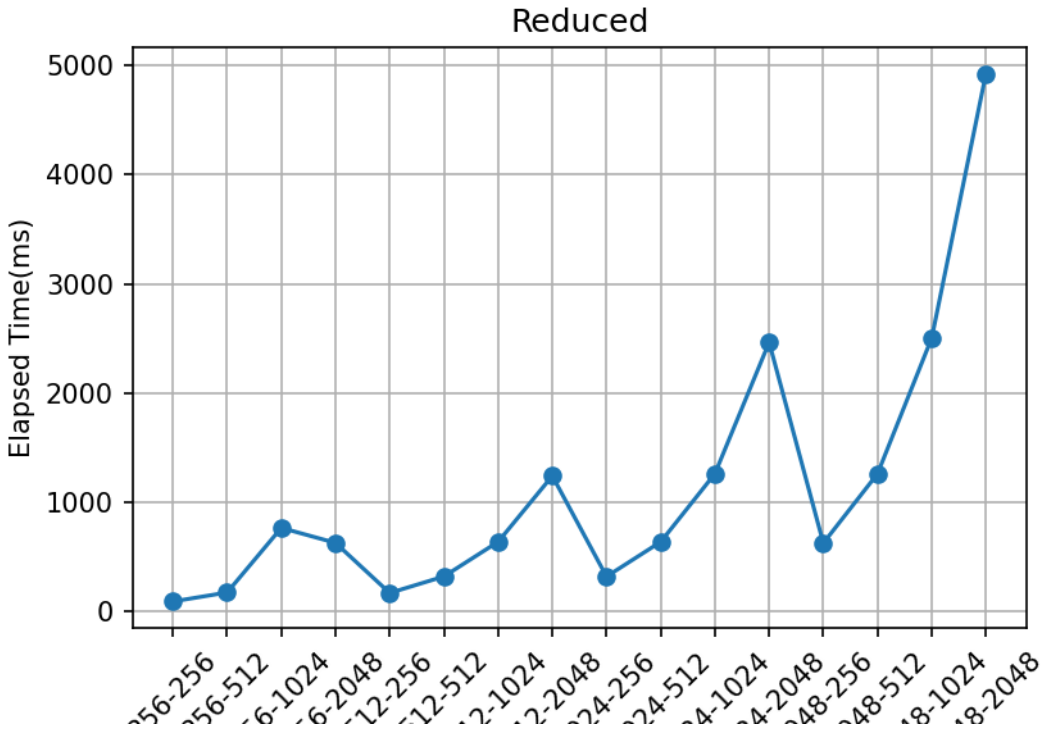
Σχήμα 21: $K = 1024$



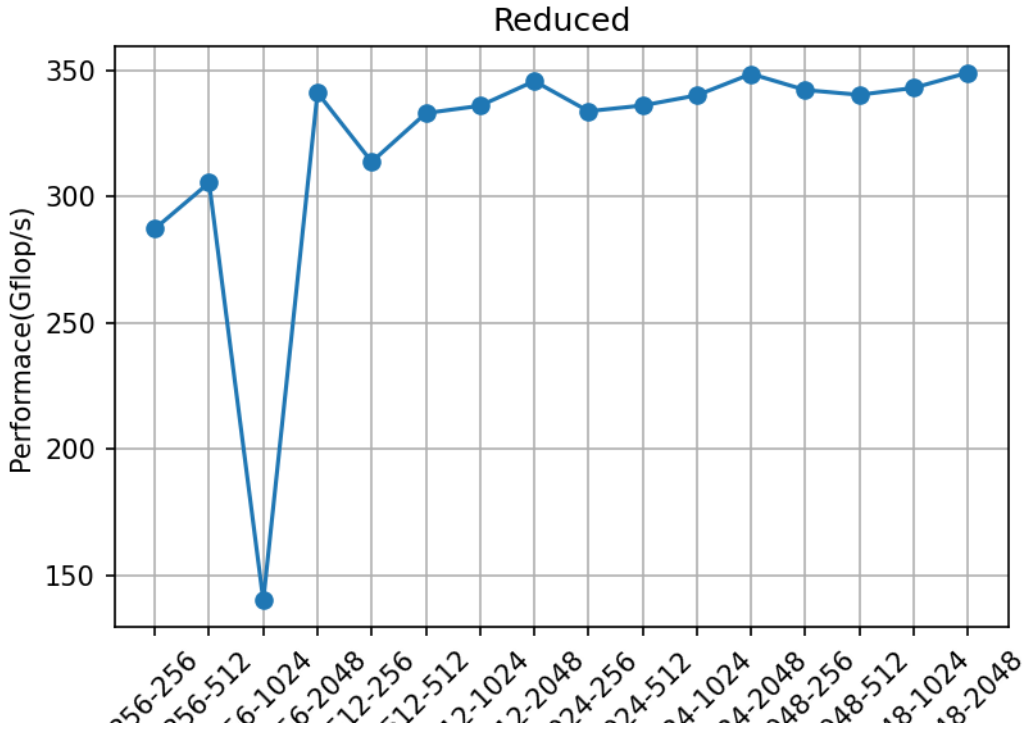
Σχήμα 22: $K = 1024$



Σχήμα 23: $K = 2048$



Σχήμα 24: $K = 2048$

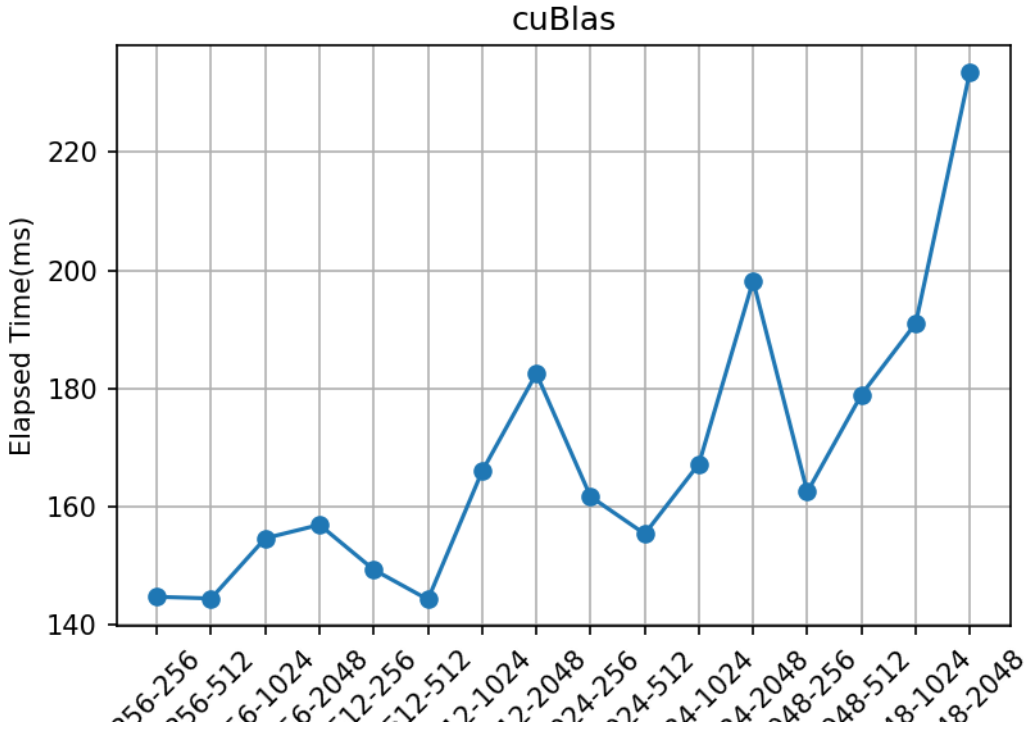


Παρατηρήσεις

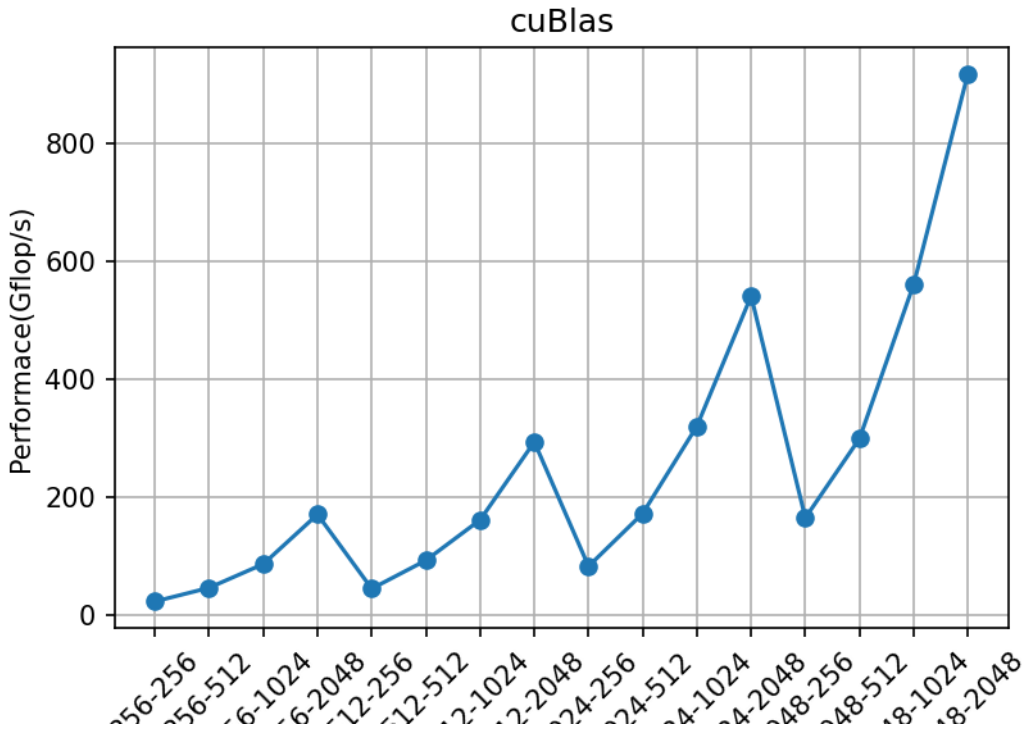
Στην υλοποίηση shared memory kernel η επίδοση αυξάνεται όσο πιο μεγάλες και ομοιογενείς είναι οι διαστάσεις των πινάκων. Αυτό είναι λογικό εφόσον έχουμε βελτιστοποιήσει όλες τις προσβάσεις στη μνήμη και μπορούμε να αξιοποιήσουμε αρκετά καλά τις δυνατότητες της GPU. Σε κάθε διάγραμμα του shared memory kernel υπάρχει μια παραμετροποίηση που ρίχνει πάρα πολύ την επίδοση, πράγμα που πιθανώς οφείλεται στην αρχιτεκτονική της GPU και τα όρια αποθήκευσης του κάθε πίνακα.

