9ο Εξάμηνο, Ακαδημαϊκό Έτος 2021-2022

Συστήματα Παράλληλης Επεξεργασίας

Αναφορά 4ης Εργαστηριακής Άσκησης:
Παράλληλος προγραμματισμός σε επεξεργαστές γραφικών

parlab20

Ζάρα Στέλλα, 03117154
Λιάγκα Αικατερίνη, 03117208

2. Ζητούμενα

2.1 Υλοποίηση για επεξεργαστές γραφικών (GPUs)

Βασική υλοποίηση

Αρχικά, ορίσαμε τις διαστάσεις των block και grid για τη GPU στο αρχείο dmm_main.cu, όπως φαίνεται παρακάτω:

```
dim3 gpu_block(THREAD_BLOCK_Y, THREAD_BLOCK_X);

dim3 gpu_grid((N+THREAD_BLOCK_Y-1)/THREAD_BLOCK_Y,
(M+THREAD_BLOCK_X-1)/THREAD_BLOCK_X);
```

Ορίστηκε ένα block 2 διαστάσεων, με διαστάσεις THREAD_BLOCK_X, THREAD_BLOCK_Y πάνω στις οποίες βασίστηκε και ο ορισμός των διαστάσεων του grid, το οποίο ορίστηκε με τέτοιο τρόπο ώστε να οι διαστάσεις του να είναι πολλαπλάσιες των διαστάσεων του block.

Παρακάτω παρατίθεται μια πρώτη υλοποίηση για το DMM σε GPU (οι υλοποιήσεις των kernel βρίσκονται όλες στο αρχείο dmm_gpu,cu). Κάθε thread είναι υπεύθυνο για τον υπολογισμό ενός στοιχείου με βάση τη θέση του στο thread block. Από τον πίνακα A θέλουμε τα στοιχεία των γραμμών και από τον πίνακα B θέλουμε αντίστοιχα τα στοιχεία των στηλών. Για K επαναλήψεις έχουμε το άθροισμα όλων των επιμέρους γινομένων, το οποίο στο τέλος τοποθετείται στην κατάλληλη θέση του πίνακα C.

```
__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)
{

    int tidx,tidy;
    value_t sum = 0;
    tidx = blockDim.x * blockIdx.x + threadIdx.x;
    tidy = blockDim.y * blockIdx.y + threadIdx.y;

    if (tidx<N && tidy<M)
    {
        for (int i=0; i<K; i++)
        {
            sum += A[tidy*K+i]*B[tidx+N*i];
        }

        C[tidy*N+tidx] = sum;
    }
}
```

1. Ο πίνακας A εκτελεί K προσβάσεις στη μνήμη αφού φέρνει τα K στοιχεία της σειράς υπό επεξεργασία και K προσβάσεις αντίστοιχα εκτελεί και ο πίνακας B αφού η κάθε στήλη έχει K στοιχεία. Έτσι έχουμε $2 \cdot K$ προσβάσεις σύνολο για κάθε ένα από τα $M \cdot N$ στοιχεία του πίνακα C. Άρα, έχουμε συνολικά $2 \cdot K \cdot M \cdot N$ προσβάσεις στη μνήμη.

2. Στο εσωτερικό loop έχουμε 2 προσβάσεις στη μνήμη, μία από τον πίνακα A και μία για τον πίνακα B ενώ έχουμε ακόμα 2 floating point operations, μία πρόθεση και έναν πολλαπλασιασμό. Έτσι, συνολικά έχουμε $\frac{2\ flops}{2 \cdot 4\ bytes} = \frac{1}{4}\frac{flops}{byte}$.

   Η GPU που χρησιμοποιήθηκε, η NVIDIA Tesla K40c με αρχιτεκτονική Kepler, έχει τα εξής χαρακτηριστικά:
   - ➔ Peak single precision floating point performance: 4.29Tflops
   - ➔ Memory Bandwidth: 288 GB/s

   Άρα, το Operational Intensity της GPU είναι $\frac{4.29\ Tflops}{288\ GB/s} = 14.89\frac{Flops}{byte}$. Αφού ο kernel έχει 0.25 $\frac{Flops}{byte}$ θα βρίσκεται αριστερότερα του “γονάτου” στο roofline model, οπότε η υλοποίηση είναι memory-bound.

3. Ο πίνακας A διαχειρίζεται γραμμές και αφού οι πίνακες είναι αποθηκευμένοι στη μνήμη σε row-major format, οι προσβάσεις του στην κύρια μνήμη συνενώνονται. Αντιθέτως, ο πίνακας B διαχειρίζεται στήλες οπότε οι προσβάσεις του δεν συνενώνονται.

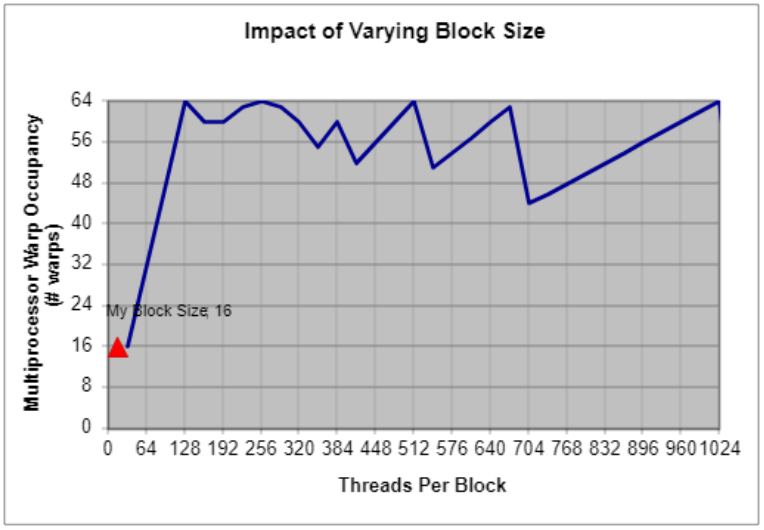
4. Το CUDA Occupancy Calculator μας δίνει πληροφορίες σχετικά με τη χρησιμοποίηση των πολυεπεξεργαστικών στοιχείων, λαμβάνοντας υπόψη πλήθος παραμέτρων. Για αρχή, λαμβάνονται υπόψη τα Compute Capability και Shared Memory Size Config που για όλες τις περιπτώσεις είναι σταθερά και έχουν τις εξής τιμές:
   - Compute Capability: 3,5
   - Shared Memory Size Config: 65536 bytes

   Για να κάνουμε δοκιμές για τη χρησιμοποίηση των πολυεπεξεργαστικών στοιχείων θα πρέπει εκτός από το Block Size να μεταβάλουμε αντίστοιχα τα μεγέθη Registers Per Thread και User Shared Memory Per Block. Τις πληροφορίες αυτές μπορούμε να τις λάβουμε κατά το make στο log του αρχείου .err που ορίσαμε. Παρακάτω παρουσιάζονται τα αποτελέσματα για Block Size 4x4, 8x8, 16x16, 32x32, τιμές για τις οποίες έγιναν και οι προσομοιώσεις αργότερα.

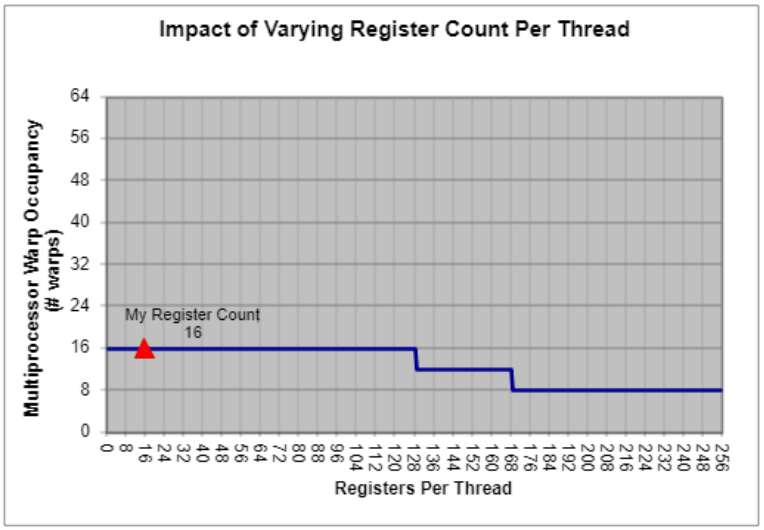
   - Block Size : 4x4 = 16 threads
     Registers Per Thread : 16
     User Shared Memory Per Block : 0

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	512
Active Warps per Multiprocessor	16
Active Thread Blocks per Multiprocessor	16
Occupancy of each Multiprocessor	25%

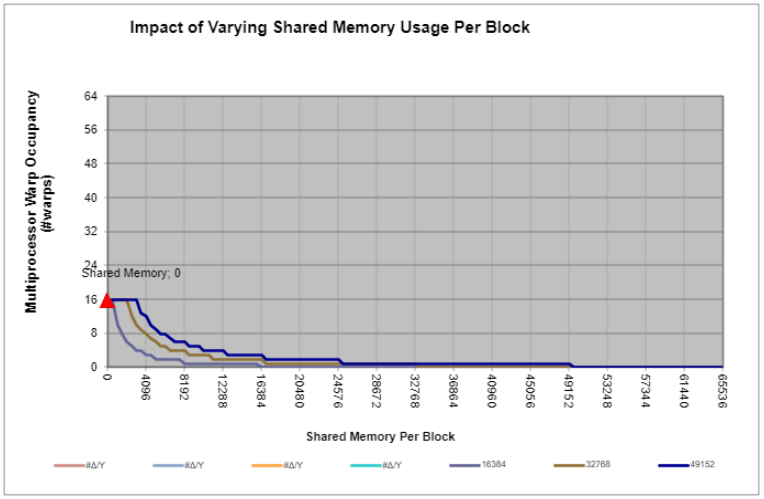
Χρησιμοποίηση πολυεπεργαστικών στοιχείων - Naive Υλοποίηση - Block Size 4x4



Warp Occupancy depending on Threads per Block - Naive Υλοποίηση - Block Size 4x4



Warp Occupancy depending on Registers per Thread - Naive Υλοποίηση - Block Size 4x4



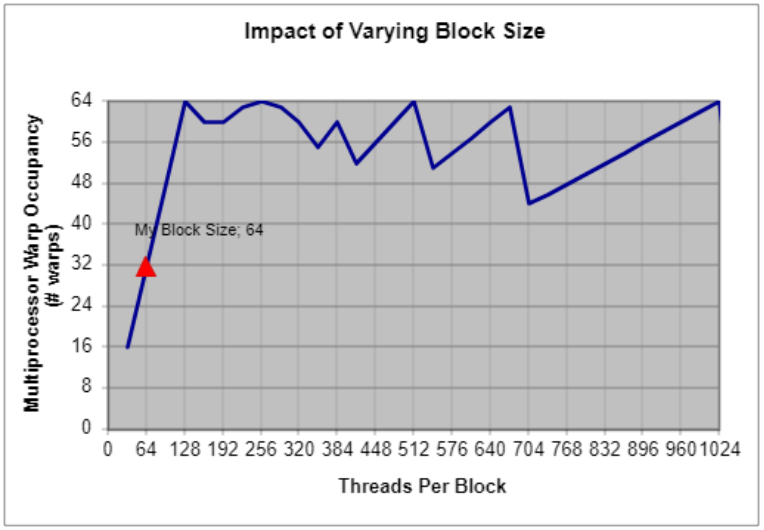
Warp Occupancy depending on Shared Memory Usage per Block - Naive Υλοποίηση - Block Size 4x4

- Block Size : 8x8 = 64 threads
  Registers Per Thread : 16

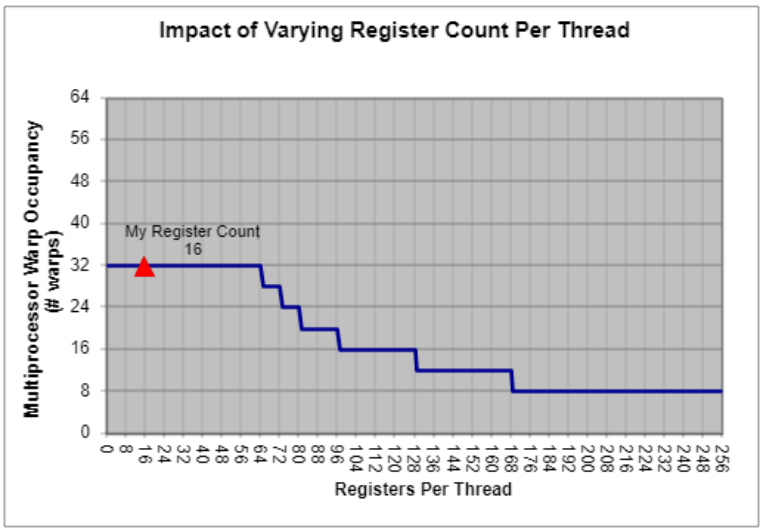
User Shared Memory Per Block : 0

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	16
Occupancy of each Multiprocessor	50%

