Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Συστήματα Παράλληλης Επεξεργασίας

9ο εξάμηνο

Άσκηση 4 : Παράλληλος προγραμματισμός σε επεξεργαστές γραφικών

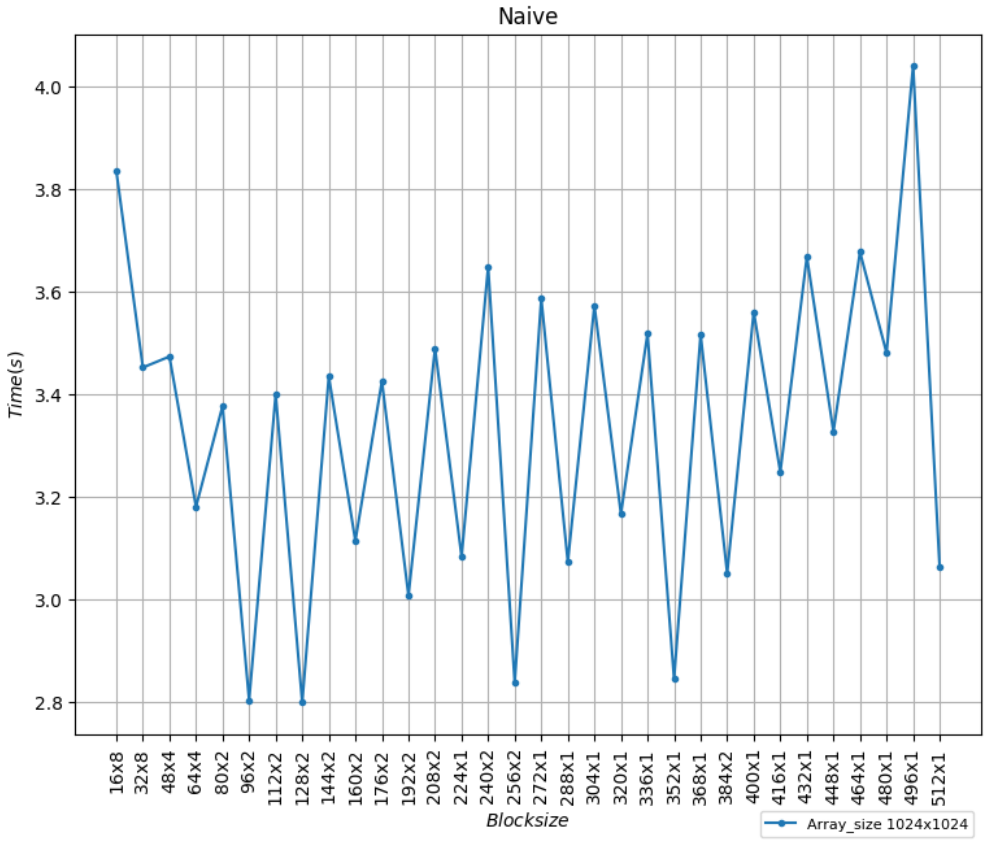
Δαζέα Ελένη	03114060
Καναβάκης Ελευθέριος	03114180

