Κουτρούμπας Αθανάσιος 03116073
Αραπίδης Μανώλης 03116071

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
www.cslab.ece.ntua.gr

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ
4η ΑΣΚΗΣΗ

Παράλληλος προγραμματισμός σε επεξεργαστές γραφικών
Ακ. έτος 2020-2021, 9ο Εξάμηνο, Σχολή ΗΜ&ΜΥ

2. Ζητούμενα

2.1 Υλοποίηση για επεξεργαστές γραφικών (GPUs)

Βασική υλοποίηση

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

Για όλες τις υλοποιήσεις του αλγορίθμου για DMM στην GPU, αρχικά πρέπει πρώτα να ορίσουμε τον αριθμό των threads στα block, καθώς και τον αριθμό των block στο grid. Χρησιμοποιούμε 2D block και αντίστοιχα 2D grid για αντιστοιχία με τους πίνακες που έχουμε.

- Για το block:
  - `THREAD_BLOCK_Y` threads σε αντιστοιχία με τις γραμμές του πίνακα
  - `THREAD_BLOCK_X` threads σε αντιστοιχία με τις στήλες του πίνακα
- Για το grid: Θέλουμε τουλάχιστον αριθμό των γραμμών/στηλών διά τον αντίστοιχο αριθμό block. Για να πάρουμε το άνω όριο της διαίρεσης, προσθέτουμε στον αριθμητή τον παρονομαστή μείων 1.

Κώδικας `dmm_main.cu`

```
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);
```

Για την βασική υλοποίηση, του naive αλγορίθμου για DMM στην GPU, αρχικά υπολογίζουμε την γραμμή (στον y άξονα) και την στήλη (στον x άξονα) που θα βρίσκεται το κάθε thread.

Για τον υπολογισμό της γραμμής, βρίσκουμε το block που βρίσκεται το thread στον y-άξονα (`blockIdx.y`) και το πολλαπλασιάζουμε με τον αριθμό των threads σε κάθε block του y-άξονα (`blockDim.y`). Αυτό το γινόμενο θα μας βγάλει τον αριθμό του 1ου thread στον block που βρίσκεται το thread μας, οπότε προσθέτουμε το αριθμό του thread στον y-άξονα (`threadIdx.y`) που είναι μοναδικός μόνο σε κάθε block, για να πάρουμε το μοναδικό αριθμό γραμμής του thread από όλα τα block.
Αντίστοιχα, κάνουμε το ίδιο και για τον αριθμό στήλης.

Έπειτα, ελέγχουμε αν η θέση του thread είναι μεγαλύτερη των διαστάσεων του πίνακα C, ώστε να μείνει ανενεργό το thread.
Τέλος, γίνεται ο υπολογισμός του κάθε στοιχείου του πίνακα C, από ένα thread. Το κάθε thread, υλοποιεί στην ουσία το πολλαπλασιασμό μια γραμμής του πίνακα A, με την αντίστοιχη στήλη του πίνακα B, και αθροίζει όλα τα γινόμενα σε μία μεταβλητή, την οποία αποθηκεύουμε στο τέλος στην θέση που έχει το thread στο C πίνακα.

Κώδικας `dmm_gpu.cu`

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

  value_t Cvalue = 0;

  // If the threads positions is out of array bounds, exit
  if (row >= M || col >= N) return;

  // Each thread computes one element of C by accumulating results into Cvalue
  for (int e = 0; e < K; e++) {
       Cvalue += A[row * K + e] * B[e * N + col];
  }

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

1. *Υπολογίστε τον αριθμό των προσβάσεων στην κύρια μνήμη για τους πίνακες A και B συναρτήσει των διαστάσεων M, N, K του προβλήματος και των διαστάσεων του μπλοκ νημάτων (THREAD_BLOCK_X/THREAD_BLOCK_Y για 2D μπλοκ).*

Το κάθε thread, αναλαμβάνει τον υπολογισμό ενός στοιχείου του C πίνακα. Άρα όπως αναφέραμε χρειάζεται να πάρει από την global memory, K στοιχεία της γραμμής του A και K στοιχεία της στήλης του B. Άρα σύνολο $2K$ προσβάσεις στην μνήμη, για κάθε νήμα.
Άρα αφού θα έχουμε $M \cdot N$, στοιχεία του C, άρα και threads, θα έχουμε συνολικά:

- $2 \cdot K \cdot M \cdot N$ προσβάσεις στην μνήμη (global memory)

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

Για το κάθε thread, σε μία επανάληψη του for loop, έχουμε:

- 2 floating operations (πολλαπλασιασμό + πρόσθεση floats)
- 2 προσβάσεις στην μνήμη για floats

Ένας float είναι μεγέθους 4 byte, άρα έχουμε:

- 2 FLOPs / 8 byte = **1/4 FLOPs/byte**

Άρα, έχουμε 1 Floating-point operation για κάθε 4 byte που φέρνουμε από την κύρια μνήμη. Βλέπουμε ότι το πρόγραμμά μας είναι memory bound, καθώς σε μία GPU (και γενικότερα σε μία επεξεργαστική μονάδα), το computational speed είναι πολύ μεγαλύτερο από το memory bandwidth, άρα θα θέλαμε για κάθε 4 byte που φέρνουμε από την κύρια μνήμη, να μπορούμε να κάνουμε περισσότερα operations.

Πιο αναλυτικά, για την NVIDIA Tesla K40c, που χρησιμοποιούμε για τις μετρήσεις, έχει:

- Bandwidth: **288.4 GB/s**
- FP32 (float) performance: **4.29 TFLOPS**
- Operational Intensity (σημείο γονάτου): 4.29 TFLOPS/288.4 GB/s = **14.9 FLOPs/byte**

Άρα προφανώς ο δικός μας kernel, με **0.25 FLOPs/byte** Operational Intensity, βρίσκεται αριστερότερα του σημείου γονάτου στο roofline μοντέλο, άρα είναι memory-bound.

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

Εφόσων οι πίνακες είναι αποθηκευμένοι στην μνήμη κατά γραμμές (row-major μορφή), τότε οι προσβάσεις στην μνήμη για τον πίνακα A συνενώνονται στην κρυφή μνήμη του κάθε thread, καθώς κάθε thread παίρνει μία γραμμή του πίνακα. Όμως οι προσβάσεις στην μνήμη για τον πίνακα B δεν συνενώνονται γιατί κάθε thread παίρνει μία στήλη του πίνακα, η οποία δεν είναι συνεχόμενα αποθηκευμένη στην μνήμη.

4. *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.*

Για να χρησιμοποιήσουμε το CUDA Occupancy Calculator, χρειάζεται να ξέρουμε κάποια χαρακτηριστικά για την GPU που χρησιμοποιούμε. Για να βρούμε αυτά τα χαρακτηριστικά εκτελέσαμε `make query` στο dungani.

Για την NVIDIA Tesla K40c ισχύουν:

- Computer Capability: **3.5**
- Shared Memory Size Config: **65536 bytes**

Για το resource usage, αρκεί από το κάθε compile της εφαρμογής (έχοντας στο Makefile `REGINFO=1`), να πάρουμε τις πληροφορίες για το kernel που εξετάζουμε, από το ptxas.

Για τον **naive kernel**, έχουμε (ανεξάρτητα του Block Size):

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

Για διαφορετικά μεγέθη block size έχουμε πάντα την ίδια χρήση shared memory και registers per thread:

- Registers per Thread: **16 registers**
- Used Shared Memory per Block: **0 bytes**

Το μόνο που αλλάζει είναι τα Threads per Block, άρα δοκιμάζοντας για διάφορα block sizes:

Για resource usage **block 4x4**:

- Threads per Block: **16 threads**

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%

Άρα βλέπουμε ότι έχουμε **25% χρησιμοποιήση των πολυεπεξεργαστικών στοιχείω**ν.

Για resource usage **block 8x8**:

- Threads per Block: **64 threads**

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%

Άρα βλέπουμε ότι έχουμε **50% χρησιμοποιήση των πολυεπεξεργαστικών στοιχείων.**

Για resource usage **block 16x16**:

- Threads per Block: **256 threads**

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%

Άρα βλέπουμε ότι έχουμε **100% χρησιμοποιήση των πολυεπεξεργαστικών στοιχείων.**

Για resource usage **block 32x32**:

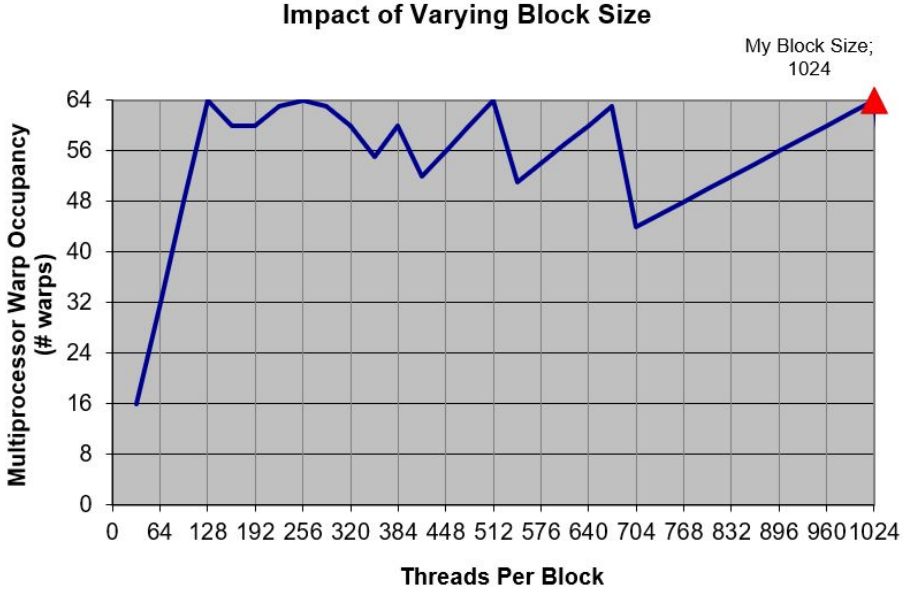
- Threads per Block: **1024 threads**

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%

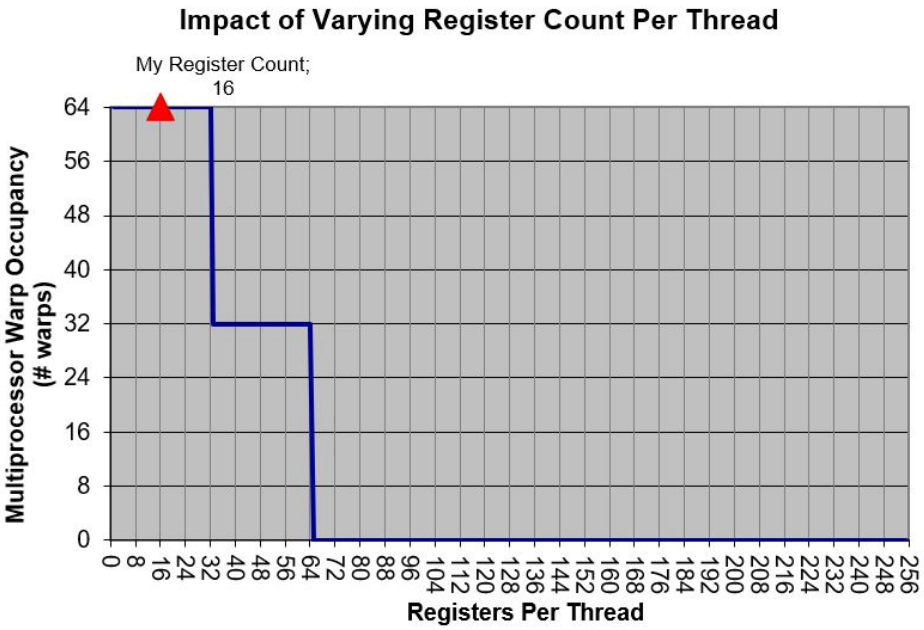
Άρα βλέπουμε ότι έχουμε **100% χρησιμοποιήση των πολυεπεξεργαστικών στοιχείων.**