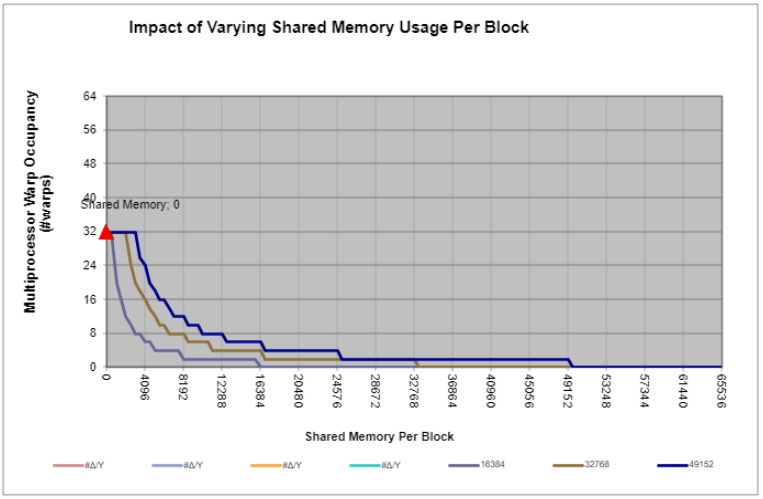
Χρησιμοποίηση πολυεπεργαστικών στοιχείων - Naive Υλοποίηση - Block Size 8x8



Warp Occupancy depending on Threads per Block - Naive Υλοποίηση - Block Size 8x8



Warp Occupancy depending on Registers Per Thread - Naive Υλοποίηση - Block Size 8x8

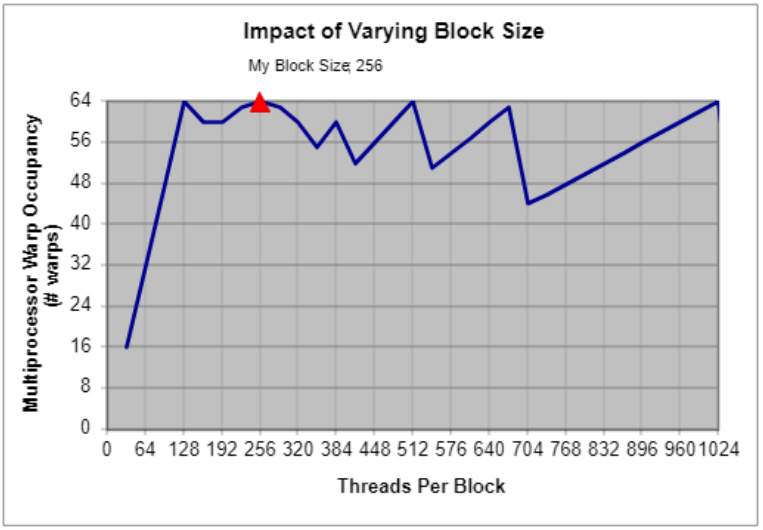


Warp Occupancy depending on Shared Memory Usage Per Block - Naive Υλοποίηση - Block Size 8x8

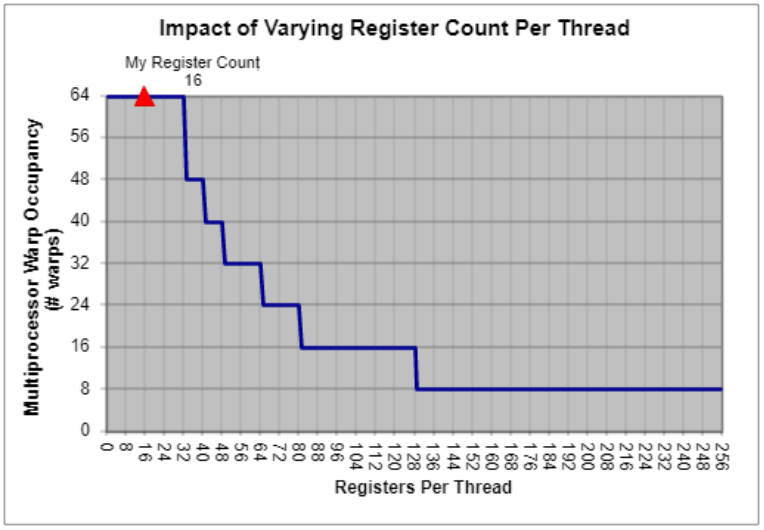
- Block Size : 16x16 = 256 threads
  Registers Per Thread : 16
  User Shared Memory Per Block : 0

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	8
Occupancy of each Multiprocessor	100%

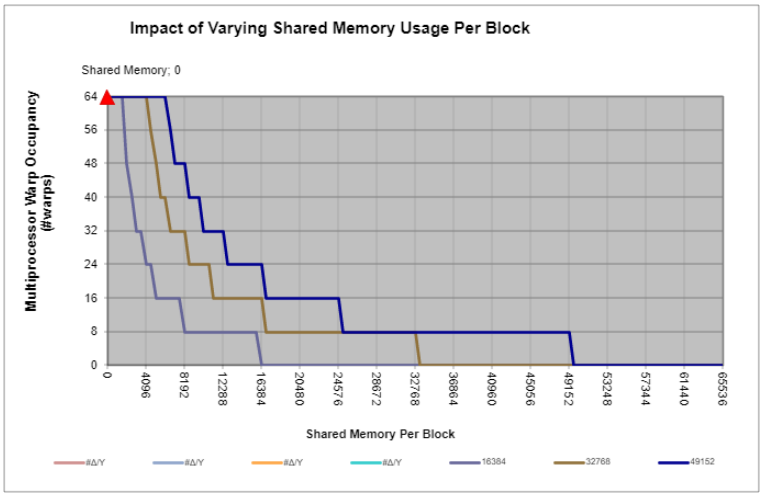
Χρησιμοποίηση πολυεπεργαστικών στοιχείων - Naive Υλοποίηση - Block Size 16x16



Warp Occupancy depending on Threads per Block - Naive Υλοποίηση - Block Size 16x16



Warp Occupancy depending on Registers per Thread - Naive Υλοποίηση - Block Size 16x16

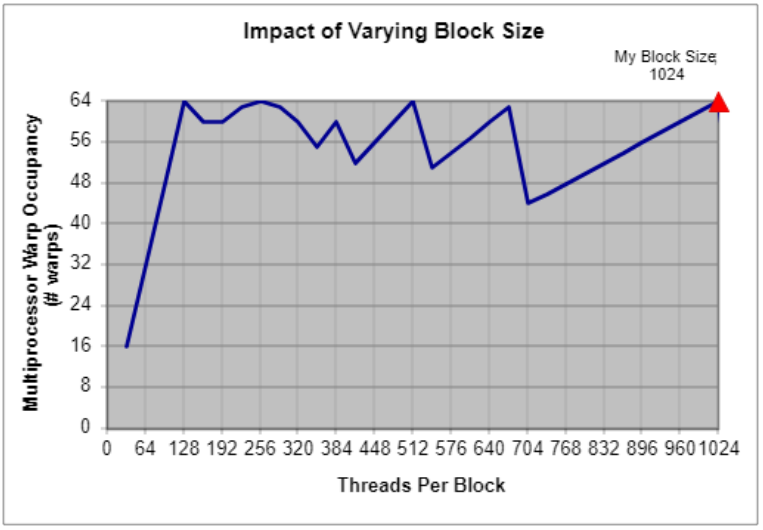


Warp Occupancy depending on Shared Memory Usage Per Block - Naive Υλοποίηση - Block Size 16x16

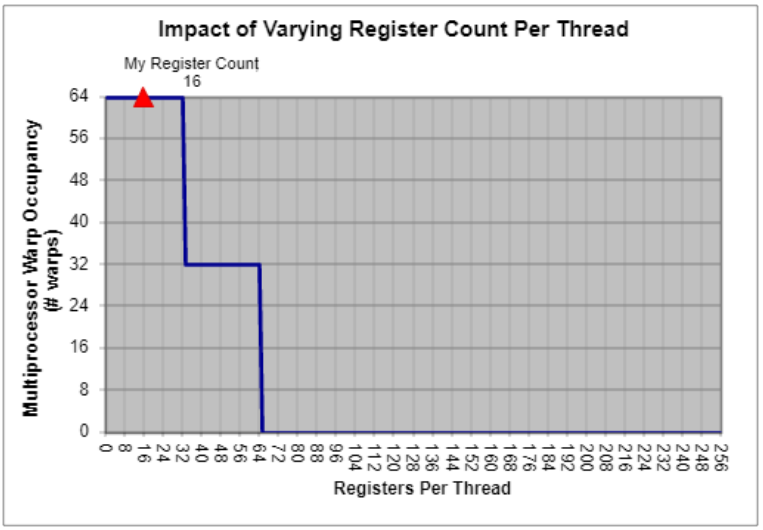
- Block Size : 32x32 = 1024 threads
  Registers Per Thread : 16
  User Shared Memory Per Block : 0

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	2
Occupancy of each Multiprocessor	100%

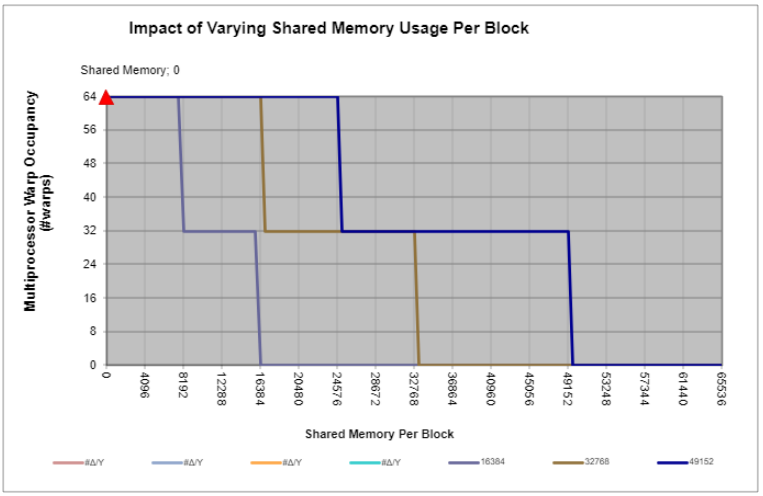
Χρησιμοποίηση πολυεπεργαστικών στοιχείων - Naive Υλοποίηση - Block Size 32x32



Warp Occupancy depending on Threads per Block - Naive Υλοποίηση - Block Size 32x32



Warp Occupancy depending on Registers per Thread - Naive Υλοποίηση - Block Size 32x32



Warp Occupancy depending on Shared Memory Usage Per Block - Naive Υλοποίηση - Block Size 32x32

Παρατηρούμε ότι το ποσοστό του occupancy αυξάνεται με την αύξηση του μεγέθους του block, φτάνοντας στο 100% ήδη από το 16x16.

Συνένωση των προσβάσεων στην κύρια μνήμη

Στη συνέχεια, τροποποιήθηκε ο βασικός κώδικας ο οποίος παρουσιάστηκε παραπάνω ώστε να επιτευχθεί συνένωση των προσβάσεων για τον πίνακα A. Για να γίνει αυτό θα πρέπει τμηματικά στοιχεία του πίνακα να μεταφέρονται στην τοπική μνήμη των πολυεπεξεργαστικών στοιχείων (shared memory).