5.2.4 cuBLAS kernel

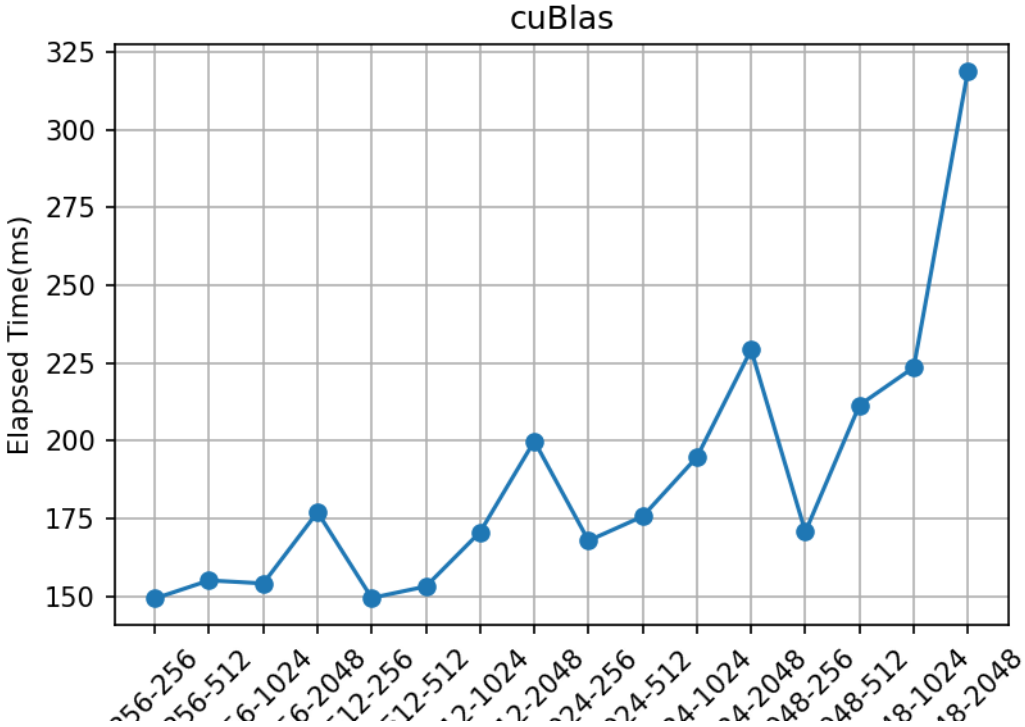
Σχήμα 25: $K = 256$



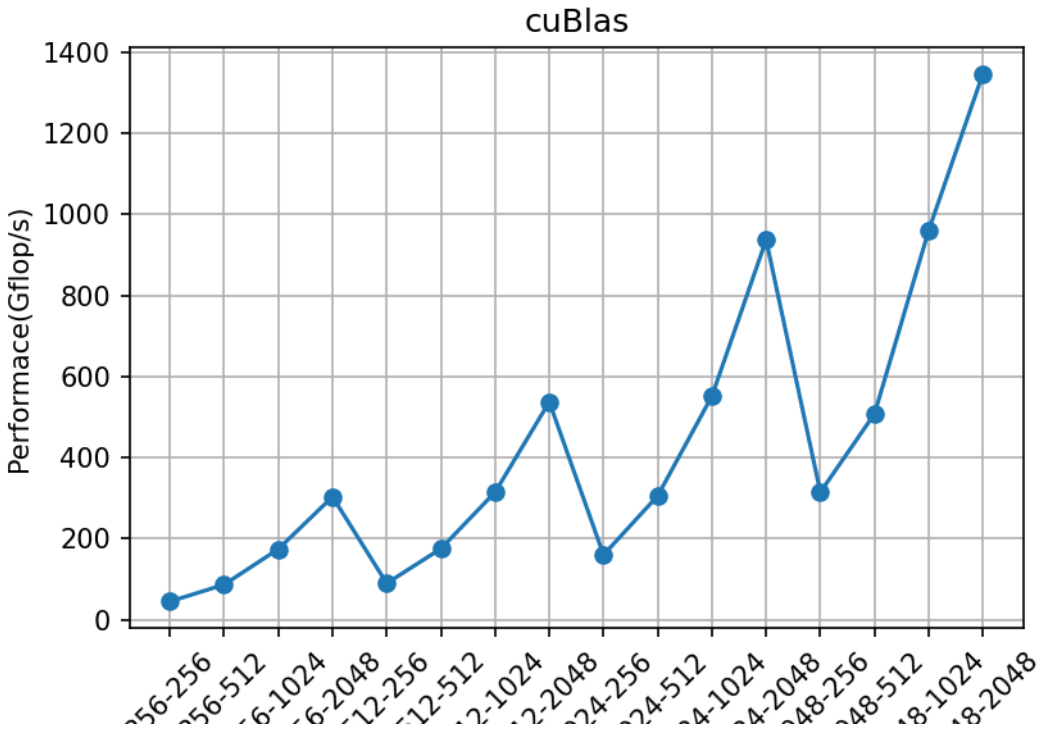
Σχήμα 26: $K = 256$



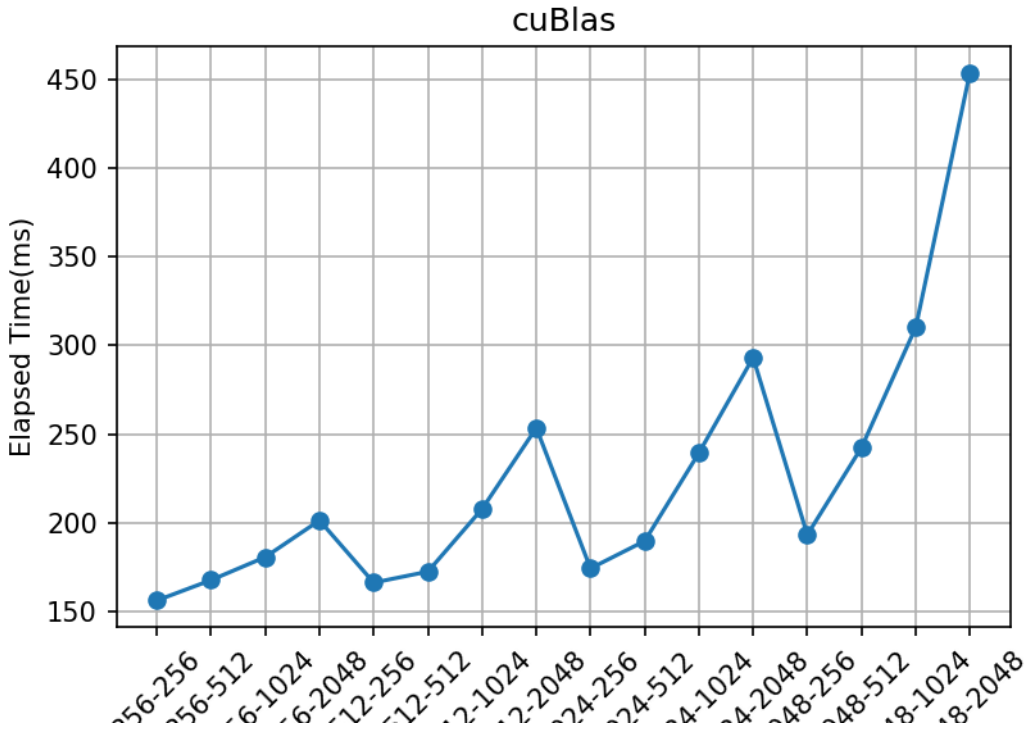