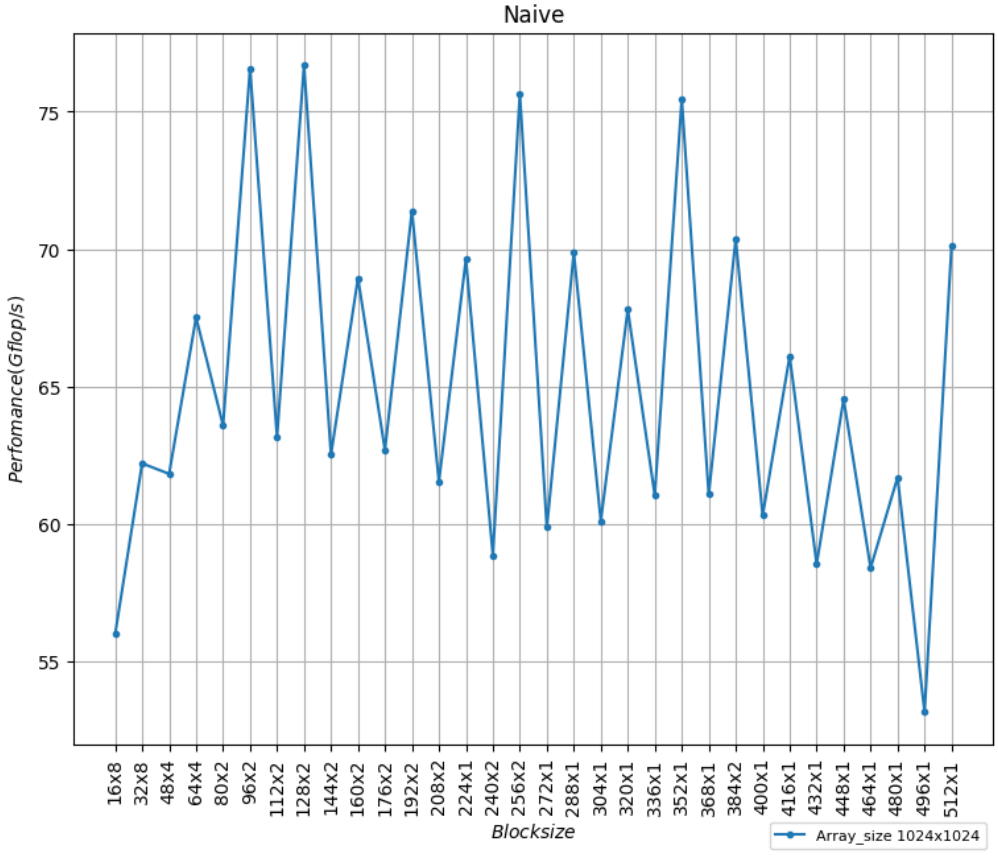
Εισαγωγή : Στην άσκηση αυτή υλοποιήσαμε τον αλγόριθμο DMM (Dense Matrix multiplication) σε επεξεργαστές γραφικών (GPUs) . Πιο συγκεκριμένα , χρησιμοποιήσαμε το προγραμματιστικό περιβάλλον CUDA της nvidia καθώς και τις κάρτες γραφικών της ουράς termis του εργαστηρίου . Πριν προχωρήσουμε όμως στην παρουσίαση των αποτελεσμάτων μας θεωρούμε σκόπιμο να πούμε λίγα πράγματα για τον αλγόριθμο DMM ο οποίος είναι ένας από τους πιο σημαντικούς υπολογιστικούς πυρήνες αλγεβρικών υπολογισμών . Ο αλγόριθμος DMM λοιπόν στην ουσία υλοποιεί την παρακάτω πράξη :
C = A×B όπου για κάθε στοιχείο C_{ij} έχουμε → $C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$.

Βασική υλοποίηση: Αρχικά, υλοποιήσαμε μια naive έκδοση πυρήνα του αλγορίθμου DMM . Στην έκδοση αυτή αρχικά περνάμε τους πίνακες A,B στην μνήμη της GPU . Υπενθυμίζουμε ότι η cuda ακολουθεί row-major σύστημα αποθήκευσης και έτσι οι πίνακες αποθηκεύονται σαν μονοδιάστατοι πίνακες κατά γραμμές . Η παρατήρηση αυτή θα φανεί πολύ χρήσιμη στην μετέπειτα ανάπτυξη και υλοποίηση του naive πυρήνα μας . Αφού λοιπόν κάναμε copy του πίνακες μας καλέσαμε τον naïve πυρήνα μας για την εκτέλεση του αλγορίθμου DMM . Στον πυρήνα αυτό αρχικά υπολογίσαμε την γραμμή και την στήλη που έχει αναλάβει το κάθε νήμα και στην συνέχεια αφού επιβεβαιώσαμε ότι το νήμα αυτό δεν αντιστοιχεί σε ανενεργό νήμα υπολογίσαμε ένα στοιχείο του πίνακα C . Για τον υπολογισμό της γραμμής και της στήλης του κάθε νήματος χρησιμοποιήσαμε τόσο τα thread,block ids όσο και τις διαστάσεις του block .