Ο κώδικας παρατίθεται παρακάτω:

```
__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) {

    __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;

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

    value_t sum = 0;

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

        __syncthreads();

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

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

1. Ο πίνακας A, όπως είδαμε προηγουμένως έκανε K προσβάσεις στη μνήμη, μία για κάθε στοιχείο της γραμμής που επεξεργάζεται το thread. Οι K αυτές προσβάσεις γίνονται $M \cdot N$ φορές, αφού ο τελικός πίνακας C έχει διαστάσεις $M\,x\,N$. Φέρνοντας κάποια στοιχεία στην τοπική μνήμη μειώνουμε την ανάγκη για προσβάσεις και πλέον

για τον πίνακα A χρειάζονται $\frac{K \cdot M \cdot N}{TILE\_X \cdot TILE\_Y}$ προσβάσεις. Άρα οι προσβάσεις στη μνήμη μειώνονται κατά $\frac{K \cdot M \cdot N}{\frac{K \cdot M \cdot N}{TILE\_X \cdot TILE\_Y}} = TILE\_X \cdot TILE\_Y$.

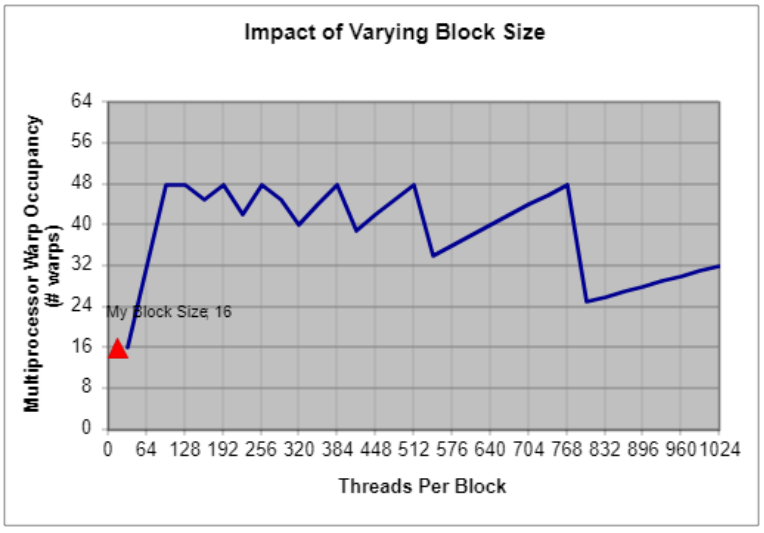
2. Προηγουμένως στο εσωτερικό loop είχαμε 2 προσβάσεις στη μνήμη, μία από τον πίνακα A και μία για τον πίνακα B, το οποίο αλλάζει καθώς με την συνένωση για τον πίνακα A έχουμε μόνο μια πρόσβαση για τον πίνακα B. Όπως και πριν, έχουμε ακόμα 2 floating point operations, οπότε συνολικά έχουμε $\frac{2\ flops}{1 \cdot 4\ bytes} = \frac{1}{2}\frac{flops}{byte}$. Παρόμοια με την naive υλοποίηση και εδώ η υλοποίηση μας είναι memory-bound αφού έχουμε 0.5 $\frac{flops}{byte}$, σημείο το οποίο βρίσκεται πιο αριστερά του “γονάτου” στο roofline model .

3. Για να κάνουμε δοκιμές για τη χρησιμοποίηση των πολυεπεξεργαστικών στοιχείων θα πρέπει εκτός από το Block Size να μεταβάλουμε αντίστοιχα τα μεγέθη Registers Per Thread και User Shared Memory Per Block. Τις πληροφορίες αυτές μπορούμε να τις λάβουμε κατά το make στο log του αρχείου .err που ορίσαμε. Παρακάτω παρουσιάζονται τα αποτελέσματα για Block Size 4x4, 8x8, 16x16, 32x32, τιμές για τις οποίες έγιναν και οι προσομοιώσεις αργότερα.

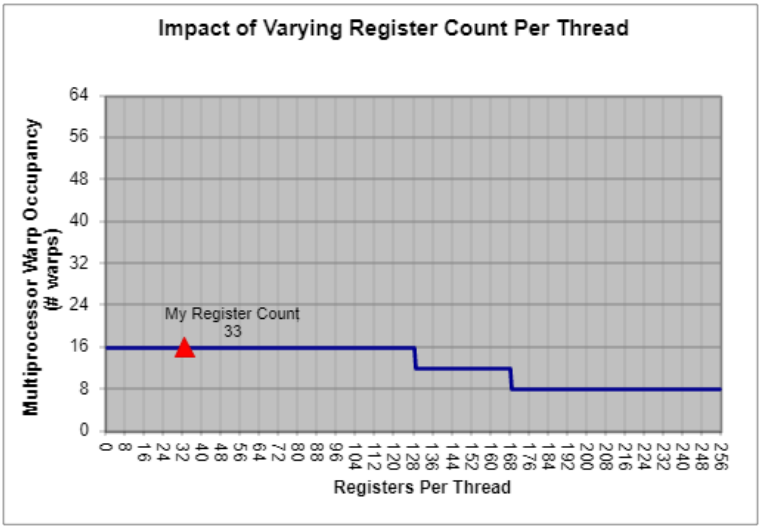
   - Block Size : 4x4 = 16 threads
     Registers Per Thread : 33
     User Shared Memory Per Block : 64

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	512
Active Warps per Multiprocessor	16
Active Thread Blocks per Multiprocessor	16
Occupancy of each Multiprocessor	25%

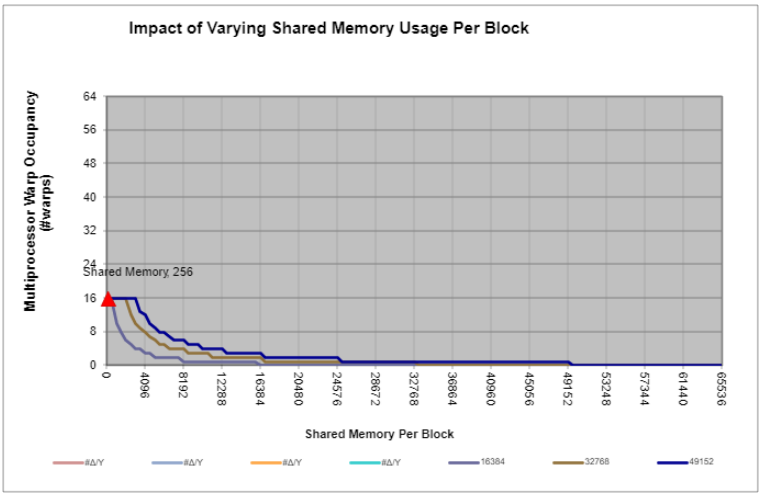
Χρησιμοποίηση πολυεπεργαστικών στοιχείων - Coalesced_A Υλοποίηση - Block Size 4x4



Warp Occupancy depending on Threads per Block - Coalesced_A Υλοποίηση - Block Size 4x4



Warp Occupancy depending on Registers per Thread - Coalesced_A Υλοποίηση - Block Size 4x4

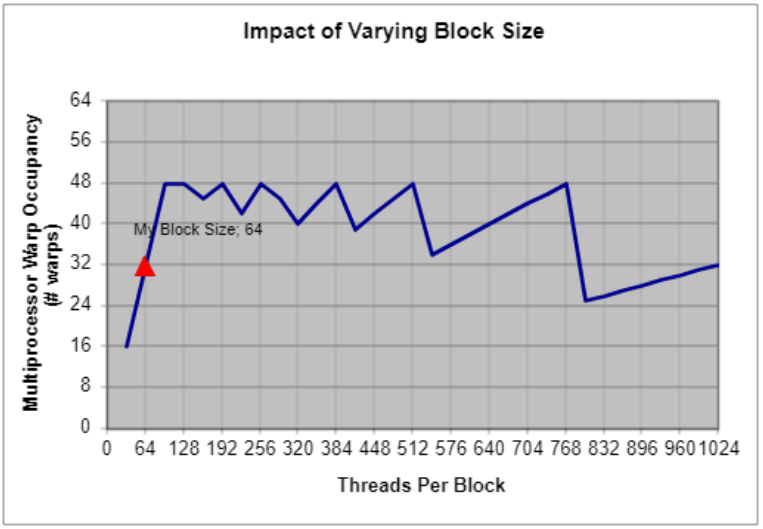


Warp Occupancy depending on Shared Memory Usage Per Block - Coalesced_A Υλοποίηση - Block Size 4x4

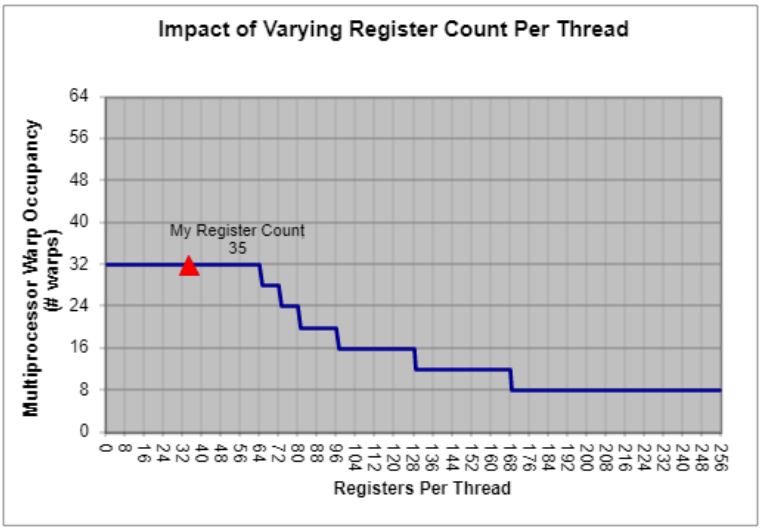
- Block Size : 8x8 = 64 threads
  Registers Per Thread : 35
  User Shared Memory Per Block : 256

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	16
Occupancy of each Multiprocessor	50%

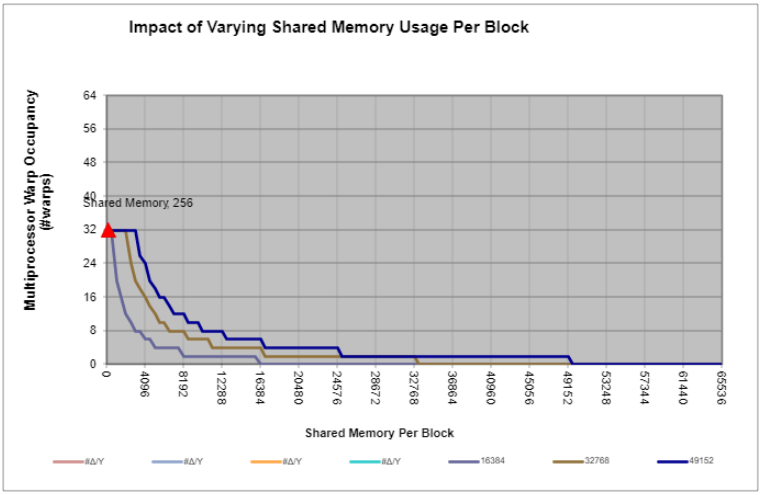
Χρησιμοποίηση πολυεπεργαστικών στοιχείων - Coalesced_A Υλοποίηση - Block Size 8x8



Warp Occupancy depending on Threads per Block - Coalesced_A Υλοποίηση - Block Size 8x8



Warp Occupancy depending on Registers per Thread - Coalesced_A Υλοποίηση - Block Size 8x8



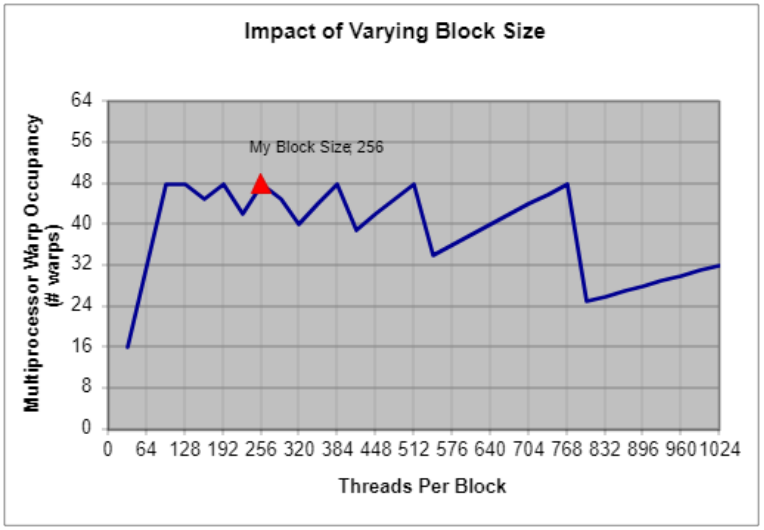
Warp Occupancy depending on Shared Memory Usage Per Block - Coalesced_A Υλοποίηση - Block Size 8x8

- Block Size : 16x16 = 256 threads
  Registers Per Thread : 37

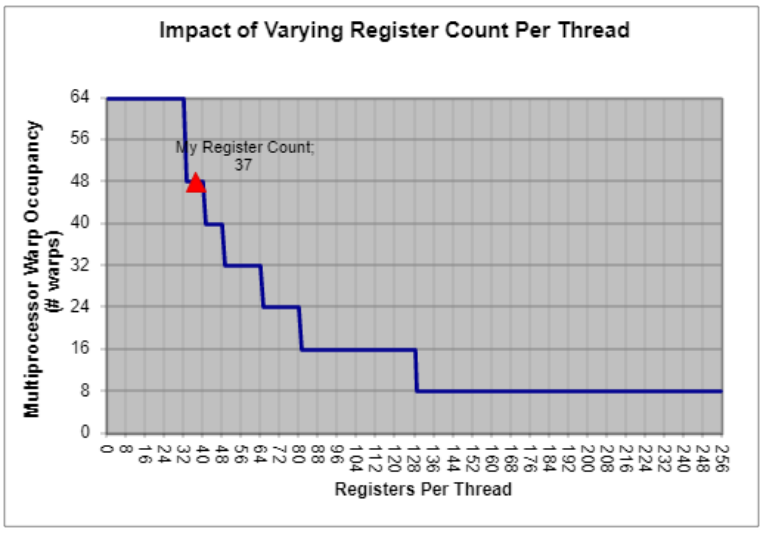
User Shared Memory Per Block : 1024

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1536
Active Warps per Multiprocessor	48
Active Thread Blocks per Multiprocessor	6
Occupancy of each Multiprocessor	75%

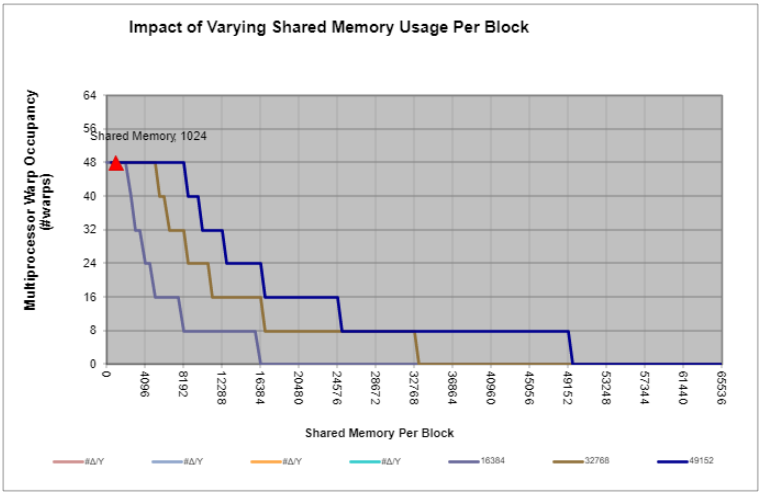
Χρησιμοποίηση πολυεπεργαστικών στοιχείων - Coalesced_A Υλοποίηση - Block Size 16x16



Warp Occupancy depending on Threads per Block - Coalesced_A Υλοποίηση - Block Size 16x16



Warp Occupancy depending on Registers per Thread - Coalesced_A Υλοποίηση - Block Size 16x16

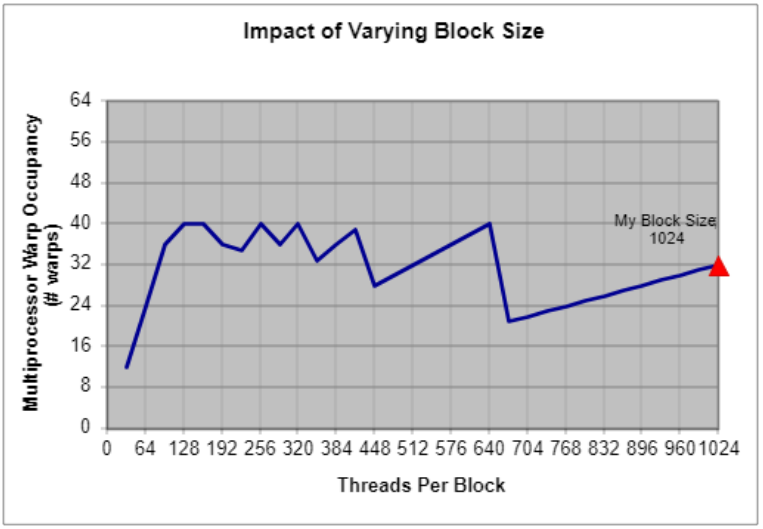


Warp Occupancy depending on Shared Memory Usage Per Block - Coalesced_A Υλοποίηση - Block Size 16x16

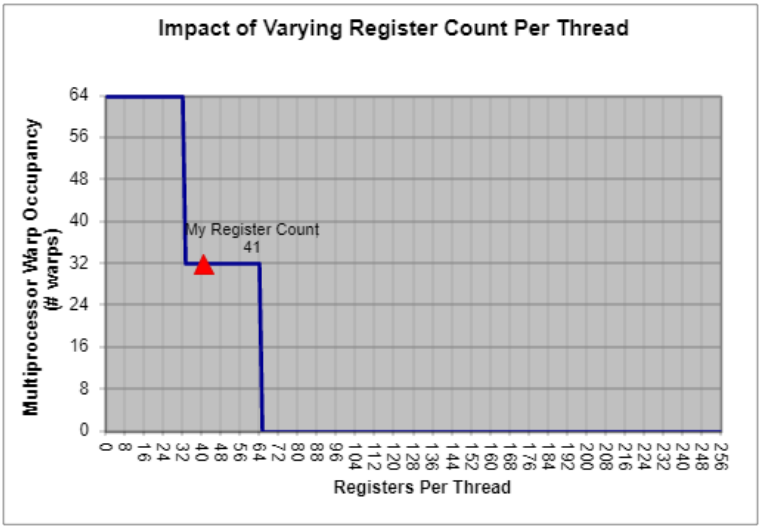
- Block Size : 32x32 = 1024 threads
  Registers Per Thread : 41
  User Shared Memory Per Block : 4096

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	1
Occupancy of each Multiprocessor	50%

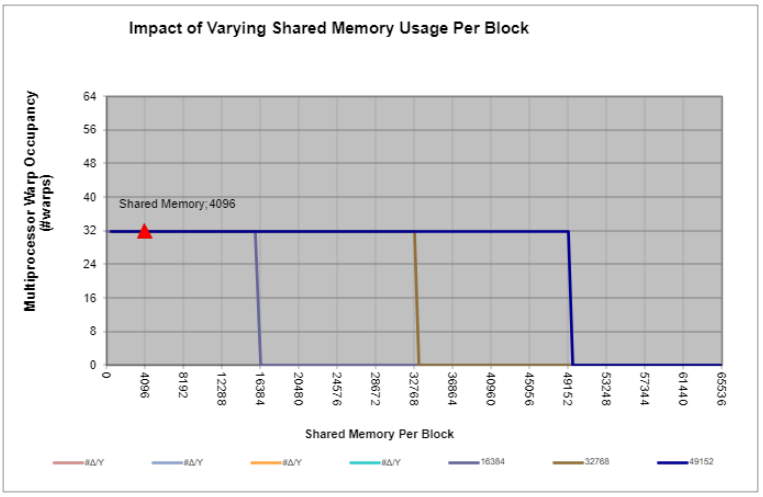
Χρησιμοποίηση πολυεπεργαστικών στοιχείων - Coalesced_A Υλοποίηση - Block Size 32x32



Warp Occupancy depending on Threads per Block - Coalesced_A Υλοποίηση - Block Size 32x32



Warp Occupancy depending on Registers per Thread - Coalesced_A Υλοποίηση - Block Size 32x32



Warp Occupancy depending on Shared Memory Usage Per Block - Coalesced_A Υλοποίηση - Block Size 32x32

Παρατηρούμε ότι το ποσοστό του occupancy αυξάνεται με την αύξηση του μεγέθους του block, φτάνοντας όμως μόνο μέχρι το 75% στο block μεγέθους 16x16, ενώ στο 32x32 παρατηρείται μείωση της πίσω στο 50%.

Μείωση των προσβάσεων στην κύρια μνήμη

Για να πετύχουμε περαιτέρω μείωση των προσβάσεων στη μνήμη τροποποίησαμε τον προηγούμενο κώδικα ώστε να φορτώνουμε στην shared memory και κομμάτια του πίνακα B.

Ο τροποποιημένος κώδικας φαίνεται παρακάτω:

```
__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) {

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

    value_t sum = 0;

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

        __syncthreads();

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

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

1. Πλέον τόσο ο πίνακας A όσο και ο πίνακας B κάνουν $\frac{K}{TILE\_X \cdot \_TILE\_Y}$ προσβάσεις, $M \cdot N$ φορές, άρα έχουμε συνολικά $\frac{2 \cdot K \cdot M \cdot N}{TILE\_X \cdot \_TILE\_Y}$. Συγκριτικά με τη naive υλοποίηση, έχουμε μείωση των προσβάσεων κατά $\frac{2 \cdot K \cdot M \cdot N}{\frac{2 \cdot K \cdot M \cdot N}{TILE\_X \cdot \_TILE\_Y}} = TILE\_X \cdot \_TILE\_Y$.

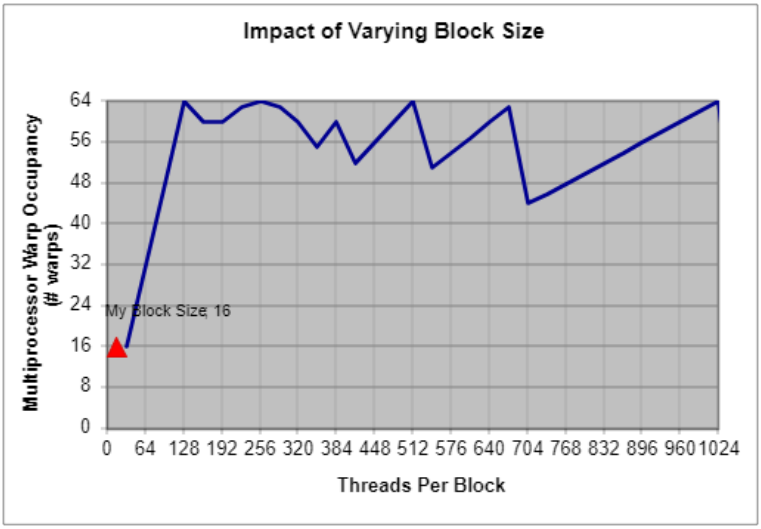
2. Με τις αλλαγές που κάναμε, δεν γίνονται πλέον προσβάσεις στη μνήμη αφού φέρνουμε τα δεδομένα που χρειαζόμαστε στην shared memory. Οπότε πλέον η υλοποίηση δεν είναι memory-bound αλλά είναι compute-bound.

3. Για να κάνουμε δοκιμές για τη χρησιμοποίηση των πολυεπεξεργαστικών στοιχείων θα πρέπει εκτός από το block size να μεταβάλουμε αντίστοιχα τα μεγέθη Registers Per Thread και User Shared Memory Per Block. Τις πληροφορίες αυτές μπορούμε να τις λάβουμε κατά το make στο log του αρχείου .err που ορίσαμε. Παρακάτω παρουσιάζονται τα αποτελέσματα για Block Size 4x4, 8x8, 16x16, 32x32, τιμές για τις οποίες έγιναν και οι προσομοιώσεις αργότερα.

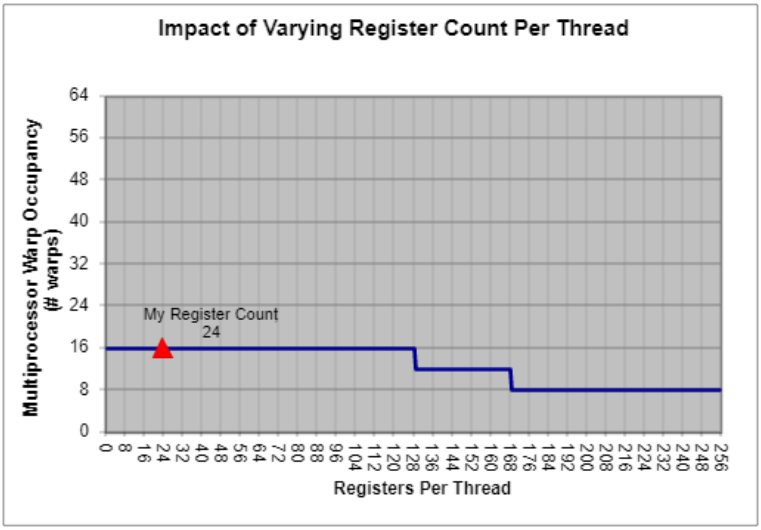
    - Block Size : 4x4 = 16 threads
      Registers Per Thread : 24
      User Shared Memory Per Block : 128

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	512
Active Warps per Multiprocessor	16
Active Thread Blocks per Multiprocessor	16
Occupancy of each Multiprocessor	25%

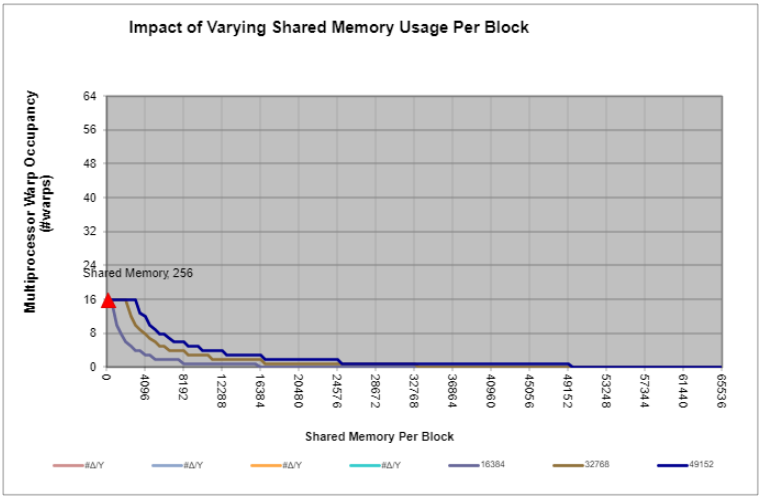
Χρησιμοποίηση πολυεπεργαστικών στοιχείων - reduced_global Υλοποίηση - Block Size 4x4