Σχήμα 27: $K = 512$



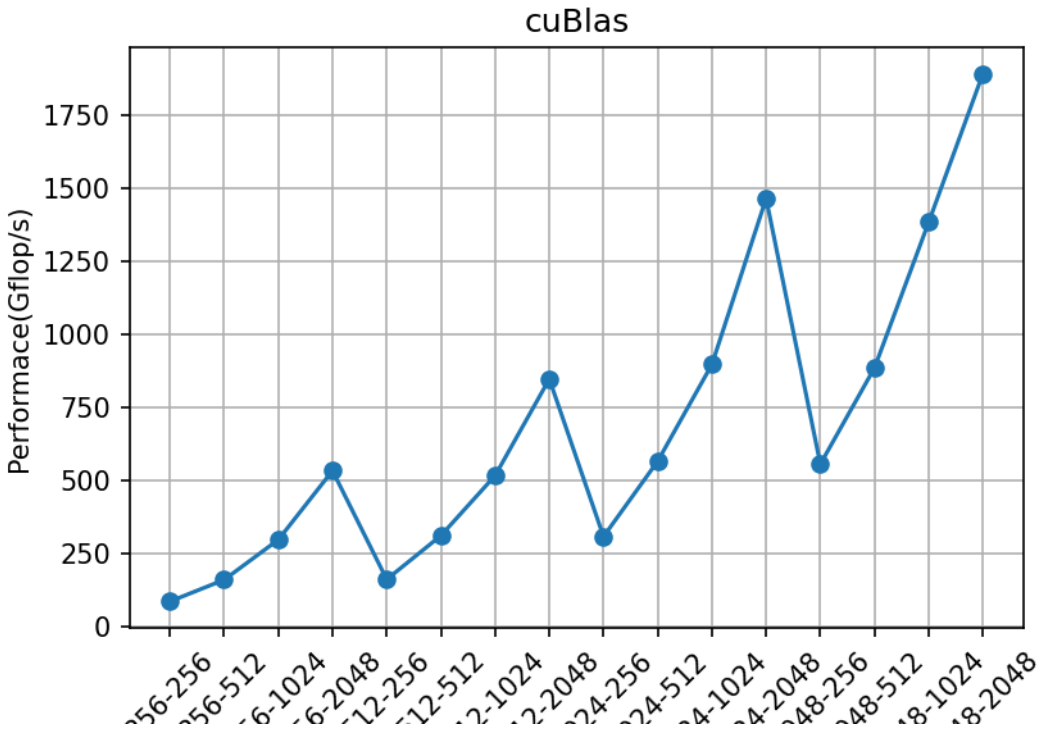
Σχήμα 28: $K = 512$



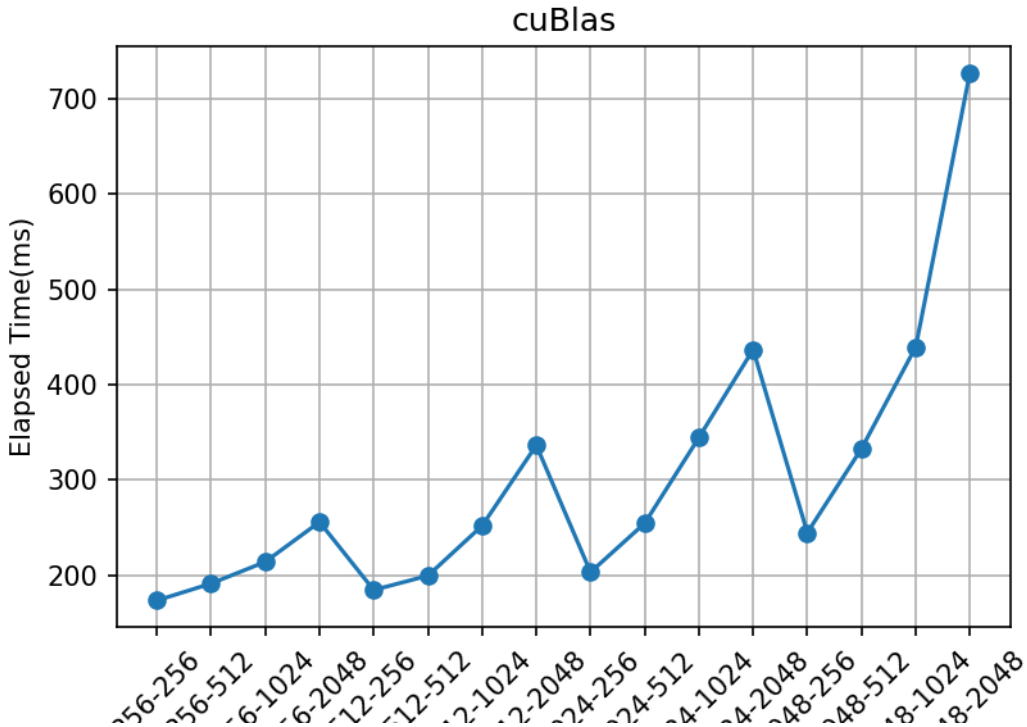
Σχήμα 29: $K = 1024$