Παρακάτω έχουμε τα γραφήματα που βγάζει το Calculator (για **block 32x32**), σε σύγκριση με διαφορετικό:

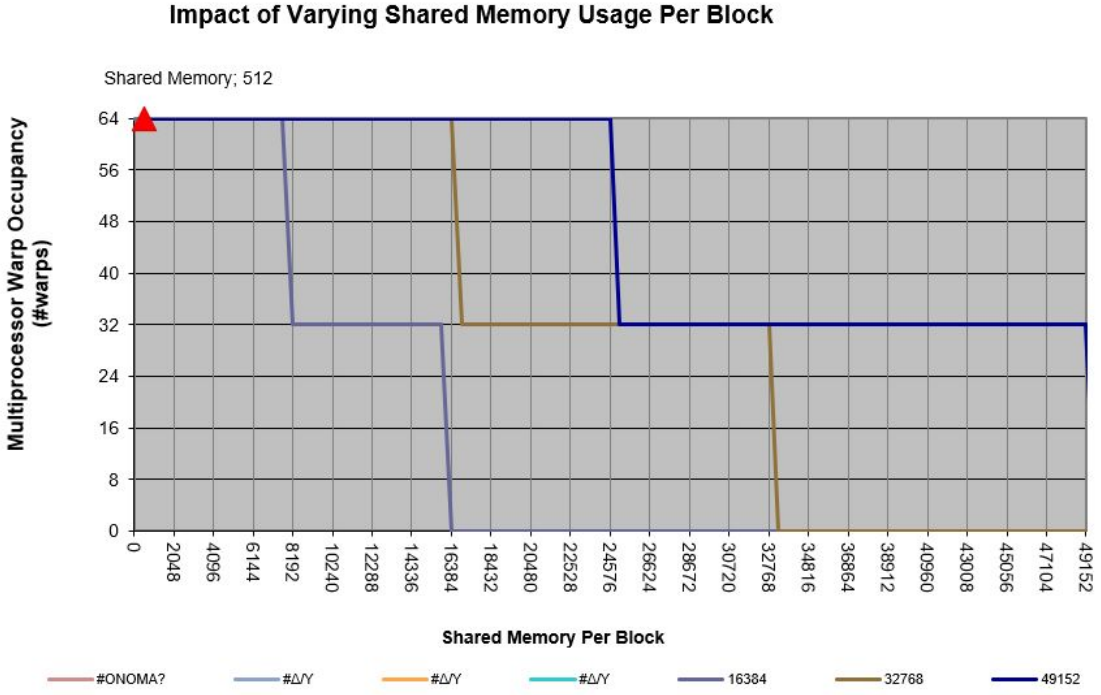
- Αριθμό Threads per Block
- Αριθμό Registers per Thread
- Αριθμό Used Shared Memory per Block



Χρησιμοποίηση πολυεπεξεργαστικών στοιχείων με βάση των αριθμό Threads per Block



Χρησιμοποίηση πολυεπεξεργαστικών στοιχείων με βάση των αριθμό Registers per Thread



Χρησιμοποίηση πολυεπεξεργαστικών στοιχείων με βάση την Shared Memory per Block

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.

Για την υλοποίηση, του αλγορίθμου συνένωσης των προσβάσεων στην κύρια μνήμη για τον Πίνακα Α, θα πρέπει να φέρνουμε τμηματικά στοιχεία από τον πίνακα Α στην shared memory του block, ώστε τα threads στο ίδιο block να χρειάζονται πλέον προσβάσεις μόνο στην shared memory που γίνονται πολύ πιο γρήγορα από ότι στην global memory, όπως προηγουμένως. Για να το πετύχουμε αυτό, έχουμε τα `TILE_X`,`TILE_Y` επεξεργασίας (τα οποία έχουμε υποθέσει ότι θα είναι ίδια με τα thread block), τα οποία ορίζουν το μέγεθος του υποπίνακα που θα προφορτώσουμε στην shared memory από τον πίνακα Α.

Η υλοποίηση είναι παρόμοια με την naive, απλά θα πρέπει πρώτα να κάνουμε μία εξωτερική επανάληψη για όλα τα Tiles που θα χρειαστούμε στον x-άξονα (δηλαδή, loop στην γραμμή του Α), όπου φορτώνουμε το κατάλληλο tile στην shared memory, συγχρονίζουμε τα threads, ώστε να ξεκινήσουν τα threads τον υπολογισμό μετά την φόρτωση του tile και έπειτα γίνεται ο πολλαπλασιασμός της γραμμής του Α με την στήλη του Β, και άθροισμα των γινομένων. Μετά τον υπολογισμό, έχουμε έναν ακόμα συγχρονισμό για να σιγουρευτούμε ότι όλα τα threads έχουν τελειώσει τον υπολογισμό πριν φορτωθεί το καινούριο tile στην shared memory.

Κώδικας `dmm_gpu.cu`

```
__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 memory between threads of the same block, for Tiled sub-matrix of A
  __shared__ value_t A_shared[TILE_X][TILE_Y];

  // Compute the row and column of each thread in a Tile
  int tid_y = threadIdx.y;
  int tid_x = threadIdx.x;
  int row = blockIdx.y * TILE_Y + tid_y;
  int col = blockIdx.x * TILE_X + tid_x;

  // Each thread computes one element of C by accumulating results into Cvalue
  value_t Cvalue = 0;

  // Calculate the ceiling for number of Tiles needed for A
  int tile_x_ceil = (K + TILE_X - 1) / TILE_X;

  // Loop over all the sub-matrices of A (on x-axis, for all columns) that are
  // required to compute Csub.
  for (int m = 0; m < tile_x_ceil; m++) {
       // Load sub-matrix of A from device memory to shared memory
       A_shared[tid_y][tid_x] = A[row * K + m * TILE_X + tid_x];

       // Synchronize to make sure the sub-matrix is loaded
       // before starting the computation
       __syncthreads();
```

```
        // Multiply the A sub-matrix with B and accumulate the results.
        for (int e = 0; e < TILE_X; e++) {
        Cvalue += A_shared[tid_y][e] * B[(m * TILE_X + e) * N + col];
        }
        // Synchronize to make sure that the preceding computation is done
        // before loading new sub-matrix of A and in the next iteration
        __syncthreads();
  }
  // Write Csub to device memory
  C[row * N + col] = Cvalue;
}
```

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

Προηγουμένως, συνολικά ο αριθμός των προσβάσεων στην μνήμη για τον πίνακα A ήταν:

- $K \cdot M \cdot N$ προσβάσεις στην μνήμη (global memory)

Τώρα, εφόσων φορτώνουμε πλέον Tiles στην shared memory, υποπινάκων του A, οι φορές που χρειάζεται να κάνουμε access την global memory είναι:

- $K \,/\, TILE\_X$

Άρα έχουμε μείωση των προσβάσεων στην μνήμη κατά:

- $\frac{K \cdot M \cdot N}{K \,/\, TILE\_X} = M \cdot N \cdot TILE\_X$ μείωση προσβάσεων στην μνήμη (global memory)

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

Για το κάθε thread, σε μία επανάληψη του for loop, έχουμε:

- 2 floating operations (πολλαπλασιασμό + πρόσθεση floats)
- 1 πρόσβαση στην μνήμη για floats (για τον πίνακα B)

Ένας float είναι μεγέθους 4 byte, άρα έχουμε:

- 2 FLOPs / 4 byte = **1/2 FLOPs/byte**

Άρα, πάλι είμαστε memory bound για τους λόγους που αναφέραμε στην προηγούμενη υλοποίηση και πάλι ο kernel με **0.5 FLOPs/byte** Operational Intensity, βρίσκεται αριστερότερα του σημείου γονάτου (**14.9 FLOPs/byte**) στο roofline μοντέλο για την GPU που χρησιμοποιούμε.

3. *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.*

Πάλι, όπως και πριν χρησιμοποιούμε το CUDA Occupancy Calculator, με τα χαρακτηριστικά της κάρτας που αναφέραμε.

Για τον **coalesced_A kernel**, έχουμε (π.χ. για Block Size 4x4):

```
ptxas info    : Compiling entry function '_Z19dmm_gpu_coalesced_APKfS0_Pfmmm' for 'sm_35'
ptxas info    : Function properties for _Z19dmm_gpu_coalesced_APKfS0_Pfmmm
ptxas info    : Used 28 registers, 64 bytes smem, 368 bytes cmem[0]
```

Για διαφορετικά μεγέθη block size τώρα δεν έχουμε πάντα την ίδια χρήση shared memory και registers per thread.
Οπότε, δοκιμάζοντας για διάφορα block sizes:

Για resource usage **block 4x4**:

- Threads per Block: **16 threads**
- Registers per Thread: **28 registers**
- Used Shared Memory per Block: **64 bytes**

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%

Άρα βλέπουμε ότι έχουμε **25% χρησιμοποιήση των πολυεπεξεργαστικών στοιχείων.**

Για resource usage **block 8x8**:

- Threads per Block: **64 threads**
- Registers per Thread: **34 registers**
- Used Shared Memory per Block: **256 bytes**

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%

Άρα βλέπουμε ότι έχουμε **50% χρησιμοποιήση των πολυεπεξεργαστικών στοιχείων.**

Για resource usage **block 16x16**:

- Threads per Block: **256 threads**
- Registers per Thread: **39 registers**
- Used Shared Memory per Block: **1024 bytes**

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%

Άρα βλέπουμε ότι έχουμε **75% χρησιμοποιήση των πολυεπεξεργαστικών στοιχείων.**

Για resource usage **block 32x32**:

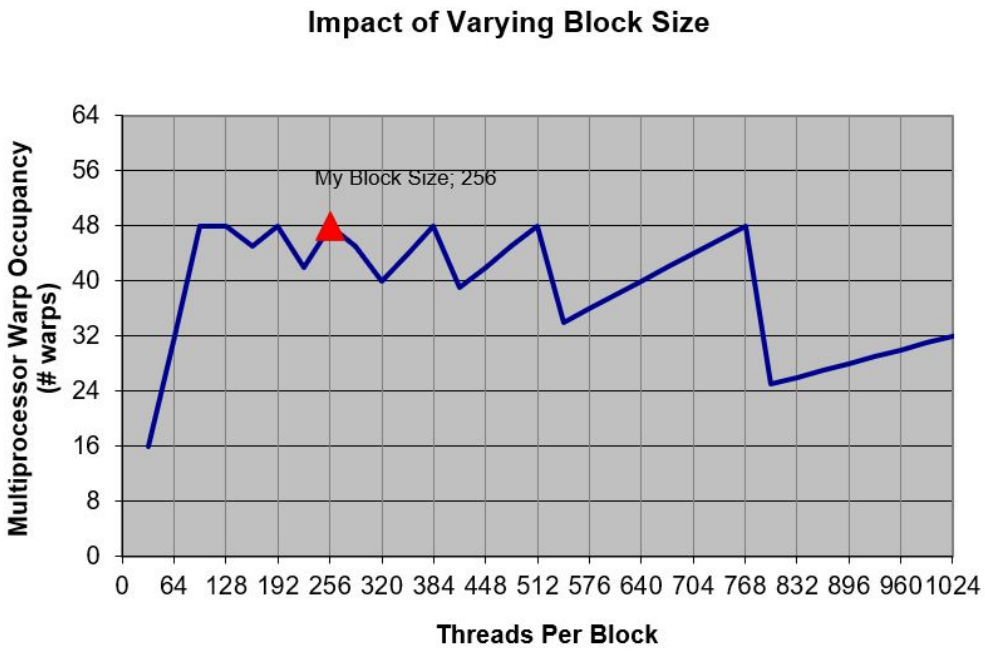
- Threads per Block: **1024 threads**
- Registers per Thread: **40 registers**
- Used Shared Memory per Block: **4096 bytes**

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%

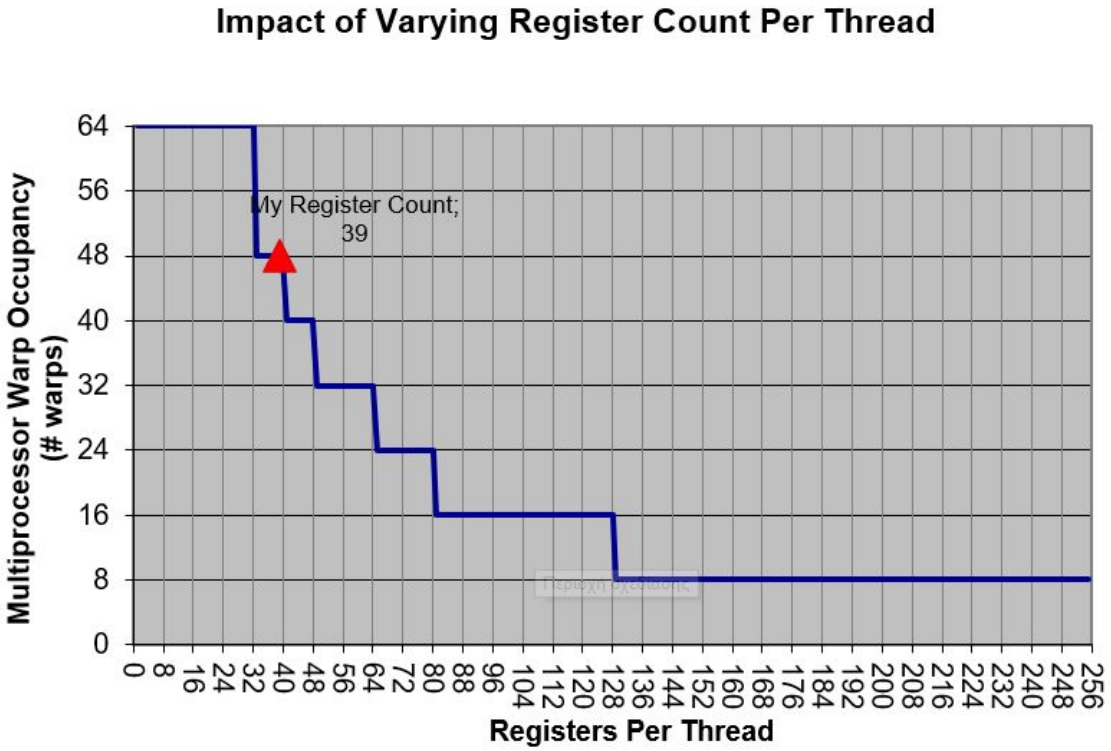
Άρα βλέπουμε ότι έχουμε **50% χρησιμοποιήση των πολυεπεξεργαστικών στοιχείω**ν.

Παρακάτω έχουμε τα γραφήματα που βγάζει το Calculator (για **block 16x16**), σε σύγκριση με διαφορετικό:

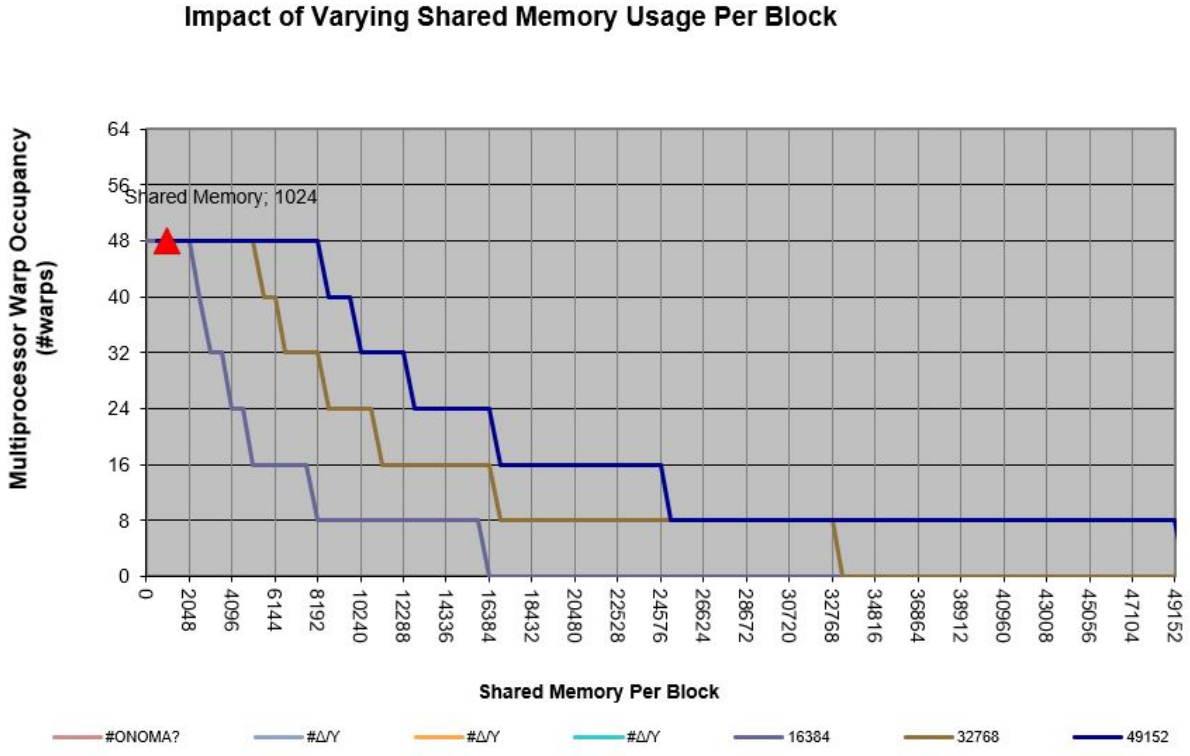
- Αριθμό Threads per Block
- Αριθμό Registers per Thread
- Αριθμό Used Shared Memory per Block



Χρησιμοποίηση πολυεπεξεργαστικών στοιχείων με βάση των αριθμό Threads per Block



Χρησιμοποίηση πολυεπεξεργαστικών στοιχείων με βάση των αριθμό Registers per Thread



Χρησιμοποίηση πολυεπεξεργαστικών στοιχείων με βάση την Shared Memory per Block

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.

Για την υλοποίηση, του αλγορίθμου μείωσης των προσβάσεων στην κύρια μνήμη και για τον Πίνακα B, απλά τροποποιήσαμε τον προηγούμενο κώδικα ώστε να φέρνουμε τμηματικά στοιχεία και από τον πίνακα A και από τον B στην shared memory του block. Οπότε τα `TILE_X,TILE_Y` επεξεργασίας (τα οποία έχουμε υποθέσει ότι θα είναι ίδια με τα thread block), ορίζουν το μέγεθος των υποπινάκων που θα προφορτώσουμε στην shared memory από τους πίνακα A και B.

Η υλοποίηση είναι ίδια με την προηγούμενη, απλά ορίζουμε και έναν υποπίνακα B μεγέθους `TILE_X,TILE_Y` στην shared memory και σε κάθε εξωτερική επανάληψη για όλα τα Tiles που θα χρειαστούμε στον x-άξονα (δηλαδή, loop στην γραμμή του A), φορτώνουμε το κατάλληλο tile στην shared memory και από τον A και από τον B.

Κώδικας `dmm_gpu.cu`

```
__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 memory between threads of the same block, for Tiled sub-matrices A,B
  __shared__ value_t A_shared[TILE_X][TILE_Y];
  __shared__ value_t B_shared[TILE_X][TILE_Y];

  // Compute the row and column of each thread in a Tile
  int tid_y = threadIdx.y;
  int tid_x = threadIdx.x;
  int row = blockIdx.y * TILE_Y + tid_y;
  int col = blockIdx.x * TILE_X + tid_x;

  // Each thread computes one element of C by accumulating results into Cvalue
  value_t Cvalue = 0;

  // Calculate the ceiling for number of Tiles needed for A
  int tile_x_ceil = (K + TILE_X - 1) / TILE_X;

  // Loop over all the sub-matrices of A (on x-axis, for all columns) and
  // B (on y-axis, for all rows) that are required to compute Csub.
  for (int m = 0; m < tile_x_ceil; m++) {
       // Load sub-matrices of A,B from device memory to shared memory
       A_shared[tid_y][tid_x] = A[row * K + m * TILE_X + tid_x];
       B_shared[tid_y][tid_x] = B[col + (m * TILE_Y + tid_y) * N];

       // Synchronize to make sure the sub-matrix is loaded
       // before starting the computation
       __syncthreads();

       // Multiply the sub-matrices together and accumulate the results.
       for (int e = 0; e < TILE_X; e++) {
       Cvalue += A_shared[tid_y][e] * B_shared[e][tid_x];
       }
```

```
        // Synchronize to make sure that the preceding computation is done
        // before loading new sub-matrix of A and in the next iteration
        __syncthreads();
    }
    // Write Csub to device memory
    C[row * N + col] = Cvalue;
}
```

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

Στην αρχική υλοποίηση, συνολικά ο αριθμός των προσβάσεων στην μνήμη για τους πίνακες A, B ήταν:

- $2 \cdot K \cdot M \cdot N$ προσβάσεις στην μνήμη (global memory)

Τώρα, εφόσων φορτώνουμε πλέον Tiles στην shared memory, υποπινάκων του A και B, οι φορές που χρειάζεται να κάνουμε access την global memory είναι:

- $K \,/\, TILE\_X$

Άρα έχουμε μείωση των προσβάσεων στην μνήμη κατά:

- $\frac{2 \cdot K \cdot M \cdot N}{K \,/\, TILE\_X} = 2 \cdot M \cdot N \cdot TILE\_X$ μείωση προσβάσεων στην μνήμη (global memory)

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

Για το κάθε thread, σε μία επανάληψη του for loop, έχουμε:

- 2 floating operations (πολλαπλασιασμό + πρόσθεση floats)
- Καμία πρόσβαση στην μνήμη (μόνο προσβάσεις στην shared memory)

Άρα προφανώς δεν είμαστε καθόλου memory-bound για το κομμάτι του εσωτερικού βρόχου, αφού δεν χρειαζόμαστε προσβάσεις στην global memory. Οπότε πλέον είμαστε compute-bound, και η επίδοση μας εξαρτάται από την υπολογιστική ικανότητα της GPU.