Warp Occupancy depending on Threads per Block - reduced_global Υλοποίηση - Block Size 4x4



Warp Occupancy depending on Registers per Thread - reduced_global Υλοποίηση - Block Size 4x4

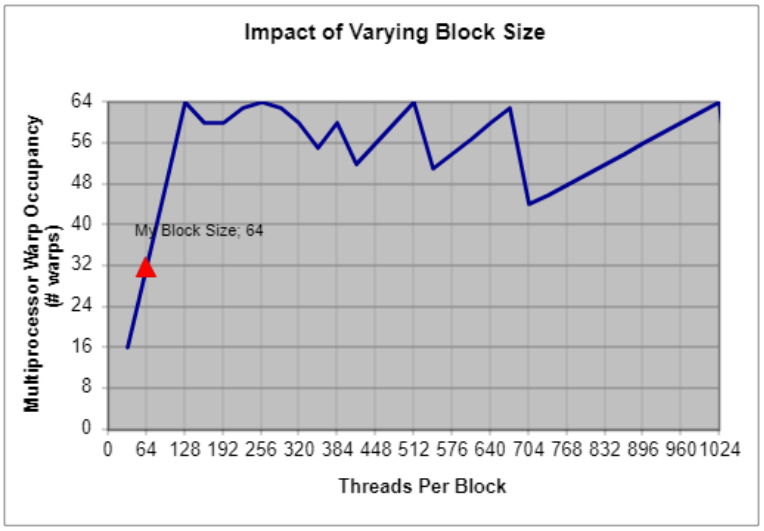


Warp Occupancy depending on Shared Memory Usage Per Block - reduced_global Υλοποίηση - Block Size 4x4

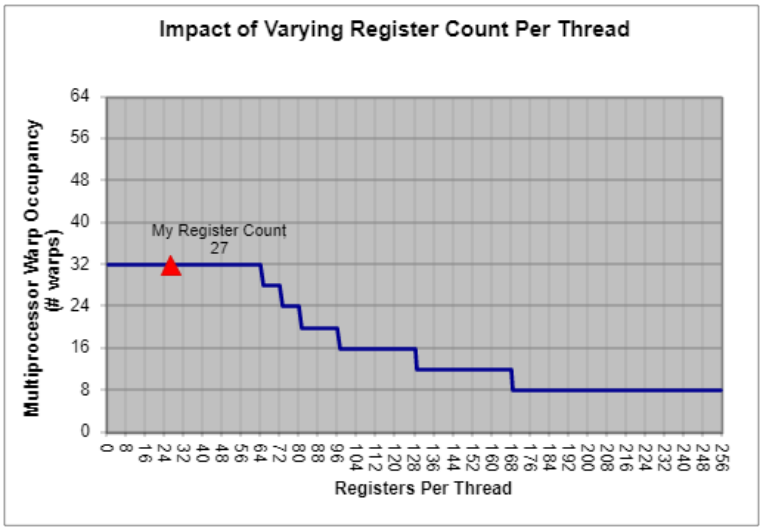
- Block Size : 8x8 = 64 threads
  Registers Per Thread : 27
  User Shared Memory Per Block : 512

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	16
Occupancy of each Multiprocessor	50%

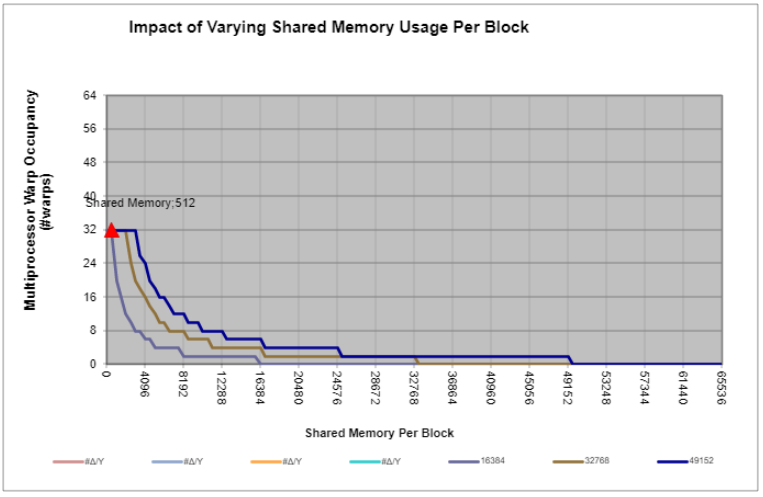
Χρησιμοποίηση πολυεπεργαστικών στοιχείων - reduced_global Υλοποίηση - Block Size 8x8



Warp Occupancy depending on Threads per Block - reduced_global Υλοποίηση - Block Size 8x8



Warp Occupancy depending on Registers per Thread - reduced_global Υλοποίηση - Block Size 8x8

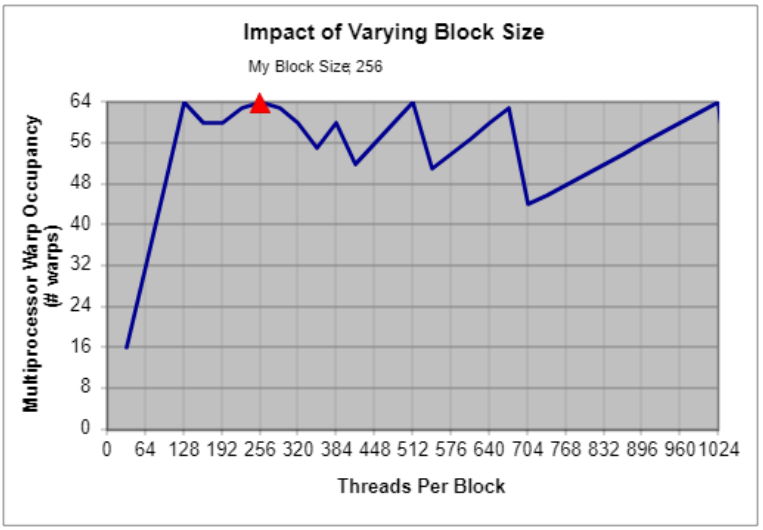


Warp Occupancy depending on Shared Memory Usage Per Block - reduced_global Υλοποίηση - Block Size 8x8

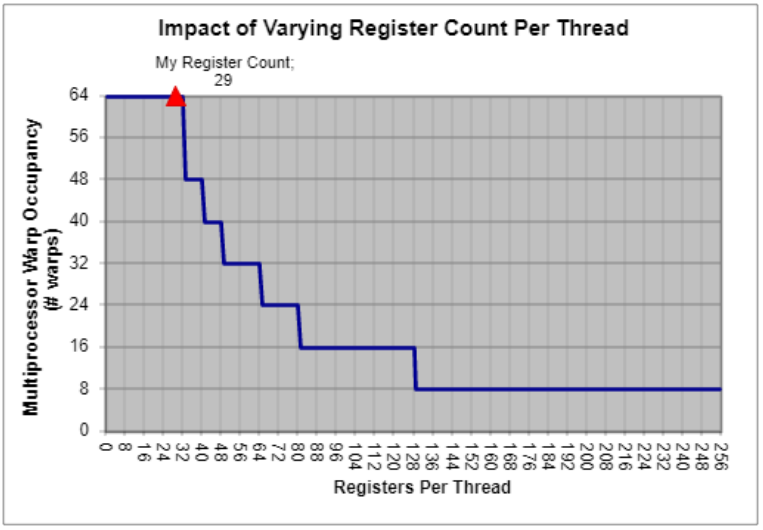
- Block Size : 16x16 = 256 threads
  Registers Per Thread : 29
  User Shared Memory Per Block : 2048

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	8
Occupancy of each Multiprocessor	100%

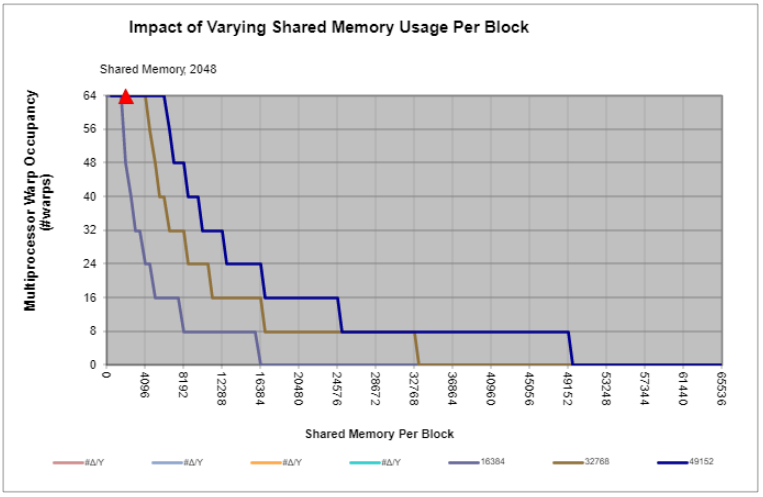
Χρησιμοποίηση πολυεπεργαστικών στοιχείων - reduced_global Υλοποίηση - Block Size 16x16



Warp Occupancy depending on Threads per Block - reduced_global Υλοποίηση - Block Size 16x16



Warp Occupancy depending on Registers per Thread - reduced_global Υλοποίηση - Block Size 16x16

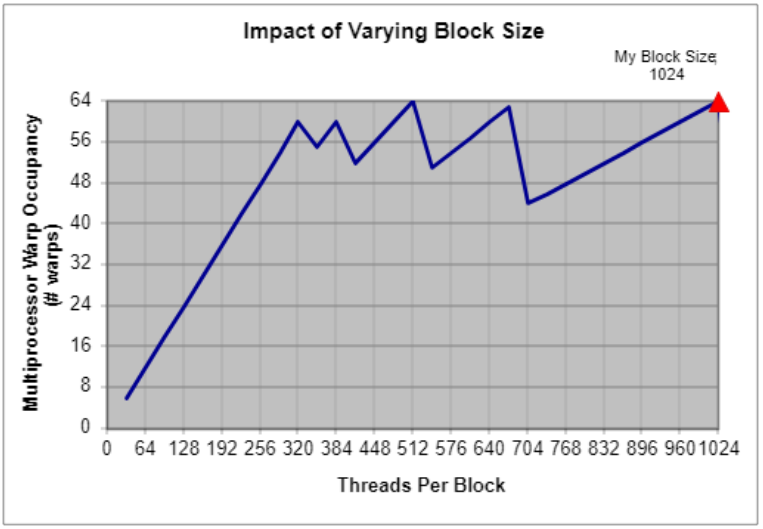


Warp Occupancy depending on Shared Memory Usage Per Block - reduced_global Υλοποίηση - Block Size 16x16

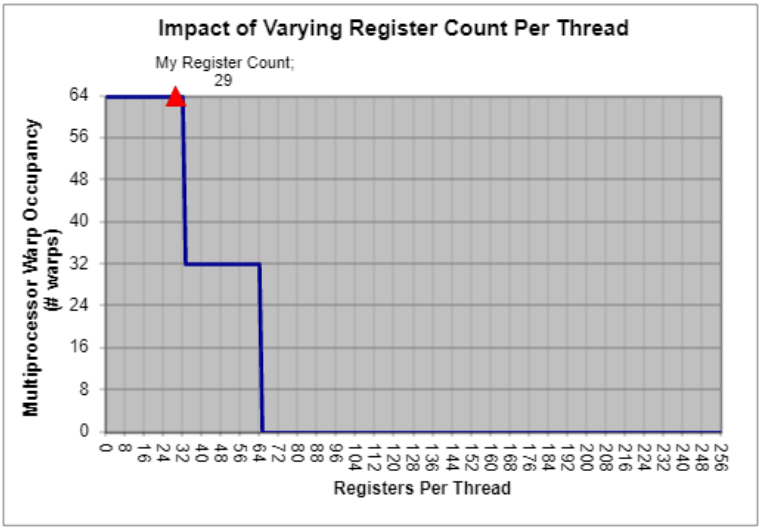
- Block Size : 32x32 = 1024 threads
  Registers Per Thread : 29
  User Shared Memory Per Block : 8192

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	2
Occupancy of each Multiprocessor	100%

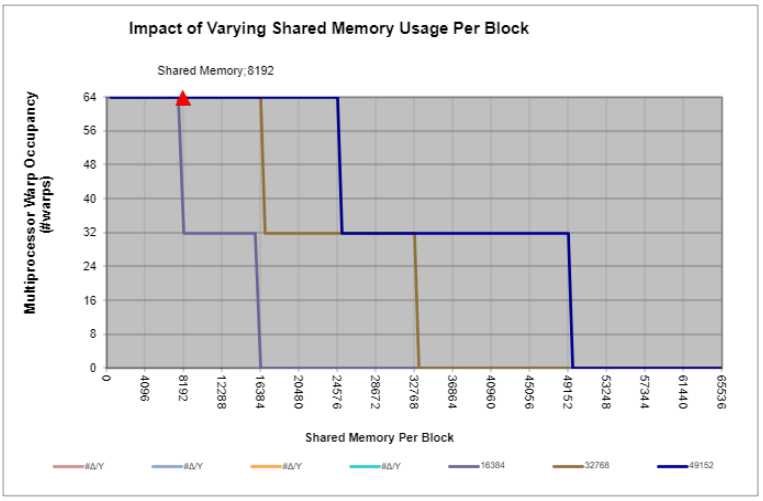
Χρησιμοποίηση πολυεπεργαστικών στοιχείων - reduced_global Υλοποίηση - Block Size 32x32



Warp Occupancy depending on Threads per Block - reduced_global Υλοποίηση - Block Size 32x32



Warp Occupancy depending on Registers per Thread - reduced_global Υλοποίηση - Block Size 32x32



Warp Occupancy depending on Shared Memory Usage Per Block - reduced_global Υλοποίηση - Block Size 32x32

Παρατηρούμε ότι το ποσοστό του occupancy αυξάνεται με την αύξηση του μεγέθους του block, φτάνοντας στο 100% ήδη από το 16x16.