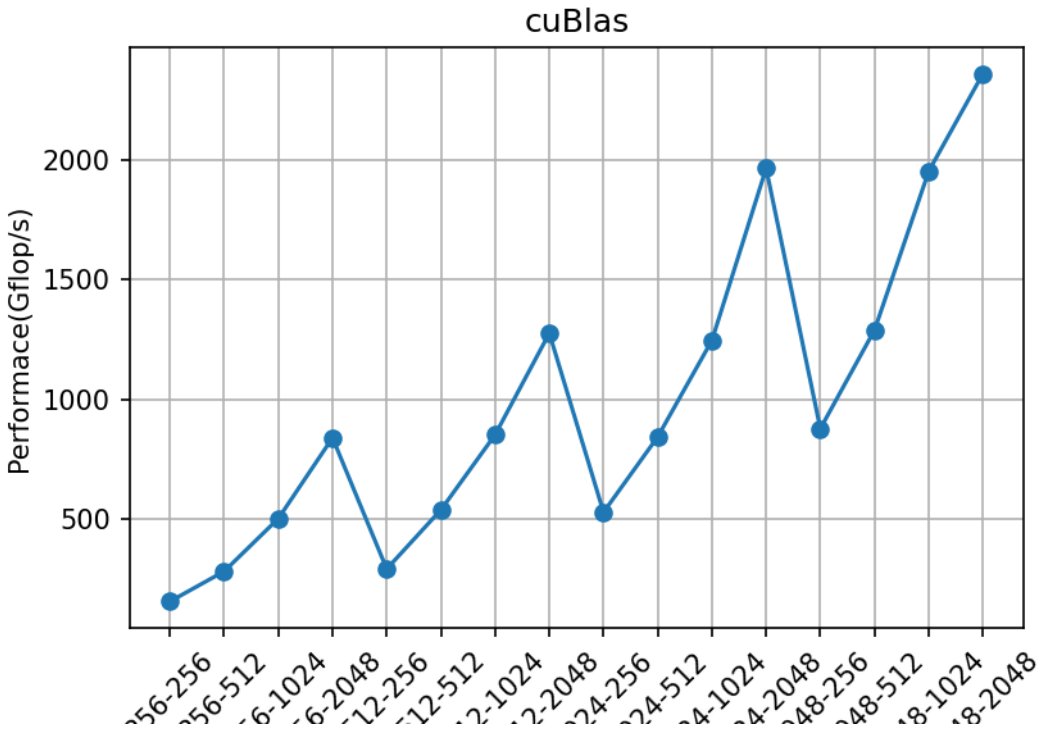
Σχήμα 30: $K = 1024$



Σχήμα 31: $K = 2048$



Σχήμα 32: $K = 2048$

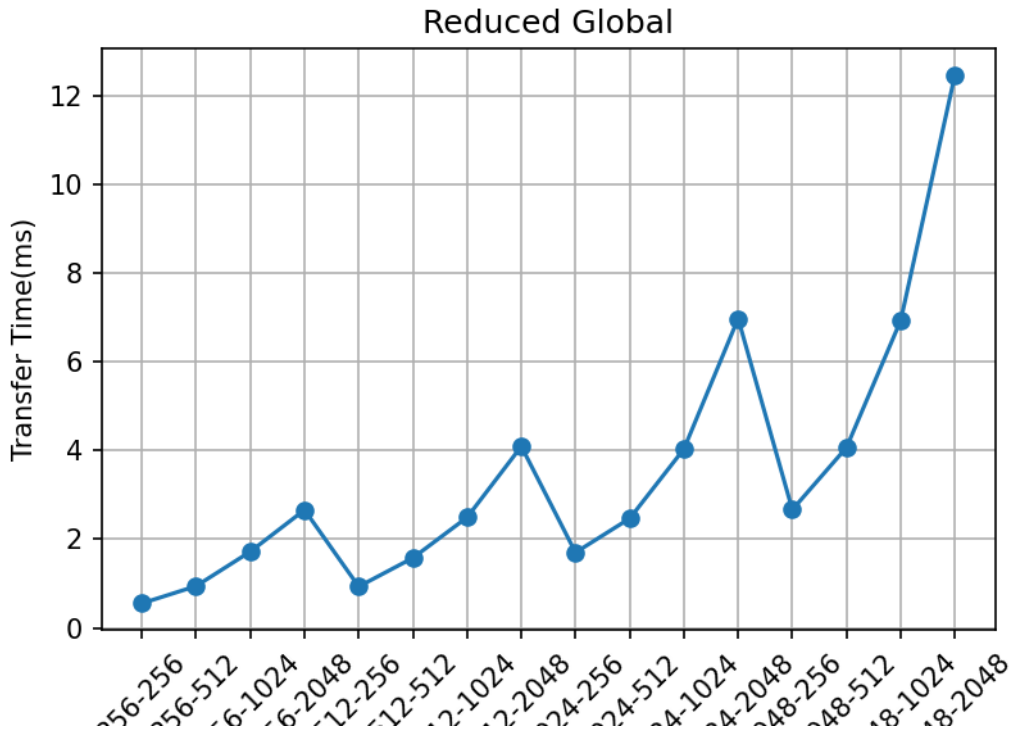


Παρατηρήσεις:

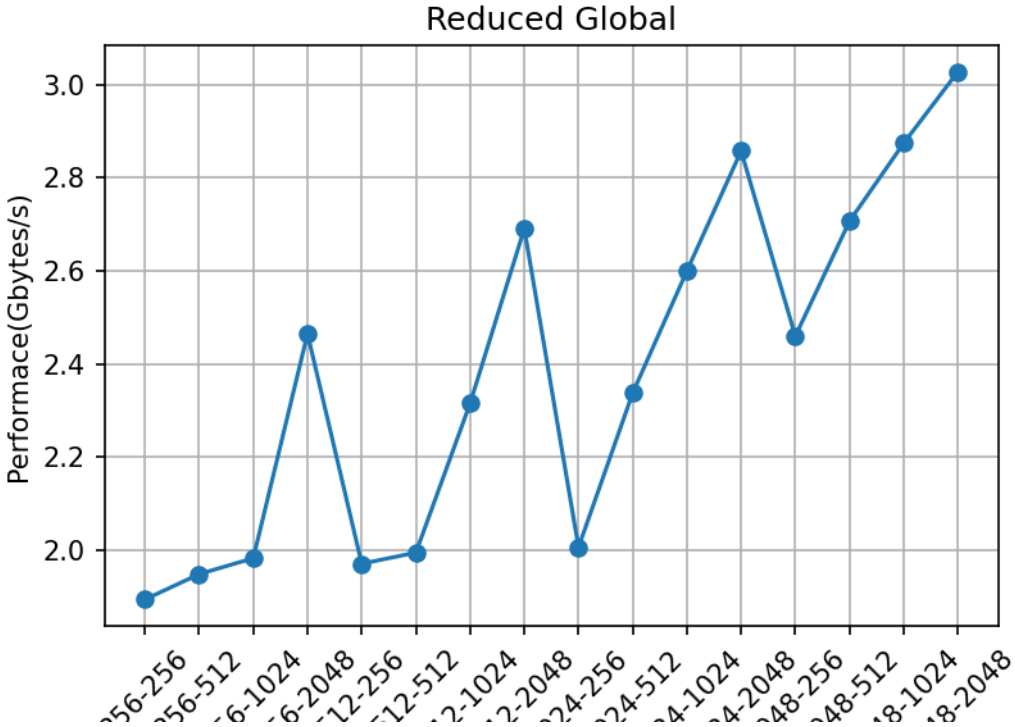
Στην υλοποίηση με cuBLAS, όσο μεγαλύτερες και όσο πιο ομοιογενείς είναι οι διαστάσεις των πινάκων, τόσο μεγαλύτερη αξιοποίηση του performance έχουμε. Αυτό είναι αναμενόμενο αφού το cuBLAS είναι μια βελτιστοποιημένη βιβλιοθήκη που έχει δημιουργηθεί για να αναλαμβάνει πολύ μεγάλες εφαρμογές.