Προφανώς και δεν θα πιάσουμε το μέγιστα TFLOPs της GPU, γιατί πάλι έχουμε συνολικά στον kernel προσβάσεις στην μνήμη που περιορίζουν την απόδοση, αλλά για το κομμάτι του εσωτερικού βρόχου είμαστε μόνο compute-bound.

3. *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.*

Πάλι, όπως και πριν χρησιμοποιούμε το CUDA Occupancy Calculator, με τα χαρακτηριστικά της κάρτας που αναφέραμε.

Για τον **reduced_global kernel**, έχουμε (π.χ. για Block Size 4x4):

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

Για διαφορετικά μεγέθη block size τώρα πάλι δεν έχουμε πάντα την ίδια χρήση shared memory και registers per thread.
Οπότε, δοκιμάζοντας για διάφορα block sizes:

Για resource usage **block 4x4**:

- Threads per Block: **16 threads**
- Registers per Thread: **24 registers**
- Used Shared Memory per Block: **128 bytes**

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%

Άρα βλέπουμε ότι έχουμε **25% χρησιμοποιήση των πολυεπεξεργαστικών στοιχείων.**

Για resource usage **block 8x8**:

- Threads per Block: **64 threads**
- Registers per Thread: **26 registers**
- Used Shared Memory per Block: **512 bytes**

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%

Άρα βλέπουμε ότι έχουμε **50% χρησιμοποιήση των πολυεπεξεργαστικών στοιχείων.**

Για resource usage **block 16x16**:

- Threads per Block: **256 threads**
- Registers per Thread: **29 registers**
- Used Shared Memory per Block: **2048 bytes**

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%

Άρα βλέπουμε ότι έχουμε **100% χρησιμοποιήση των πολυεπεξεργαστικών στοιχείων.**

Για resource usage **block 32x32**:

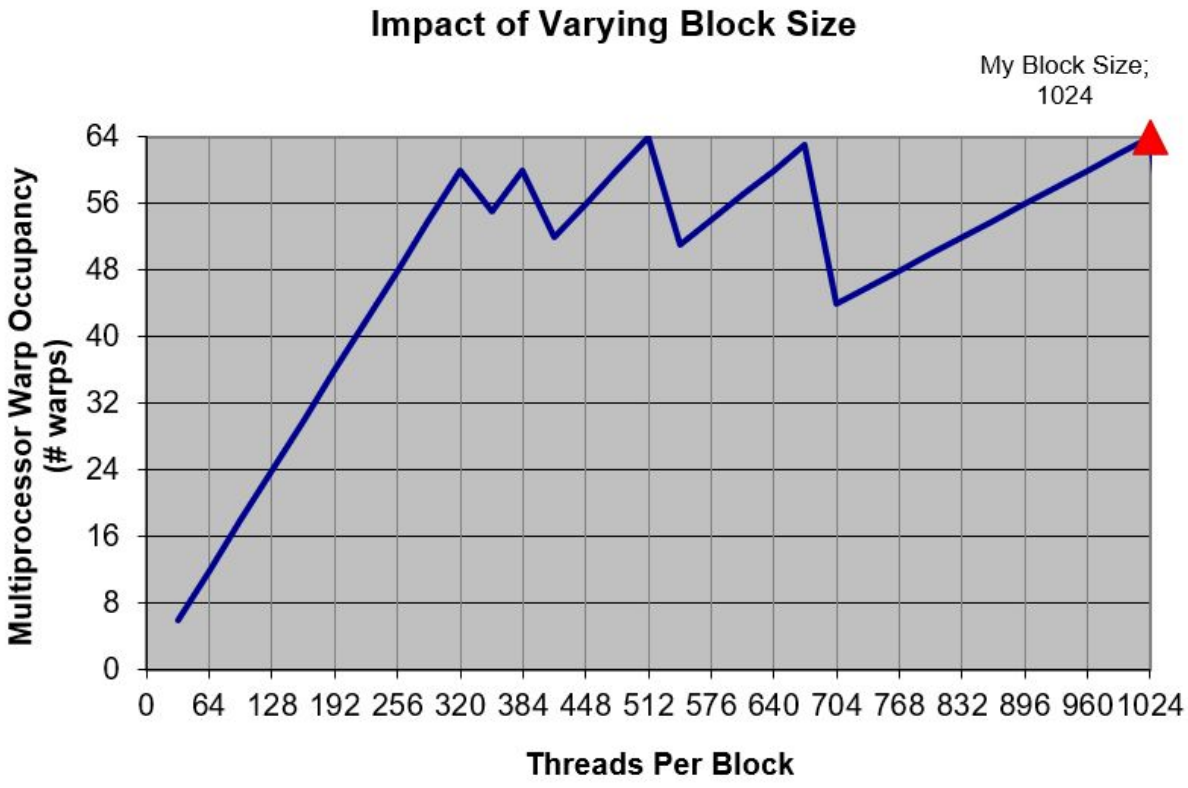
- Threads per Block: **1024 threads**
- Registers per Thread: **29 registers**
- Used Shared Memory per Block: **8192 bytes**

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%

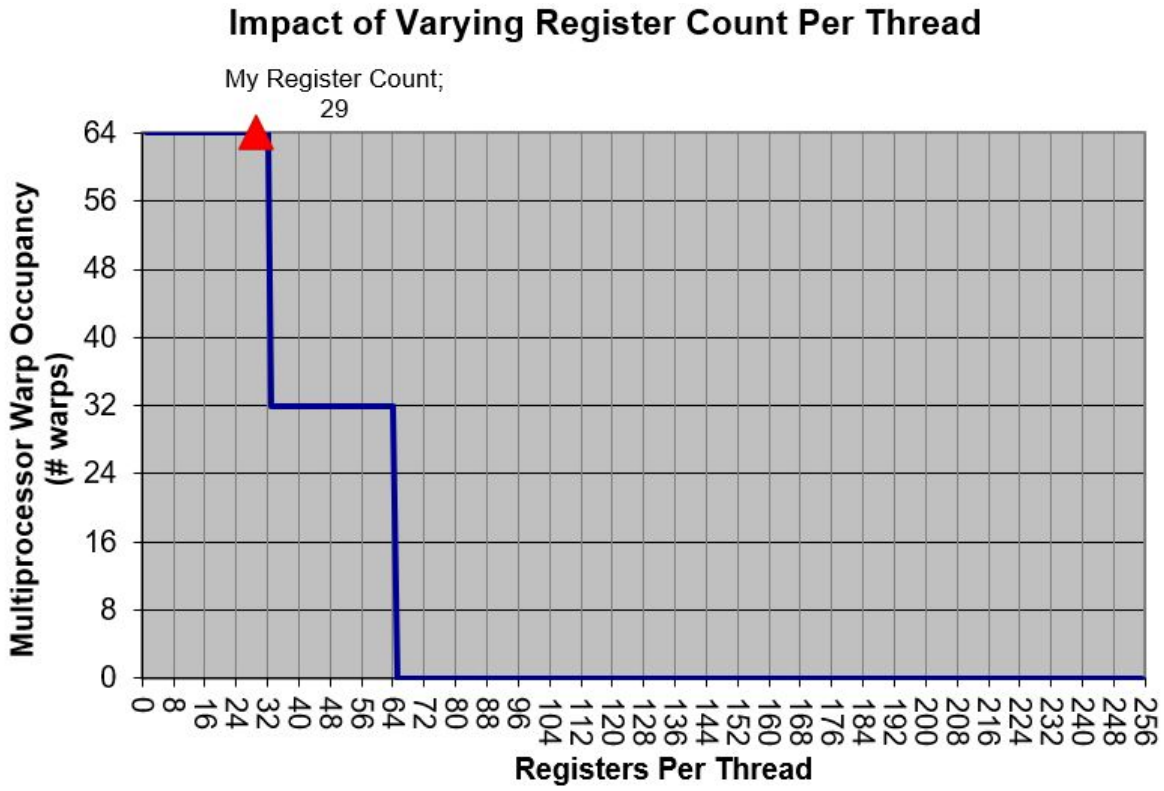
Άρα βλέπουμε ότι έχουμε **100% χρησιμοποιήση των πολυεπεξεργαστικών στοιχείων**.

Παρακάτω έχουμε τα γραφήματα που βγάζει το Calculator (για **block 32x32**), σε σύγκριση με διαφορετικό:

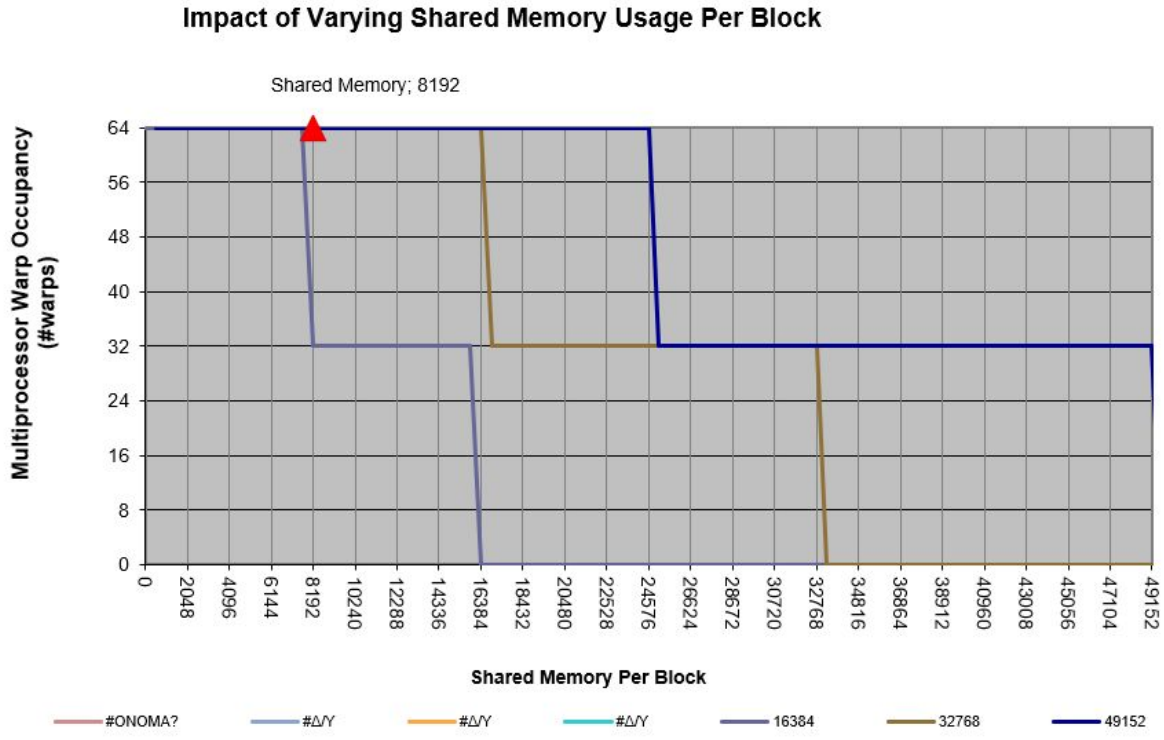
- Αριθμό Threads per Block
- Αριθμό Registers per Thread
- Αριθμό Used Shared Memory per Block



Χρησιμοποίηση πολυεπεξεργαστικών στοιχείων με βάση των αριθμό Threads per Block



Χρησιμοποίηση πολυεπεξεργαστικών στοιχείων με βάση των αριθμό Registers per Thread



Χρησιμοποίηση πολυεπεξεργαστικών στοιχείων με βάση την Shared Memory per Block

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.