Χρήση της βιβλιοθήκης cuBLAS

Στη συνέχεια, χρησιμοποιήθηκε η συνάρτηση cublasSgemm() της βιβλιοθήκης cuBLAS. Η βιβλιοθήκη αυτή είναι μία επέκταση της BLAS, η οποία διαθέτει συναρτήσεις για τις υλοποιήσεις διαφόρων πράξεων γραμμικής άλγεβρας. Παρακάτω παρατίθεται ο κώδικας ο οποίος χρησιμοποιήθηκε:

```
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) {

    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);
}
```

Η βιβλιοθήκη cuBLAS είναι υλοποιημένη με τέτοιο τρόπο ώστε να θεωρεί πως τα στοιχεία ενός πίνακα είναι αποθηκευμένα σε column-major format στη μνήμη. Καθώς εμείς θεωρούμε πως τα στοιχεία μας είναι αποθηκευμένα σε row-major format, θα πρέπει να τροποποιήσουμε ανάλογα την κλήση της συνάρτησης στο αρχείο dmm_main.cu. Συγκεκριμένα έχουμε:

```
  cudaEventRecord(start);
  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);
  } else {
    for (size_t i = 0; i < NR_ITER; ++i)
      gpu_kernels[kernel].fn<<<gpu_grid, gpu_block>>>(gpu_A,
                  gpu_B,
                  gpu_C,
                  M, N, K);
```

```
  }
```

Στην ουσία, αυτό το οποίο κάνουμε είναι ότι δίνουμε τους πίνακες A και B στην συνάρτηση με διαφορετική σειρά. Έτσι, η cublasSgemm() θα χρησιμοποιήσει τον πίνακα B σε row-major format και τον πίνακα A σε column-major format οπότε θα λάβουμε ως τελικό αποτέλεσμα τον πίνακα C σε row-major format.

4. Πειράματα και μετρήσεις επιδόσεων

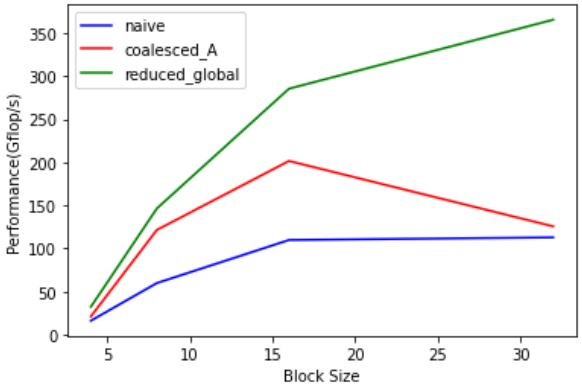
4.1 Σενάριο μετρήσεων και διαγράμματα

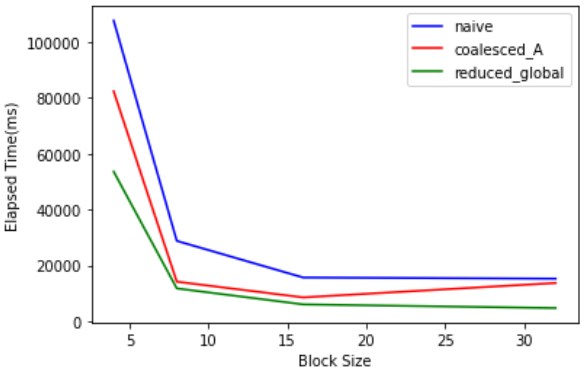
Μελέτη της επίδρασης του μεγέθους του μπλοκ νημάτων/υπολογισμού στην επίδοση των υλοποιήσεών

Για να είμαστε εντός των προδιαγραφών για την κάρτα γραφικών που χρησιμοποιούμε θέλουμε το Block να έχει το μέγιστο 1032 νήματα. Έτσι, τα διαθέσιμα Block Sizes για πειραματισμό, υποθέτοντας ότι THREAD_BLOCK_X = THREAD_BLOCK_Y = TILE_X = TILE_Y, είναι 4x4, 8x8, 16x16, 32x32. Το script που χρησιμοποιήθηκε για τη λήψη των μετρήσεων παρατίθεται παρακάτω:

```
#!/bin/bash

#PBS -o scenario1.out
#PBS -e scenario1.err
#PBS -l walltime=06:00:00
#PBS -l nodes=dungani:ppn=24:cuda

export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
export CUDA_VISIBLE_DEVICES=2

gpu_kernels="0 1 2"
problem_sizes="256 512 1024 2048"
block_sizes="4 8 16 32"
gpu_prog="./dmm_main"

cd /home/parallel/parlab20/a4/dmm-skeleton/cuda
echo "Benchmark started on $(date) in $(hostname)"
for i in $gpu_kernels; do
  for b in $block_sizes; do
      make -s clean
      make -s THREAD_BLOCK_X=$b THREAD_BLOCK_Y=$b TILE_X=$b TILE_Y=$b DEBUG=0
CHECK=0
      GPU_KERNEL=$i $gpu_prog 2048 2048 2048
  done
done
echo "Benchmark ended on $(date) in $(hostname)"
```

Τα αποτελέσματα των μετρήσεων συγκεντρώνονται στα παρακάτω γράφηματα:





Παρατηρούμε ότι την καλύτερη επίδοση την πετυχαίνει η reduced_global έκδοση, τόσο έχοντας καλύτερο performance (Gflops/s) σε κάθε Block Size αλλά και πετυχαίνοντας σταθερά τους χαμηλότερους χρόνους. Παρόλα αυτά αξίζει να σημειωθεί πώς η naive έκδοση, αν και σημαντικά πιο αργή και λιγότερο αποδοτική βελτιώνεται με την αύξηση του Block Size. Η coalesced_A έκδοση πετυχαίνει μια ενδιάμεση επίδοση, η οποία όμως σε αντίθεση με τις άλλες δύο υλοποιήσεις, δεν είναι σταθερά βελτιούμενη. Συγκεκριμένα, η καλύτερη απόδοση και ο καλύτερος χρόνος επιτυγχάνονται για 16x16 Block Size και υπάρχει πτώση στην περίπτωση 32x32. Η μείωση αυτή είχε προβλεφθεί νωρίτερα από τον CUDA Occupancy Calculator, ο οποίος για την συγκεκριμένη έκδοση είχε υπολογίσει μείωση της χρησιμοποιησιμότητας για Block Size 32x32.

Σύγκριση της επίδοσης όλων των εκδόσεων του πυρήνα DMM

Για τη λήψη των μετρήσεων, θεωρήθηκε ως βέλτιστο Block Size, το μέγεθος που απέδωσε τις καλύτερες μετρήσεις για την κάθε υλοποίηση κατά τα προηγούμενα πειράματα :

- 32x32 για την naive υλοποίηση
- 16x16 για την coalesced_A υλοποίηση
- 32x32 για τη reduced_global υλοποίηση

Όπως και προηγουμένως, υποτέθηκε ότι THREAD_BLOCK_X = THREAD_BLOCK_Y = TILE_X = TILE_Y. Το script που χρησιμοποιήθηκε για τη λήψη των μετρήσεων παρατίθεται παρακάτω:

```
#!/bin/bash

#PBS -o scenario2.out
#PBS -e scenario2.err
#PBS -l walltime=06:00:00
#PBS -l nodes=dungani:ppn=24:cuda

export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
export CUDA_VISIBLE_DEVICES=2
```

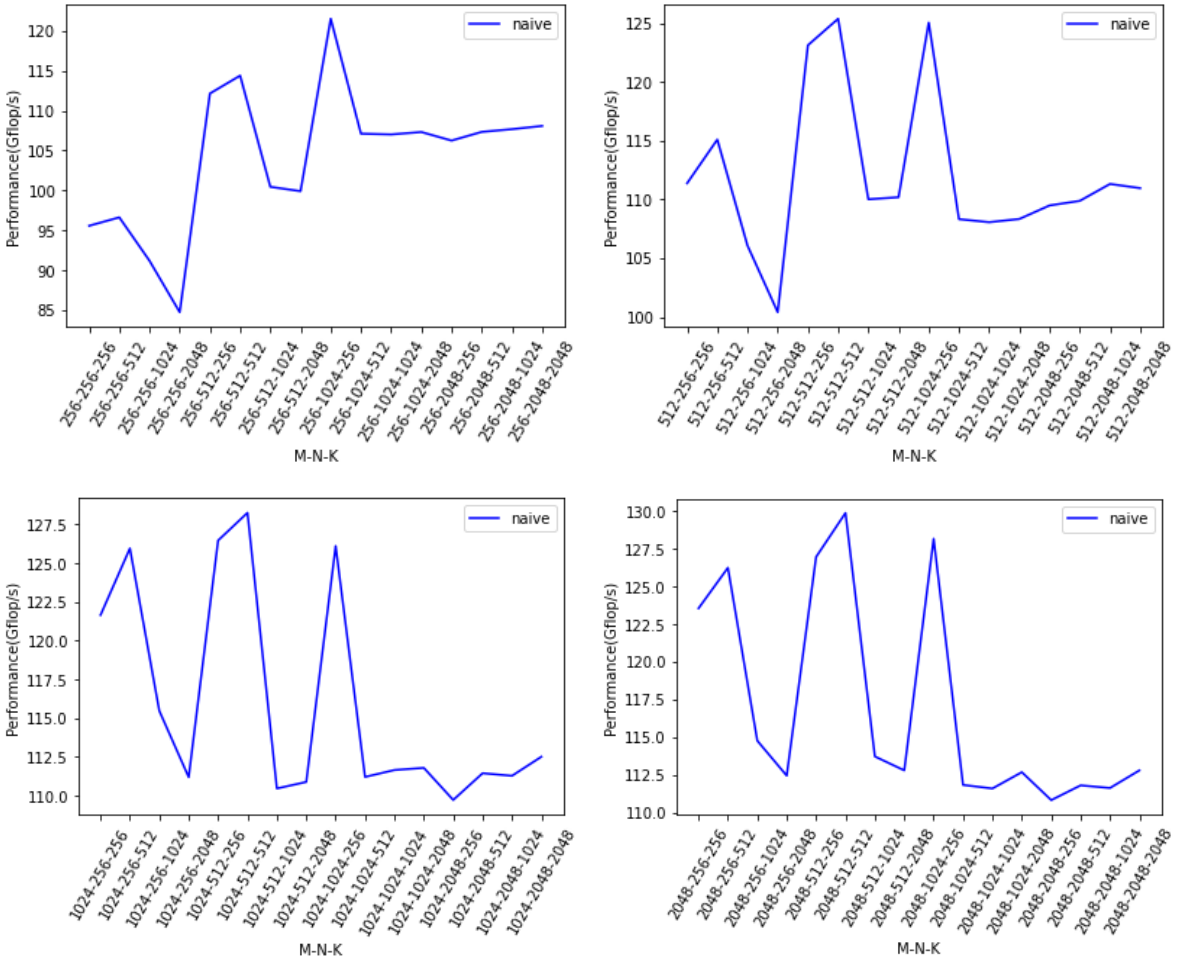
```
gpu_kernels="0 1 2"
problem_sizes="256 512 1024 2048"
block_sizes=(32 16 32)
gpu_prog="./dmm_main"

cd /home/parallel/parlab20/a4/dmm-skeleton/cuda
echo "Benchmark started on $(date) in $(hostname)"
for i in $gpu_kernels; do
  for m  in $problem_sizes; do
    for n in $problem_sizes; do
      for k in $problem_sizes; do
          b = ${block_sizes[$i]}
          make -s clean
                make  -s  THREAD_BLOCK_X=$b  THREAD_BLOCK_Y=$b  TILE_X=$b
TILE_Y=$b DEBUG=0 CHECK=0
          GPU_KERNEL=$i $gpu_prog $m $n $k
      done
    done
  done
done
echo "Benchmark ended on $(date) in $(hostname)"
```