5.3 Μετρήσεις Transfer Overhead

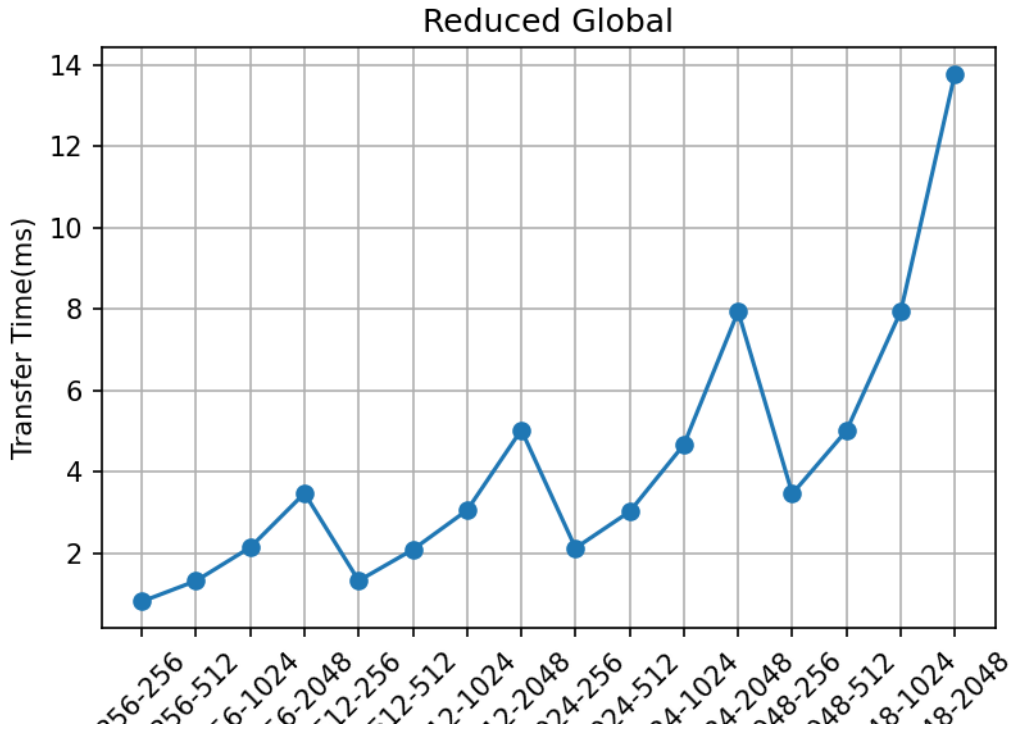
Σχήμα 33: $K = 256$



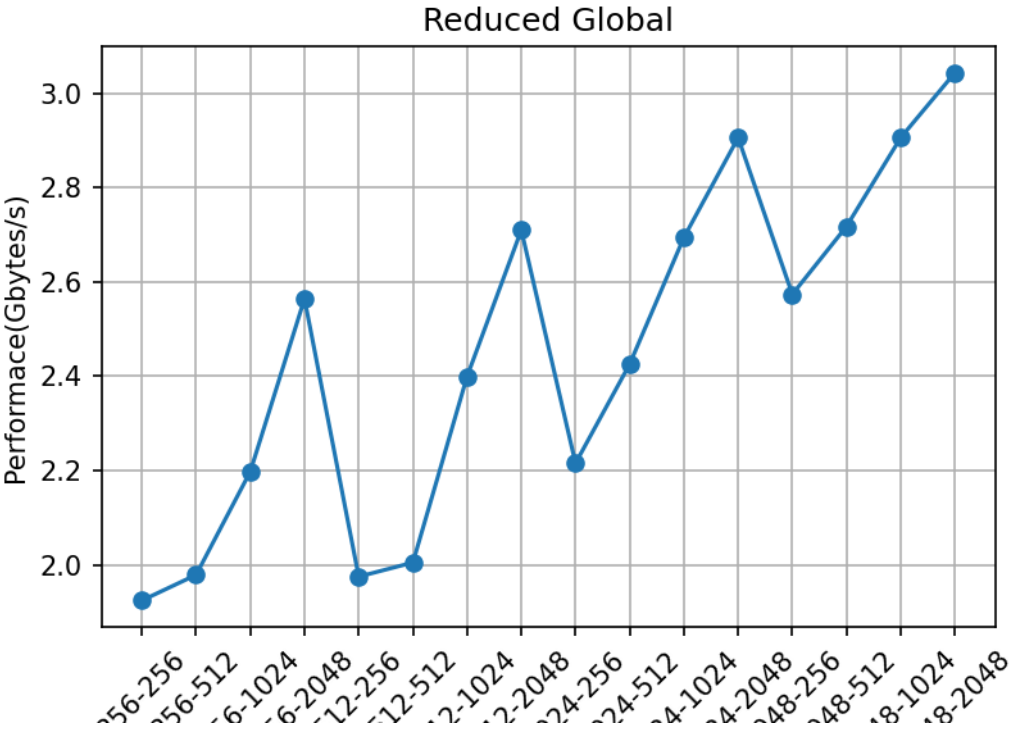
Σχήμα 34: $K = 256$



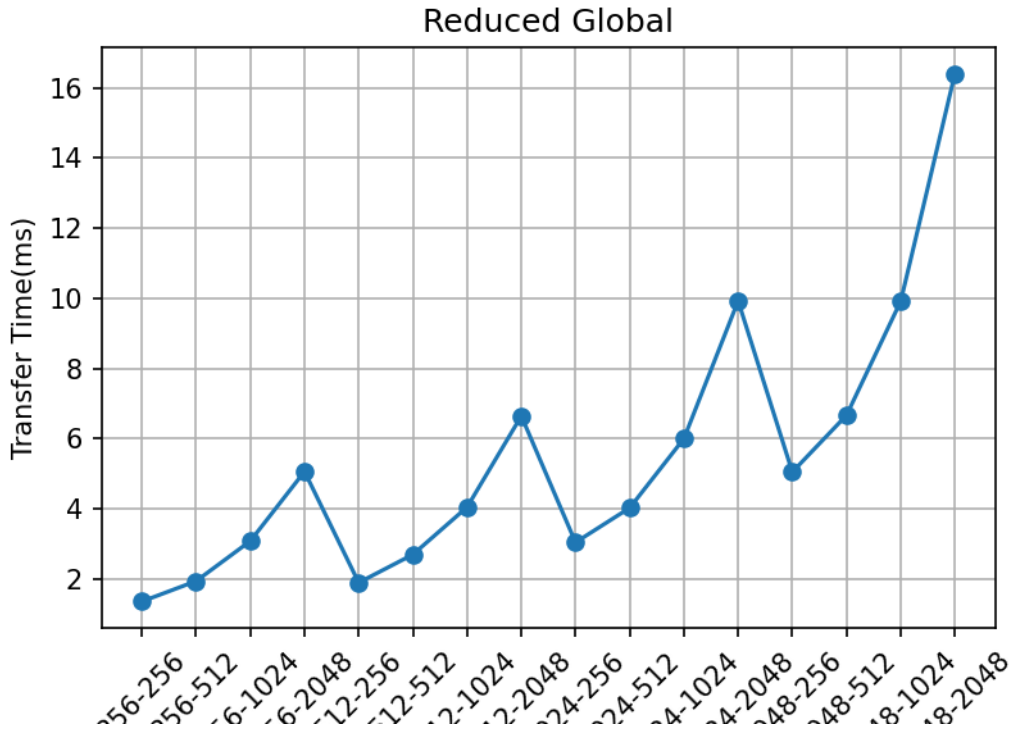
Σχήμα 35: $K = 512$



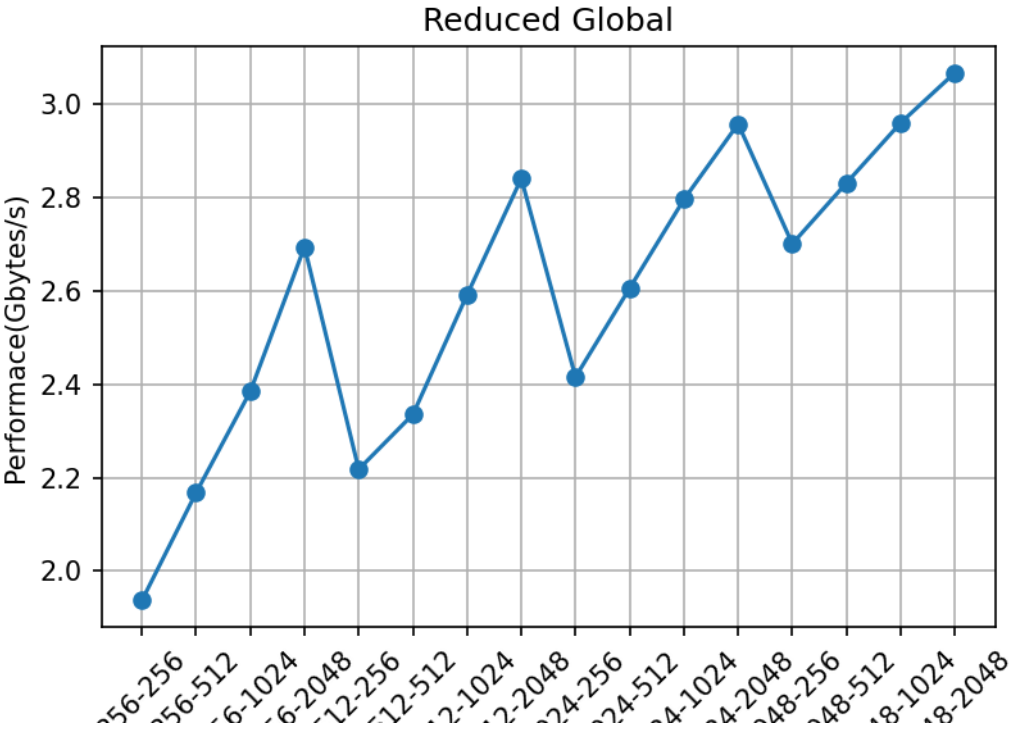
Σχήμα 36: $K = 512$



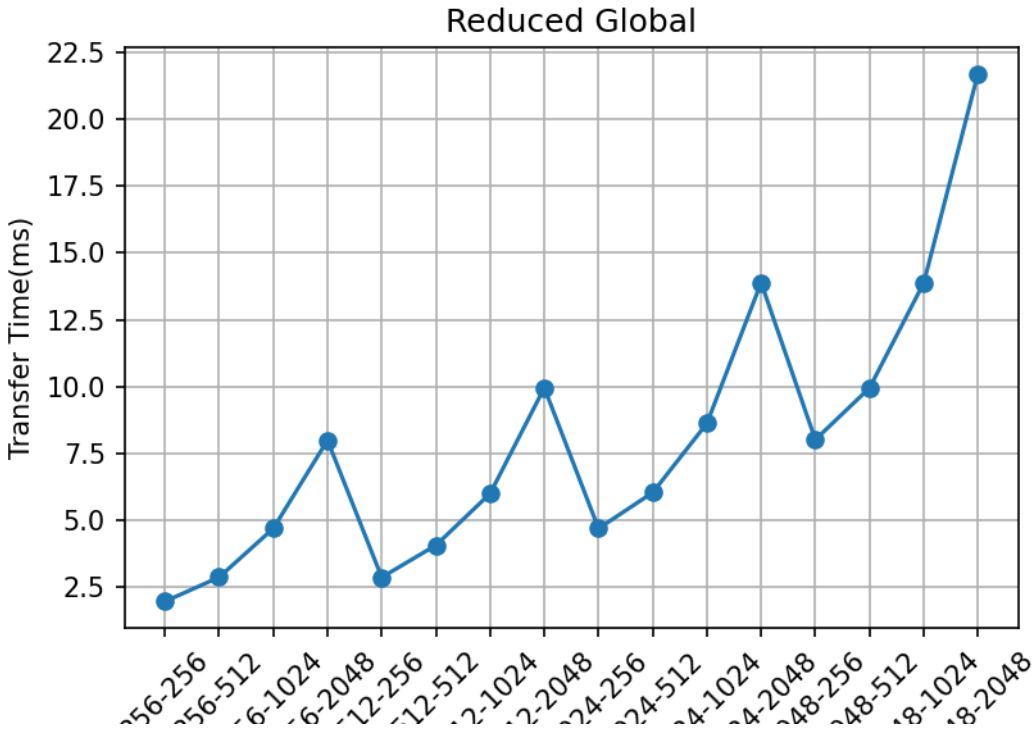
Σχήμα 37: $K = 1024$