Για να χρησιμοποιήσουμε την `cublasSgemm()`, της βιβλιοθήκης cuBLAS, χρειάστηκε απλά να ορίσουμε και να μετατρέψουμε τα κατάλληλα ορίσματα, ώστε να έχουμε το επιθυμητό αποτέλεσμα σύμφωνα και με το πώς έχουμε αποθηκευμένα τα δεδομένα μας.

Πιο αναλυτικά, η `cublasSgemm()`, εκτελεί την γενική πράξη πινάκων:

$$C = \alpha \cdot op(A) \cdot op(B) + \beta \cdot C$$

Όπου, $op(A)$:

- $op(A) = A$, non-transpose A `(transa == CUBLAS_OP_N)`
- $op(A) = A^T$, transpose A `(transa == CUBLAS_OP_T)`
- $op(A) = A^H$, conjugate A `(transa == CUBLAS_OP_C)`

Εμείς όμως για τον πολλαπλασιασμό πινάκων, μας αρκεί να έχουμε:

- $\alpha = 1$ `(alpha = 1)`
- $\beta = 0$ `(beta = 0)`

Επίσης, εμείς έχουμε αποθηκευμένους τους πίνακες σε row-major μορφή (κατά γραμμές). Οι βιβλιοθήκες της cuBLAS όμως, υποθέτουν ότι οι πίνακες είναι αποθηκευμένοι σε column-major μορφή (κατά στήλες). Άρα, το cuBLAS σύμφωνα με αυτήν την σύμβαση θα δεχθεί τους ανάστροφους (transpose) πίνακες A και B, για να κάνει πολλαπλασιασμό πινάκων.
Θα μπορούσαμε να δηλώσουμε μέσω του `transa, transb` ότι είναι ανάστροφοι οι πίνακες και να κάνουμε την πράξη, όμως και το αποτέλεσμα, ο πινακας C, θα ήταν ανάστροφος για εμάς, άρα θα έπρεπε να τον μετατρεψουμε σε row-major μορφή.

Μπορούμε όμως να αντιστρέψουμε την σειρά που δίνουμε τους πίνακες A, B (από την `dmm_main.cu`), ώστε να αξιοποιήσουμε την παρακάτω ιδιότητα της γραμμικής άλγεβρας:

$$C^T = (AB)^T = B^T A^T$$

Άρα η `cublasSgemm()`, θα πάρει ως 1ο πίνακα τον B σε row-major μορφή, και ως 2ο πίνακα τον A σε row-major μορφή, και άρα θα μας γυρίσει τον C σε row-major μορφή (από την μεριά της συνάρτησης, αυτή γυρίζει τον transpose C σε column-major μορφή).

Κώδικας `dmm_main.cu`

```
/* Execute and time the kernel */
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);
}
```