Για την έκδοση cuBLAS χρησιμοποιήθηκε Block Size 32x32 και χρησιμοποιήθηκε, ξανά, THREAD_BLOCK_X = THREAD_BLOCK_Y = TILE_X = TILE_Y. Το script που χρησιμοποιήθηκε για τη λήψη των μετρήσεων παρατίθεται παρακάτω:

```
#!/bin/bash

#PBS -o scenario_cublas.out
#PBS -e scenario_cublas.err
#PBS -l walltime=06:00:00
#PBS -l nodes=dungani:ppn=24:cuda

export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
export CUDA_VISIBLE_DEVICES=2

gpu_kernels="3"
problem_sizes="256 512 1024 2048"
block_sizes=(32)
gpu_prog="./dmm_main"

cd /home/parallel/parlab20/a4/dmm-skeleton/cuda
```

```
echo "Benchmark started on $(date) in $(hostname)"
for i in $gpu_kernels; do
  for m  in $problem_sizes; do
    for n in $problem_sizes; do
      for k in $problem_sizes; do
          b = ${block_sizes[$i]}
          make -s clean
                make  -s  THREAD_BLOCK_X=$b  THREAD_BLOCK_Y=$b  TILE_X=$b
TILE_Y=$b DEBUG=0 CHECK=0
          GPU_KERNEL=$i $gpu_prog $m $n $k
      done
    done
  done
done
echo "Benchmark ended on $(date) in $(hostname)"
```

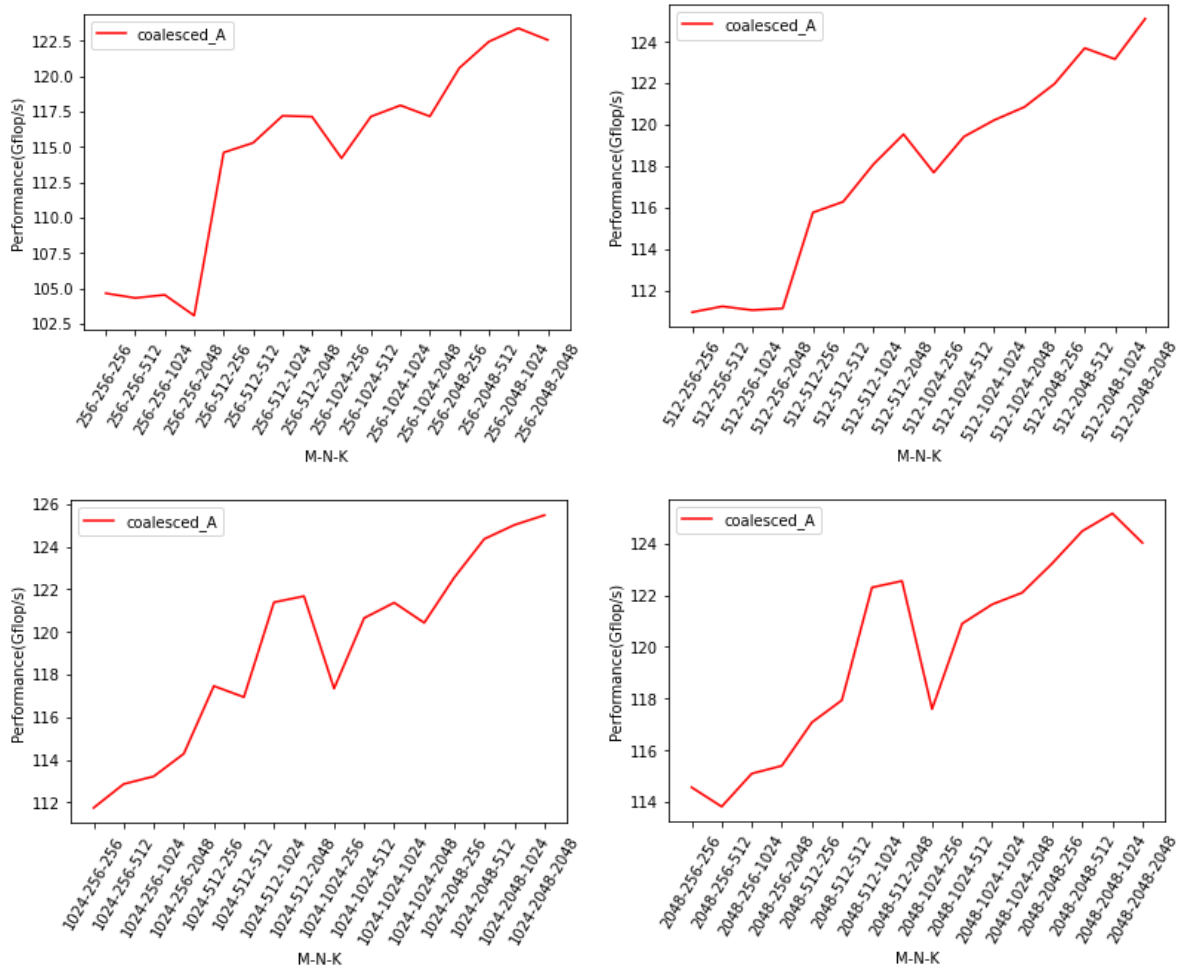
Naive



<u>Παρατηρήσεις</u>

Αρχικά, παρατηρείται μια “περιοδικότητα” στα αποτελέσματα όσον αφορά τα ανεβοκατεβάσματα στο Performance. Συγκεκριμένα, βλέπουμε ότι υπάρχει μείωση για μεγάλα K, η οποία οφείλεται στο γεγονός ότι γίνονται K προσβάσεις στη μνήμη τόσο από τον πίνακα A όσο και από τον B (2K σύνολο όπως είδαμε προηγουμένως). Ακόμα, βλέπουμε πως για μεγαλύτερα μεγέθη N και K, η επίδοση τείνει να σταθεροποιείται κοντά στα 110 $\frac{Gflops}{s}$. Τέλος, παρατηρούμε ότι για μικρότερες τιμές N και K έχουμε spikes στο performance.

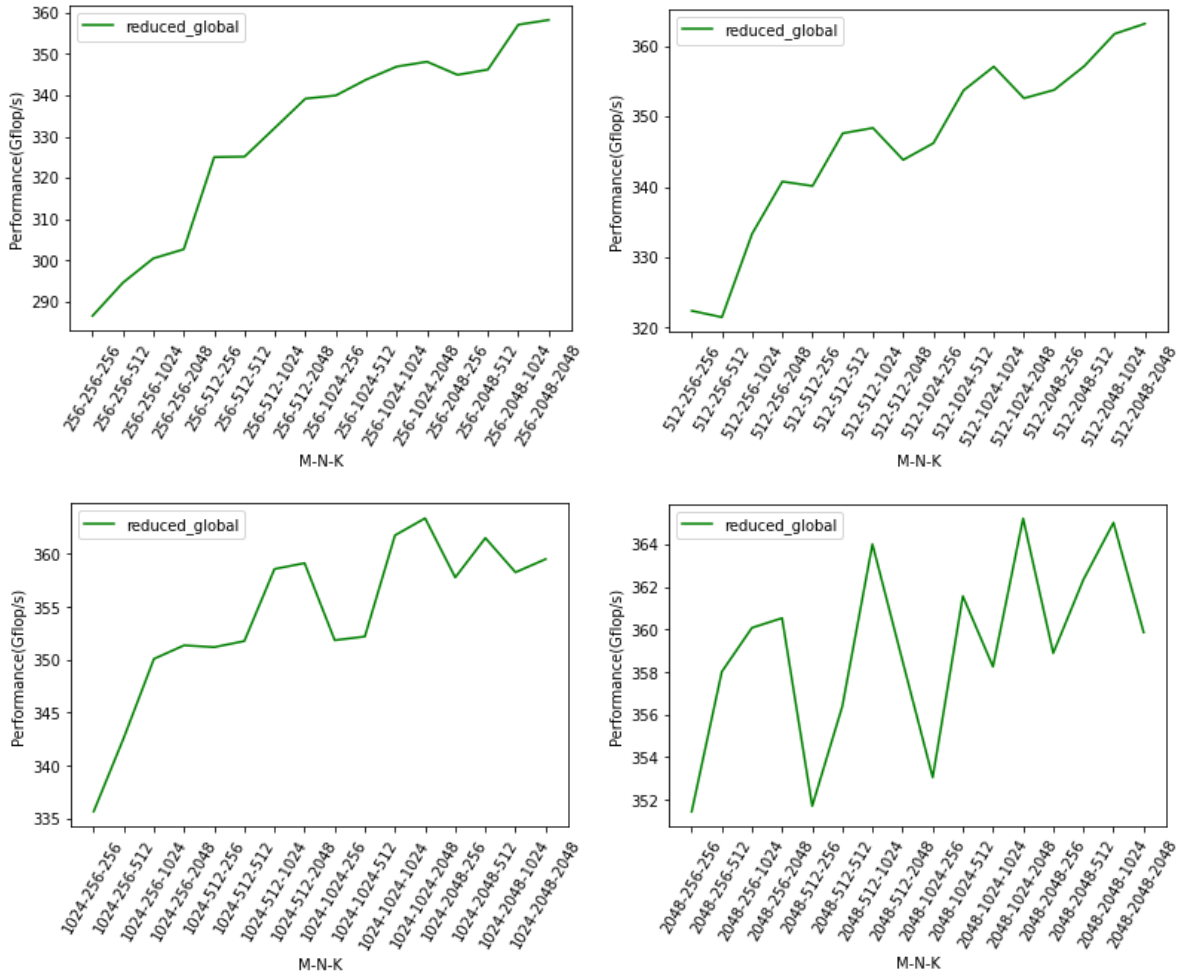
coalesced_A



<u>Παρατηρήσεις</u>

Στην coalesced_A έκδοση παρατηρούμε ότι έχουμε βελτίωση στο performance με τα τις χειρότερες περιπτώσεις να παρουσιάζουν επίδοση σημαντικά βελτιωμένη σε σχέση με την αντίστοιχη περίπτωση στη naive υλοποίηση. Η συνολική βελτίωση στην απόδοση αποδίδεται στην τμηματική φόρτωση του πίνακα A στην shared memory που σημαίνει πως πλέον γίνονται προσβάσεις στη μνήμη μόνο για τον πίνακα B. Τέλος, σε αντίθεση με τη naive υλοποίηση, παρατηρούμε ότι οι καλύτερες επιδόσεις παρατηρούνται για μεγάλες τιμές N και K και δεν έχουμε εξάρτηση την επίδοσης πλέον από το K.