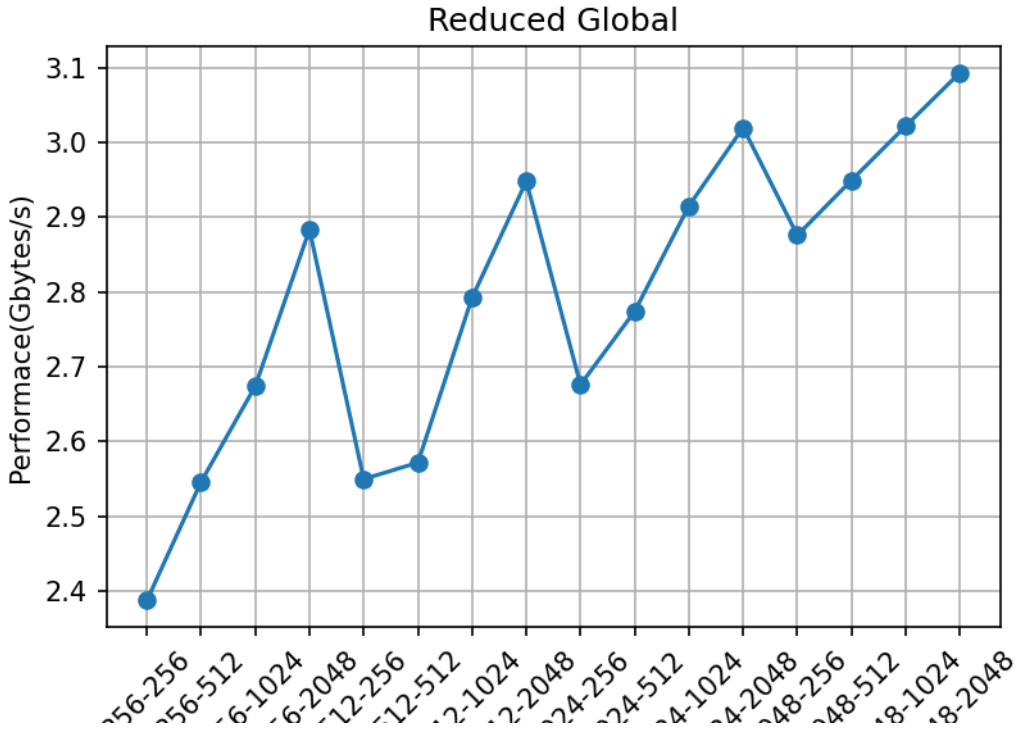
Σχήμα 38: $K = 1024$



Σχήμα 39: $K = 2048$



Σχήμα 40: $K = 2048$



Παρατηρήσεις:

Κατά βάση η χρήση της συγκεκριμένης GPU δικαιολογείται όταν έχουμε μεγάλα M, N, K ώστε να μπορούμε να επιτύχουμε όσο το δυνατόν λιγότερο transfer overhead (μέγιστο performance σε $Gbytes/s$) ώστε να μην περιοριζόμαστε τόσο από το memory bandwidth.