Κώδικας dmm_gpu.cu

```
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 variables for cuBLAS status and handle
  cublasStatus_t stat;
  cublasHandle_t handle;

  // Define leading dimensions of A,B,C matrices for cublasSgemm
  int lda = N;
  int ldb = K;
  int ldc = N;

  // Define alpha, beta values for GEMM calculation
  // C = alpha*A*B + beta*C
  const value_t alpha_val = 1;
  const value_t beta_val = 0;
  const value_t *alpha = &alpha_val;
  const value_t *beta = &beta_val;

  // Create a handle for cuBLAS
  stat = cublasCreate(&handle);
  if (stat != CUBLAS_STATUS_SUCCESS) {
       printf("CUBLAS initialization failed\n");
  }

  // Call cublasSgemm to calculate the DMM (for floats)
  stat = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K, alpha, A, lda,
                    B, ldb, beta, C, ldc);
  if (stat != CUBLAS_STATUS_SUCCESS) {
       printf("cublasSgemm failed");
  }

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

4. Πειράματα και μετρήσεις επιδόσεων

4.1 Σενάριο μετρήσεων και διαγράμματα

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

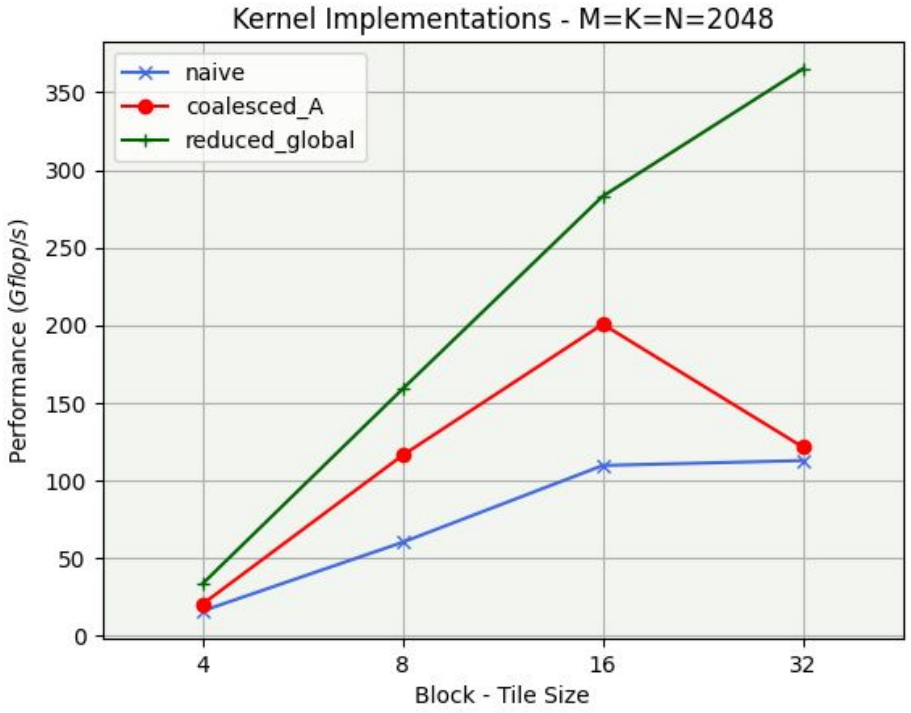
- *Για κάθε έκδοση πυρήνα που υλοποιήσατε (`naive, coalesced, reduced_global`) να καταγράψετε πώς μεταβάλλεται η επίδοση για διαφορετικές διαστάσεις του μπλοκ νημάτων (`THREAD_BLOCK_X/Y`) και υπολογισμού (`TILE_X/Y`) για διαστάσεις πινάκων `M = N = K = 2048`.*

Δοκιμάσαμε για:

- ➔ `GPU_KERNELS: naive, coalesced, reduced_global`
- ➔ `THREAD_BLOCK_X/Y=TILE_X/Y: 4x4, 8x8, 16x16, 32x32`
- ➔ `Sizes: M = N = K = 2048`

Δεν τρέχουμε για παραπάνω blocks, καθώς η GPU ορίζει μέγιστο αριθμό threads ανά block τα 1024.

Προσαρμόζοντας κατάλληλα το `run_dmm.sh` script, τρέχουμε και παίρνουμε αποτελέσματα για το πρώτο σενάριο μετρήσεων. Έπειτα τα κάνουμε plot στο παρακάτω διάγραμμα (τρέχοντας το αρχείο `plot_1.py` μέσω του `plot.sh`):



Όπως φαίνεται και από το διάγραμμα, υπάρχει μια αύξουσα τάση του performance με την αύξηση του block size, εκτός όμως από τον `coalesced_A`, που στα 16x16 έχει το peak performance και μετά πέφτει. Για την `naive` υλοποίηση αυξάνεται, με μικρή βελτίωση από τα 16 στα 32, ενώ στην `reduced_global` έχουμε γραμμική αύξηση.
Τα αποτελέσματα επίσης, συμβαδίζουν και με τα αποτελέσματα χρησιμοποίησης από το Occupancy Calculator, καθώς στην περίπτωση του `coalesced_A`, την καλύτερη χρησιμοποίηση την είχαμε για 16x16 block size.