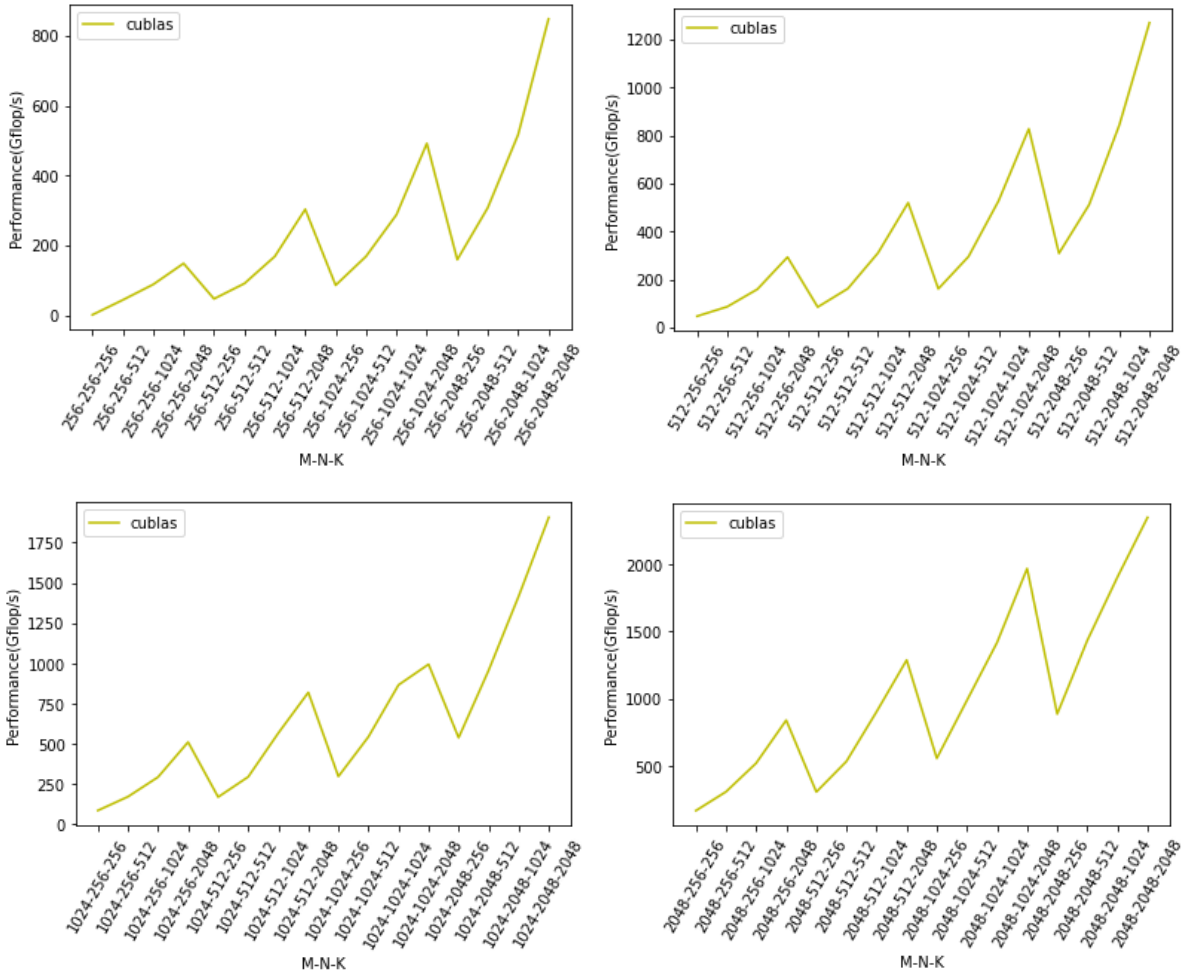
reduced_global



<u>Παρατηρήσεις</u>

Με τη συνολική μείωση των προσβάσεων βλέπουμε ακόμα μεγαλύτερη βελτίωση στα αποτελέσματα με τις χειρότερες περιπτώσεις να παρουσιάζουν επίδοση πολύ καλύτερη ακόμα και των βέλτιστων των προηγούμενων υλοποιήσεων. Παρατηρούμε ότι οι καλύτερες επιδόσεις επιτυγχάνονται για μεγάλες τιμές του N και του K ενώ οι καλύτερες τιμές συνολικά επιτυγχάνονται για M= 2048, ακόμα και αν σε σχέση με άλλες τιμές δεν παρατηρείται η ίδια αυξητική τάση στην επίδοση όσο μεγαλώνουν τα N και K.

cuBLAS

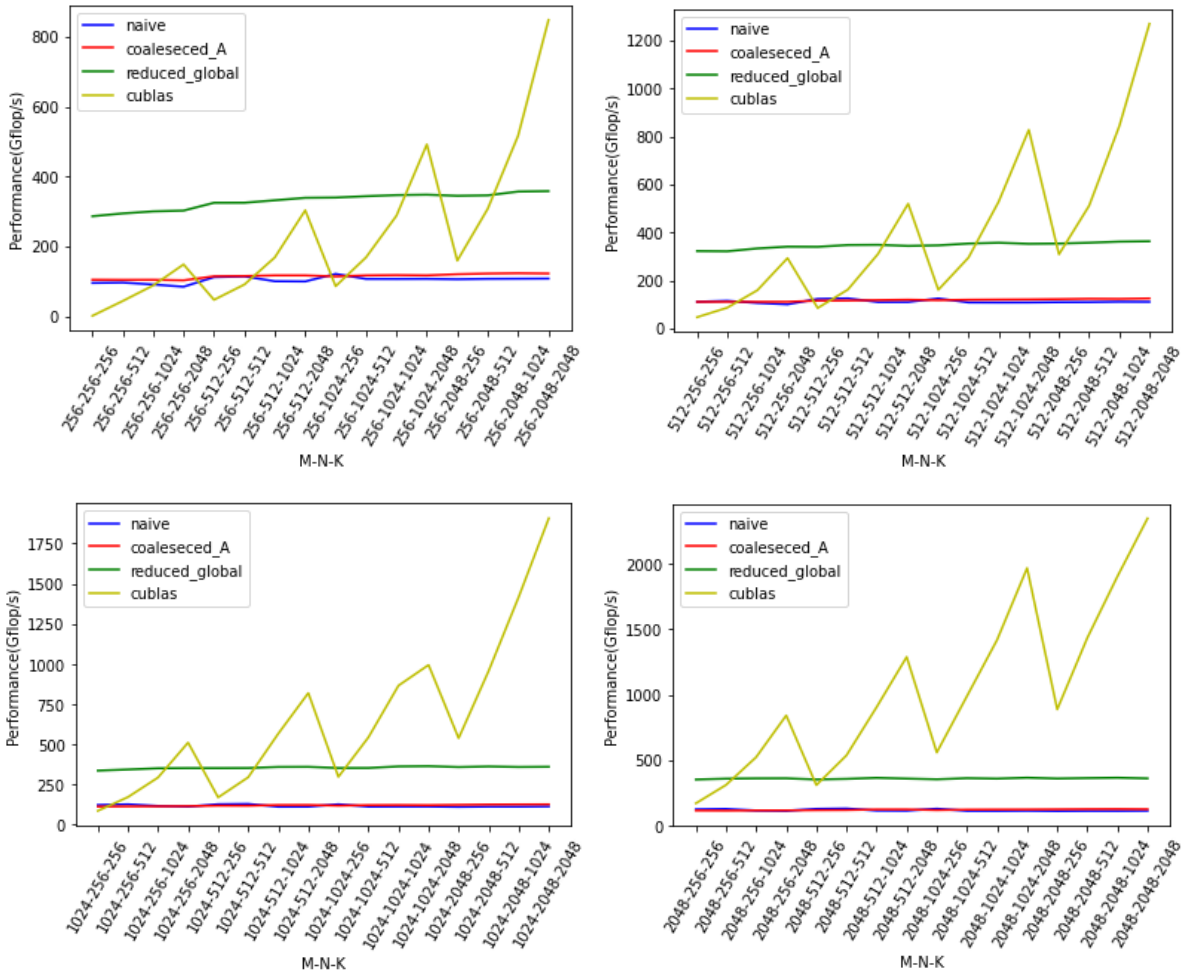


<u>Παρατηρήσεις</u>

Η υλοποίηση με τη χρήση της βιβλιοθήκης cuBLAS παρουσιάζει τις καλύτερες επιδόσεις στις περισσότερες περιπτώσεις σε σχέση με κάθε άλλη επίδοση. Παρατηρούμε ότι υπάρχει, όπως και στη naive υλοποίηση μία περιοδικότητα στον τρόπο με τον οποίο εμφανίζονται τα αποτελέσματα με τις καλύτερες επιδόσεις να επιτυγχάνονται για τις μεγαλύτερες τιμές των N και K. Για μικρά M και μικρές τιμές N και K παρατηρούνται επιδόσεις συγκρίσιμες ή και ελαφρώς χειρότερες από ότι είδαμε σε προηγούμενες υλοποιήσεις.

Συγκριτικά Αποτελέσματα

Παρακάτω, παρουσιάζονται τα παραπάνω αποτελέσματα μαζεμένα ώστε να είναι πιο εμφανείς οι συγκρίσεις μεταξύ των υλοποιήσεων.



Συνολικά, παρατηρούμε ότι οι υλοποιήσεις naive και coalesed_A έχουν παρόμοια συμπεριφορά και έχουν στη πλειάδα των περιπτώσεων τις χειρότερες επιδόσεις. Για μικρά M παρατηρούμε ότι η υλοποίηση reduced_global παρουσιάζει συνήθως τις καλύτερες επιδόσεις, με την υλοποίηση σε cuBLAS να παρουσιάζει συγκρίσιμα ή και καλύτερα αποτελέσματα για μεγαλύτερες τιμές των N και K. Τέλος, παρατηρούμε ότι η υλοποίηση cuBLAS παρουσιάζει ξεκάθαρα τα καλύτερα αποτελέσματα για μεγαλύτερα M στις πλειονότητα των περιπτώσεων και ιδιαίτερα για M = 2048 και μεγάλα N και K παρουσιάζει επιδόσεις αισθητά καλύτερες από οποιαδήποτε άλλη υλοποίηση.

Appendix

Παρακάτω παρατίθεται τα αποτελέσματα από το .err αρχείο στο οποίο φαίνονται τα Registers Per Thread και το Shared Memory για τις διαφορετικές διαστάσεις block για κάθε υλοποίηση.

Block Size 4x4

- coalesced_A

```
ptxas info    : Compiling entry function '_Z19dmm_gpu_coalesced_APKfS0_Pfmmm' for 'sm_35'
ptxas info    : Function properties for _Z19dmm_gpu_coalesced_APKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 33 registers, 64 bytes smem, 368 bytes cmem[0]
```

- naive

```
ptxas info    : Compiling entry function '_Z13dmm_gpu_naivePKfS0_Pfmmm' for 'sm_35'
ptxas info    : Function properties for _Z13dmm_gpu_naivePKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 16 registers, 368 bytes cmem[0]
```

- reduced_global

```
ptxas info    : Compiling entry function '_Z22dmm_gpu_reduced_globalPKfS0_Pfmmm' for 'sm_35'
ptxas info    : Function properties for _Z22dmm_gpu_reduced_globalPKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 24 registers, 128 bytes smem, 368 bytes cmem[0]
```

Block Size 8x8

- coalesced_A

```
ptxas info    : Compiling entry function '_Z19dmm_gpu_coalesced_APKfS0_Pfmmm' for 'sm_35'
ptxas info    : Function properties for _Z19dmm_gpu_coalesced_APKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 35 registers, 256 bytes smem, 368 bytes cmem[0]
```

- naive

```
ptxas info    : Compiling entry function '_Z13dmm_gpu_naivePKfS0_Pfmmm' for 'sm_35'
ptxas info    : Function properties for _Z13dmm_gpu_naivePKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 16 registers, 368 bytes cmem[0]
```

- reduced_global

```
ptxas info    : Compiling entry function '_Z22dmm_gpu_reduced_globalPKfS0_Pfmmm' for 'sm_35'
ptxas info    : Function properties for _Z22dmm_gpu_reduced_globalPKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 27 registers, 512 bytes smem, 368 bytes cmem[0]
```

Block Size 16x16

- coalesced_A

```
ptxas info    : Compiling entry function '_Z19dmm_gpu_coalesced_APKfS0_Pfmmm' for 'sm_35'
ptxas info    : Function properties for _Z19dmm_gpu_coalesced_APKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 37 registers, 1024 bytes smem, 368 bytes cmem[0]
```

- naive

```
ptxas info    : Compiling entry function '_Z13dmm_gpu_naivePKfS0_Pfmmm' for 'sm_35'
ptxas info    : Function properties for _Z13dmm_gpu_naivePKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 16 registers, 368 bytes cmem[0]
```

- reduced_global

```
ptxas info    : Compiling entry function '_Z22dmm_gpu_reduced_globalPKfS0_Pfmmm' for 'sm_35'
ptxas info    : Function properties for _Z22dmm_gpu_reduced_globalPKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 29 registers, 2048 bytes smem, 368 bytes cmem[0]
```

Block Size 32x32

- coalesced_A

```
ptxas info    : Compiling entry function '_Z19dmm_gpu_coalesced_APKfS0_Pfmmm' for 'sm_35'
ptxas info    : Function properties for _Z19dmm_gpu_coalesced_APKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 41 registers, 4096 bytes smem, 368 bytes cmem[0]
```

- naive

```
ptxas info    : Compiling entry function '_Z13dmm_gpu_naivePKfS0_Pfmmm' for 'sm_35'
ptxas info    : Function properties for _Z13dmm_gpu_naivePKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 16 registers, 368 bytes cmem[0]
```

- reduced_global

```
ptxas info    : Compiling entry function '_Z22dmm_gpu_reduced_globalPKfS0_Pfmmm' for 'sm_35'
ptxas info    : Function properties for _Z22dmm_gpu_reduced_globalPKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 29 registers, 8192 bytes smem, 368 bytes cmem[0]
```