Η αύξηση του block size, σημαίνει ότι έχουμε περισσότερα threads που τρέχουν παράλληλα, οπότε μπορούν να εξουδετερώνονται κάποιες καθυστερήσεις που υπάρχουν για προσβάσεις στην μνήμη. Επίσης για τις υλοποιήσεις που χρησιμοποιούν τα tiles, η αύξηση του tile size σημαίνει και λιγότερες προσβάσεις στην global memory, οπότε έχουμε και από αυτό μεγαλύτερη αύξηση της επίδοσης.

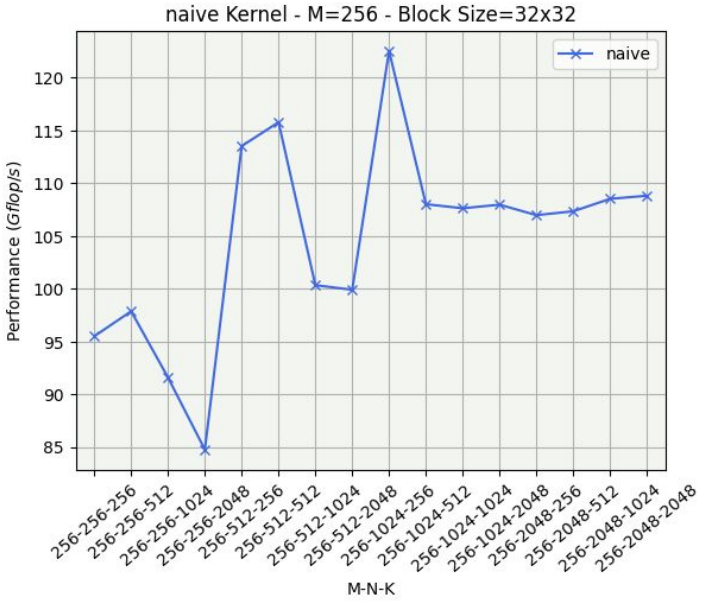
Αυτό που παρατηρούμε επίσης είναι ότι η κάθε υλοποίηση, είναι καλύτερη από την προηγούμενη, όπως ήταν και αναμενόμενο, καθώς κάθε υλοποίηση είναι μια βελτίωση της προηγούμενης, χρησιμοποιώντας την shared memory.

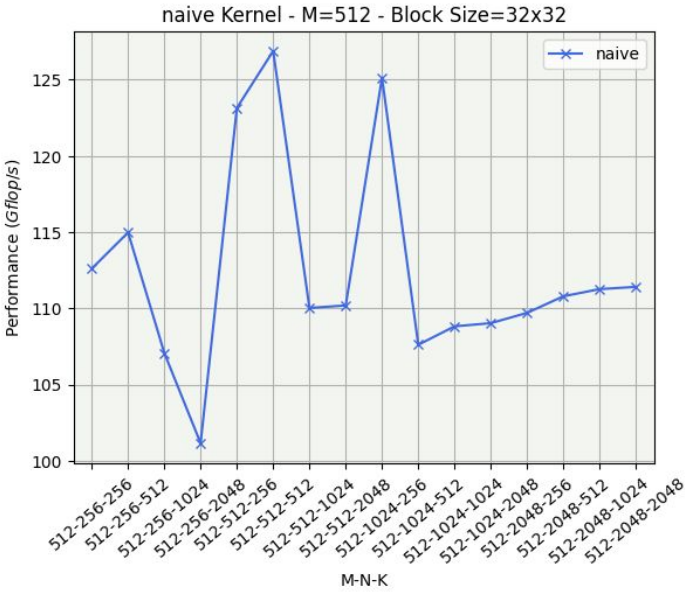
- *Για κάθε μία από τις τέσσερις εκδόσεις πυρήνων (naive, coalesced, shmem και cuBLAS) να καταγράψετε την επίδοση για μεγέθη πινάκων M, N, K ∈ [256, 512, 1024, 2048]. Για τις naive, coalesced και shmem υλοποιήσεις επιλέξτε τις βέλτιστες διαστάσεις μπλοκ νημάτων και υπολογισμών.*

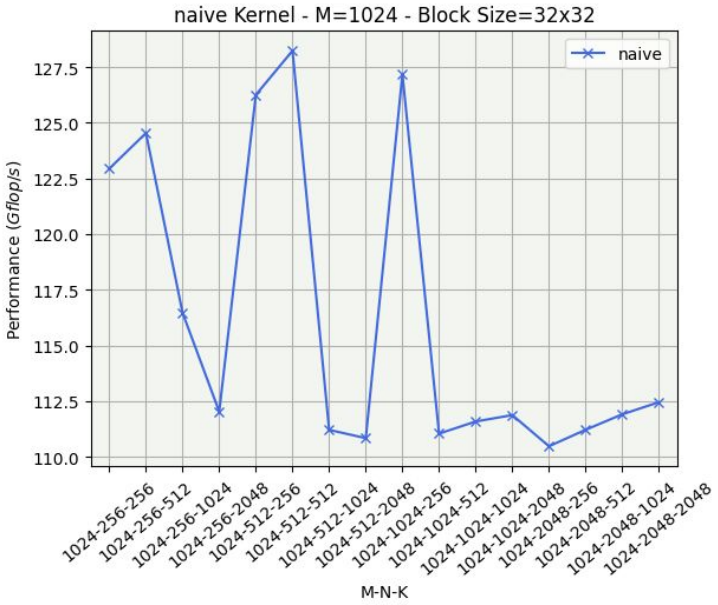
  Έχοντας βρει τα καλύτερα block-tile sizes, για κάθε υλοποίηση τώρα δοκιμάζουμε για διάφορα μεγέθη για όλες τις υλοποιήσεις:
  - ➔ `GPU_KERNELS: naive, coalesced, reduced_global, cuBLAS`
  - ➔ `Sizes: M, N, K ∈ [256, 512, 1024, 2048]`

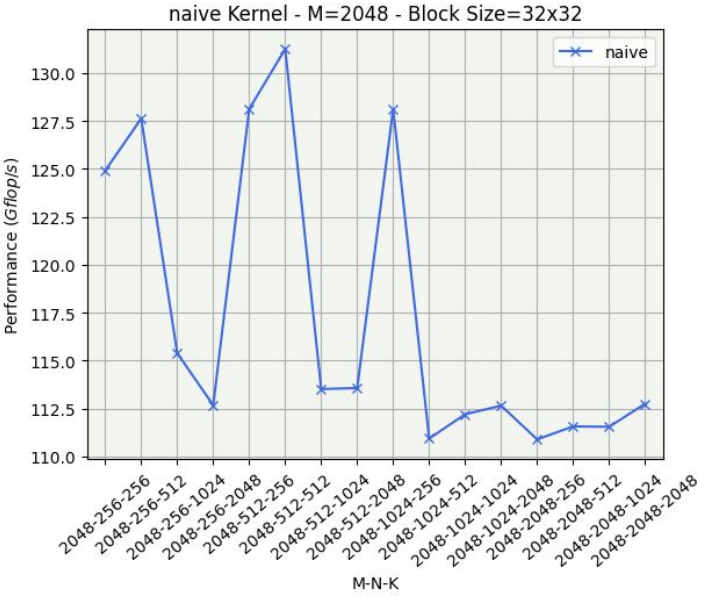
  Προσαρμόζοντας κατάλληλα το `run_dmm.sh` script, τρέχουμε και παίρνουμε αποτελέσματα για το 2ο σενάριο μετρήσεων. Επειδή έχουμε συνολικά 64 διαφορετικούς συνδυασμούς μεγεθών για τα M,N,K για την κάθε υλοποίηση, έχουμε επιλέξει να φτιάξουμε 4 διαγράμματα στα οποία κάθε φορά είναι σταθερή η M διάσταση και αλλάζουν όλες οι υπόλοιπες. Επίσης έχουμε και συγκεντρωτικά 4 διαγράμματα για όλα τα μεγέθη του M, για όλες τις υλοποιήσεις μαζί. (Τα plot δημιουργούνται τρέχοντας το αρχείο `plot_2.py` μέσω του `plot.sh`).

Για τον `naive` kernel:





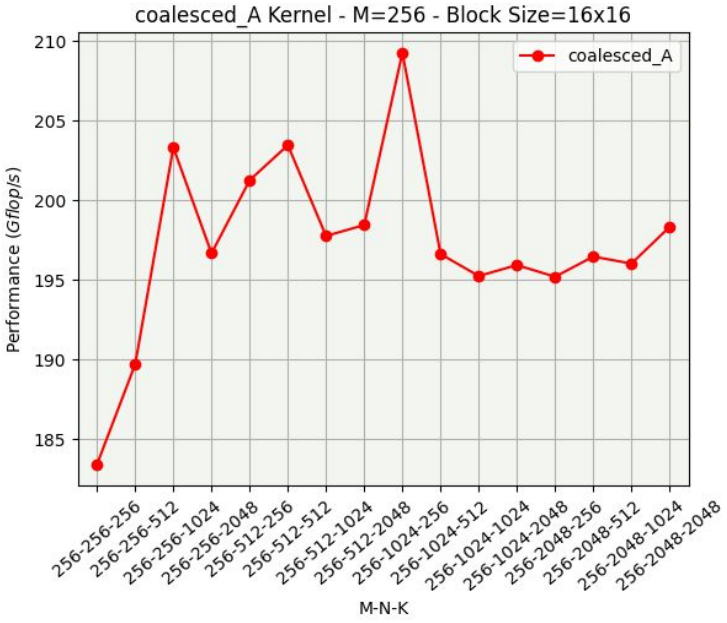


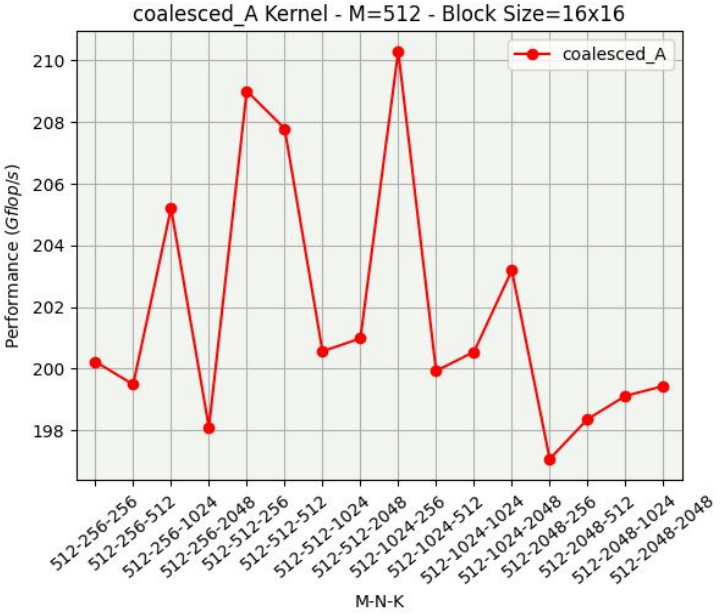


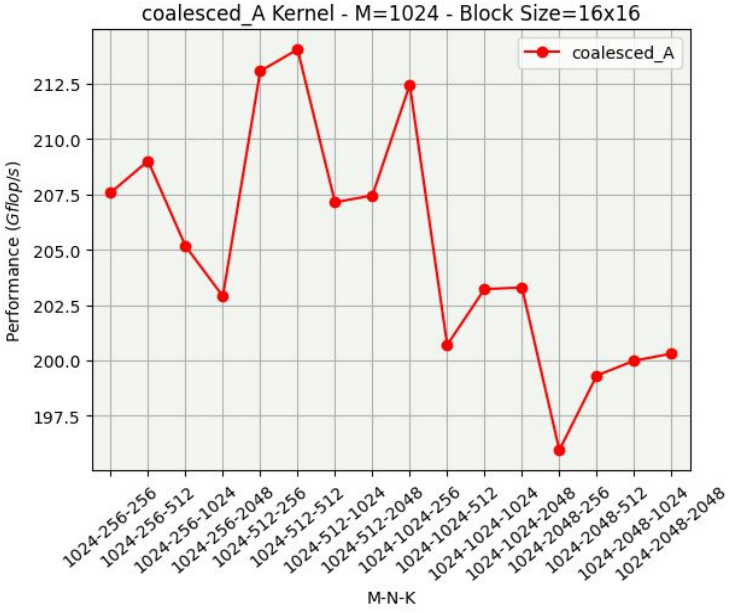
Παρατηρήσεις

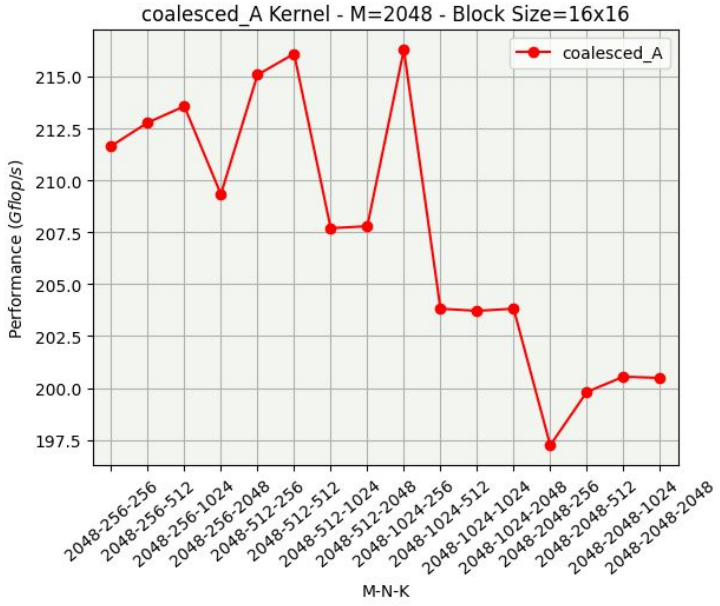
Παρατηρούμε ότι, γενικά για μικρά μεγέθη M, N όσο αυξάνεται το μέγεθος K, έχουμε πτώση της επίδοσης. Αυτό συμβαίνει, διότι το K, είναι που καθορίζει το πόσο στοιχεία χρειαζόμαστε από τον πίνακα A και B, για τον υπολογισμό ενός στοιχείου του πίνακα C. Οπότε με την αύξηση του, αυξάνονται και οι προσβάσεις στην μνήμη που θα πρέπει να γίνουν.
Για μεγάλα μεγέθη M, N όμως ότι τιμή και να έχουμε για το K η επίδοση είναι σχεδόν σταθερή.
Επίσης, τις καλύτερες αριθμητικά επιδόσεις, έχουμε για τα μεγαλύτερα μεγέθη M, N (1024, 2048) και για μικρές μεσαίες τιμές του K.

Για τον `coalesced_A` kernel:







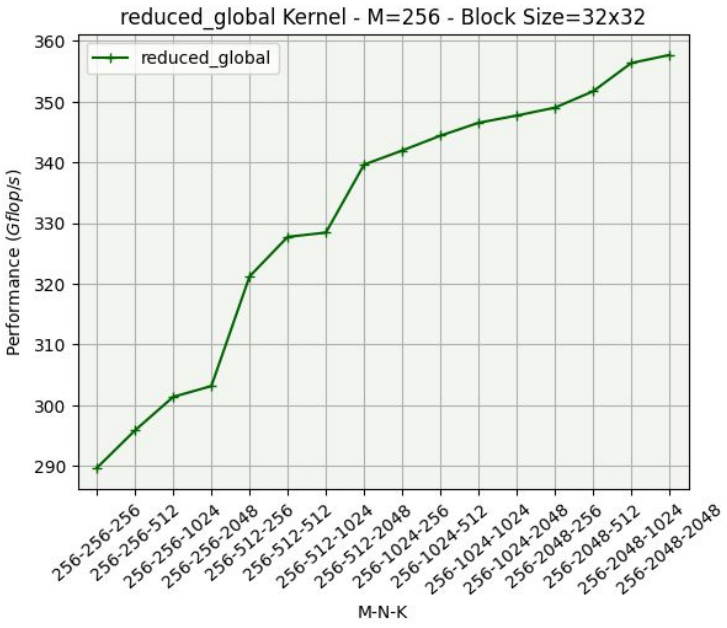


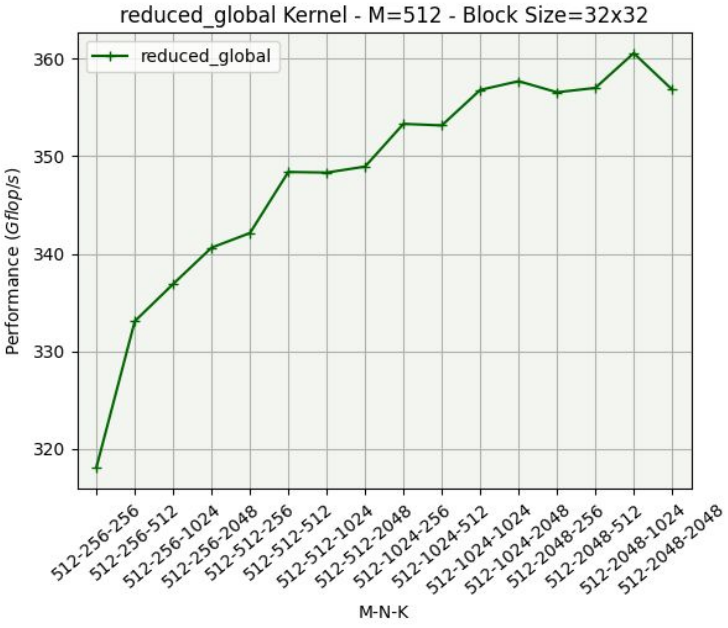
Παρατηρήσεις

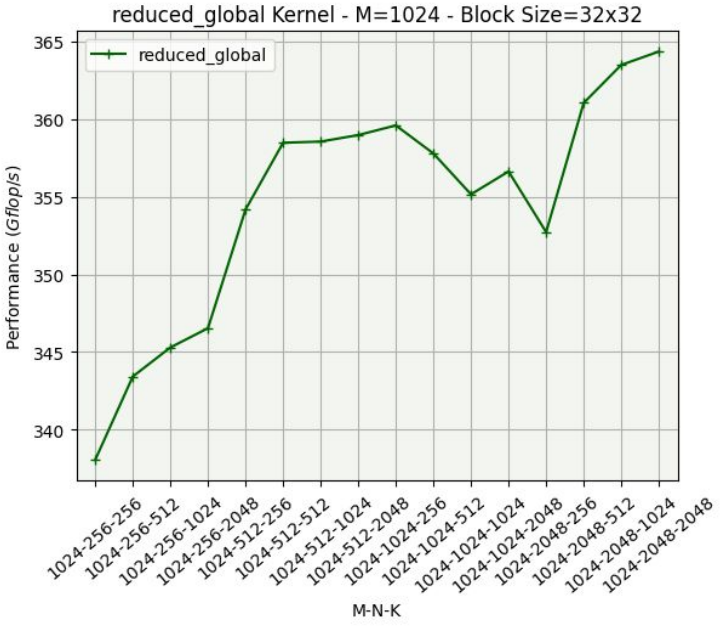
Παρατηρούμε, ότι τώρα δεν υπάρχει η ίδια συμπεριφορά με προηγουμένως, δηλαδή δεν υπάρχει τόσο μεγάλη επίδραση στην επίδοση, ανάλογα με την αύξηση του K, καθώς πλέον έχουμε ήδη σε shared memory τα στοιχεία του A που θα χρειαστούμε, όμως για τα στοιχεία του B, θα πρέπει να γίνουν πάλι προσβάσεις.

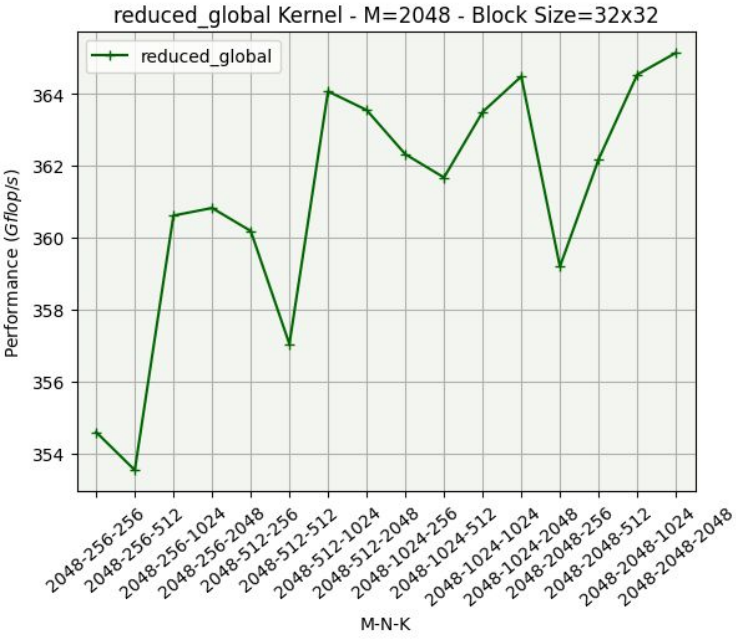
Οπότε, αν παρατηρήσουμε για κάθε μέγεθος M, έχουμε την ίδια συμπεριφορά στην γραφική, και τις καλύτερες επιδόσεις τις παίρνουμε πάλι για μεγάλα μεγέθη M,N αλλά για μικρότερα μεγέθη του K.

Για τον `reduced_global` kernel:





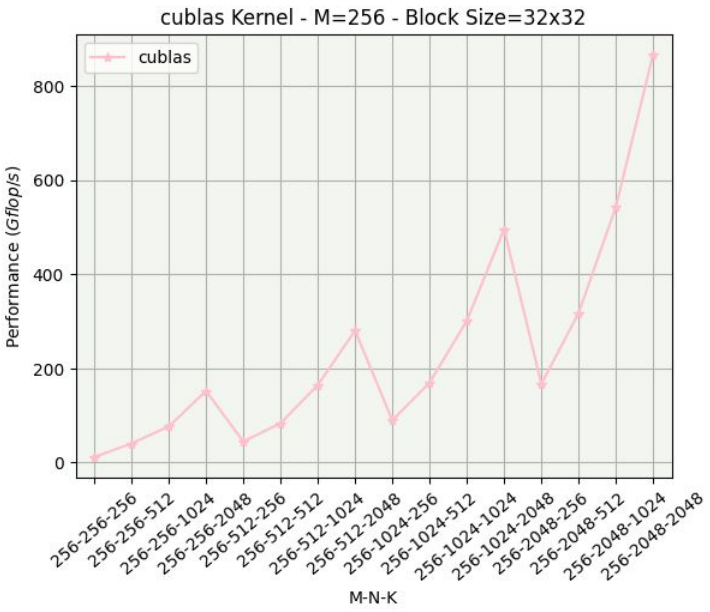


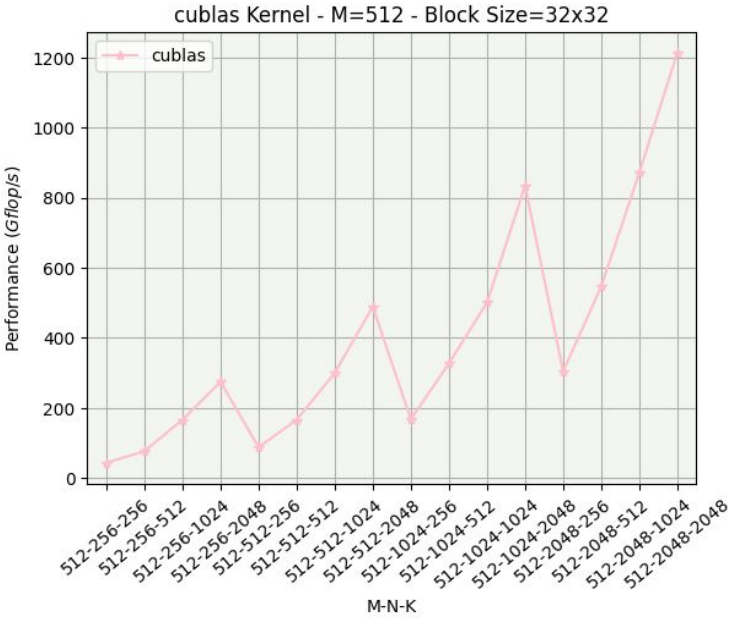


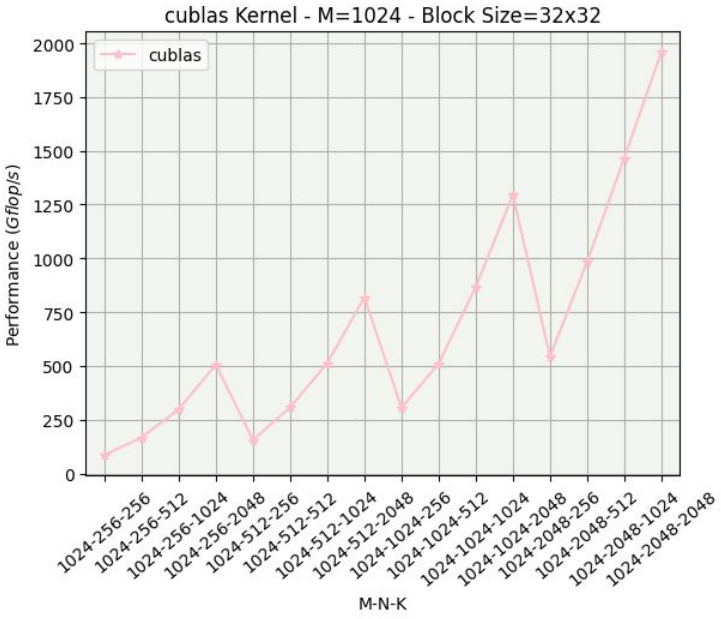
Παρατηρήσεις

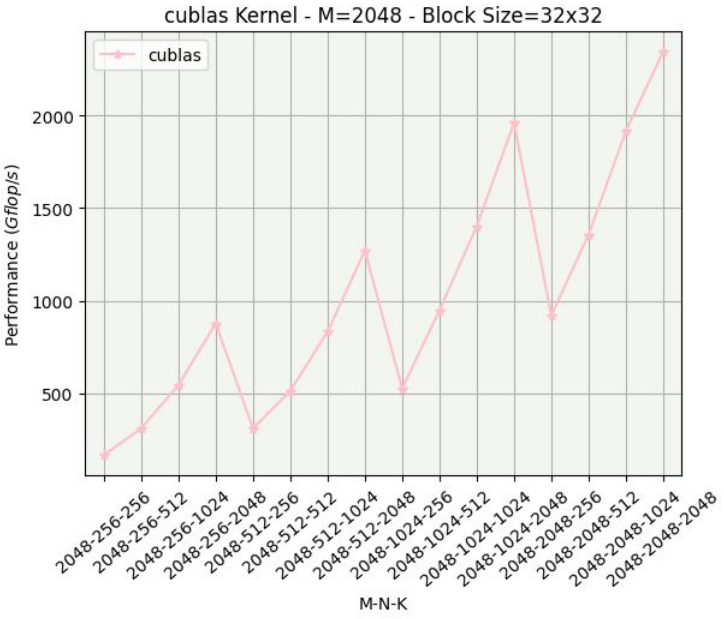
Τώρα πλέον που δεν έχουμε accesses στην global memory για τον υπολογισμό του κάθε στοιχείου, βλέπουμε ότι οι γραφικές είναι ανεξάρτητες του K, και αυξάνονται συνέχεια η επίδοση με την αύξησης των μεγεθών M,N,K. Για M=2048, δεν έχουμε την ίδια γραμμικώς αύξουσα συμπεριφορά, όμως γενικά τις καλύτερες επιδόσεις τις έχουμε για τα μεγαλύτερα μεγέθη, όπου αξιοποιούμε καλύτερα τις προσβάσεις στην μνήμη και είναι λιγότερο memory-bound ο kernel.

Για τον `cublas` kernel:







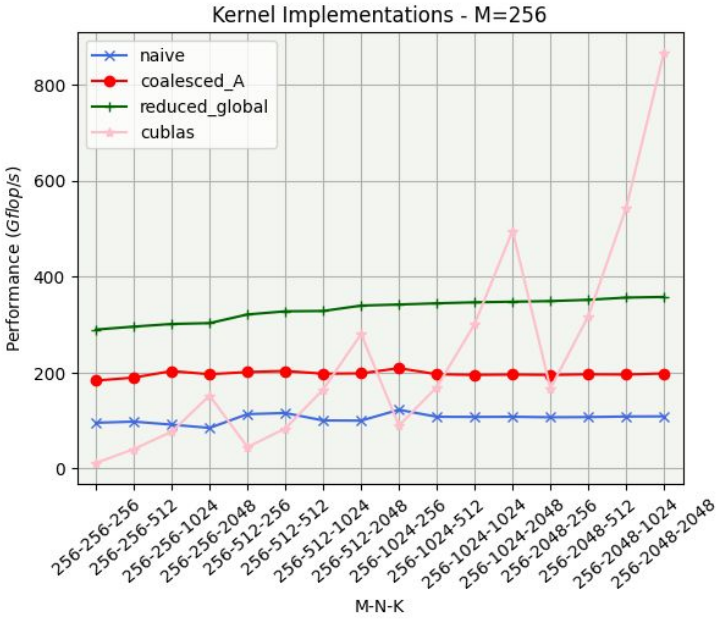


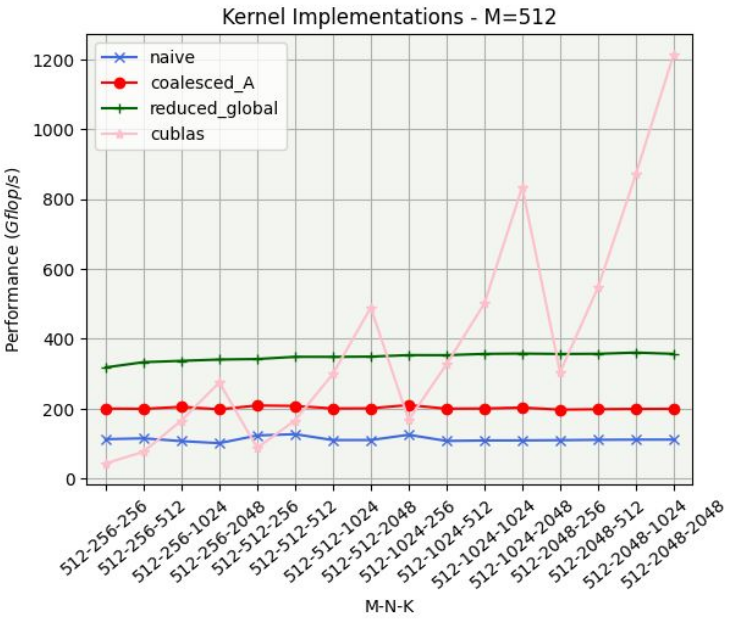
Παρατηρήσεις

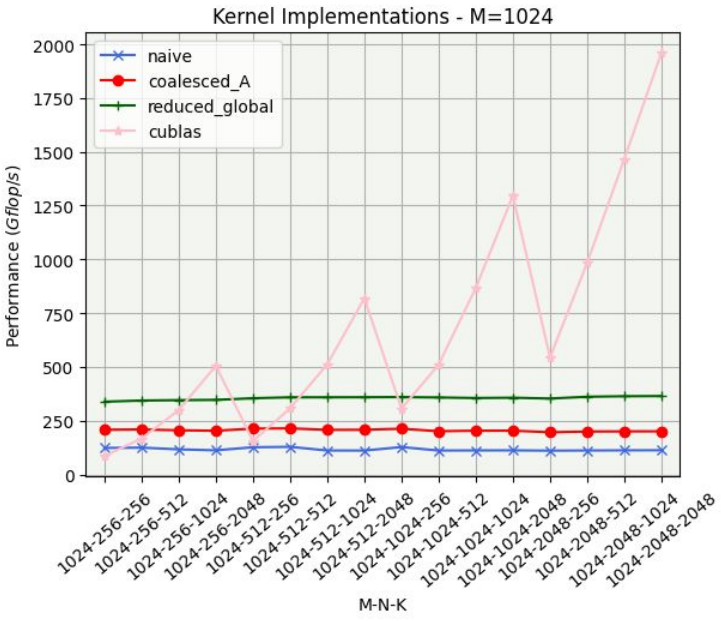
Πλέον στην υλοποίηση του cuBLAS, έχουμε αναμενόμενα τις καλύτερες επιδόσεις, και παρατηρούμε ότι υπάρχει πάλι αύξουσα τάση της επίδοσης με την αύξηση των μεγεθών M,N,K. Όμως, παρατηρούμε ότι υπάρχει μια περιοδικότητα, καθώς έχουμε σταδιακές αυξήσεις με την αύξηση του K, που είναι η ανάποδη συμπεριφορά από αυτήν που είχαμε στην naive υλοποίηση.

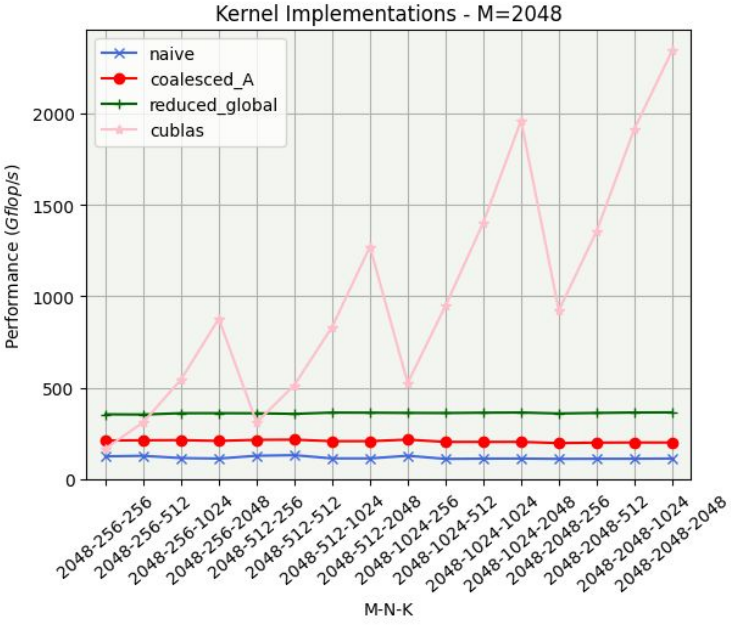
Επίσης, σε αυτήν την υλοποίηση, τις καλύτερες αριθμητικά επιδόσεις τις έχουμε για τις μεγαλύτερες τιμές των M,N,K.

Για όλους τους kernel:









Συμπεράσματα

Τέλος, έχουμε συγκεντρωτικά όλες τις υλοποιήσεις για να τις συγκρίνουμε μεταξύ τους. Όπως βλέπουμε, και είχαμε ήδη παρατηρήσει από το 1ο σενάριο μετρήσεων, η δικές μας υλοποιήσεις είναι πάντα καλύτερη η τελευταία από την προηγούμενη. Η `coalesced_A` έχει καλύτερες συνολικά επιδόσεις για όλα τα μεγέθη από την `naive`, και η `reduced_global` έχει καλύτερες συνολικά επιδόσεις από την `coalesced_A`. Όμως, αυτό που φαίνεται ξεκάθαρα είναι ότι η υλοποίηση της cuBLAS είναι πολύ καλύτερη από τις δικές μας, και η διαφορά αυτή μεγαλώνει όλο και περισσότερο για μεγαλύτερα μεγέθη πινάκων, ενώ οι επιδόσεις των δικών μας, συγκριτικά μένουν σχεδόν σταθερές.