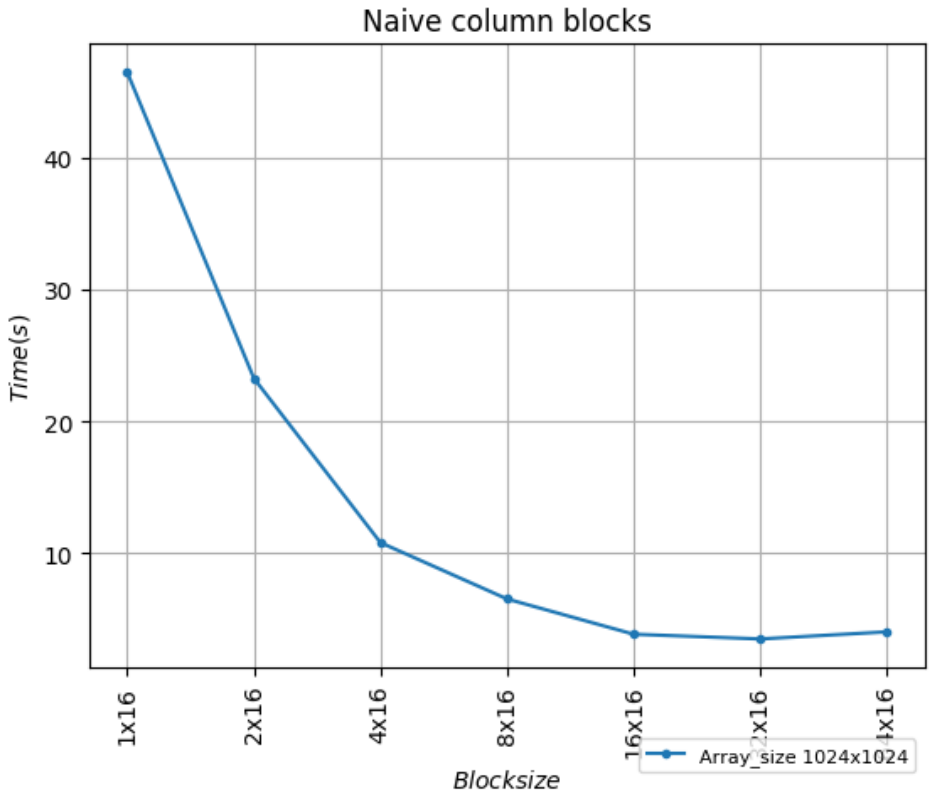
Φυσικά , οι διαστάσεις του block αποτέλεσαν και βασικό κριτήριο της αξιολόγησης του πυρήνα μας καθώς μελετήσαμε την επίδοση του για διάφορες διαστάσεις block . Πιο συγκεκριμένα , δοκιμάσαμε τόσο 2D όσο και 1D blocks . Στην μία διάσταση δώσαμε τιμές από το 16 έως με το 512 με βήμα 16 , ενώ στην άλλη δώσαμε διαστάσεις με τιμές 1,2,4,8,16,32,64 . Φυσικά , για τις διαστάσεις του μπλοκ ήταν απαραίτητο να εξασφαλίσουμε ότι block_sizex*block_sizey < 1024 . Μάλιστα , δοκιμάσαμε και τον αντίστροφο συνδυασμό διαστάσεων για το block . Δηλαδή , βάλαμε σαν διάσταση χ το διάνυσμα [1,2,4,8,16,32,64] και στα διάσταση y το διάνυσμα από 16 έως 512 με βήμα 16 . Τέλος , για τα δύο σενάρια μετρήσεων για λόγους πρακτικότητας επιλέξαμε την καλύτερη μέτρηση για κάθε διαφορετική διάσταση x και την παρουσιάσαμε παρακάτω με την μορφή γραφικών παραστάσεων .

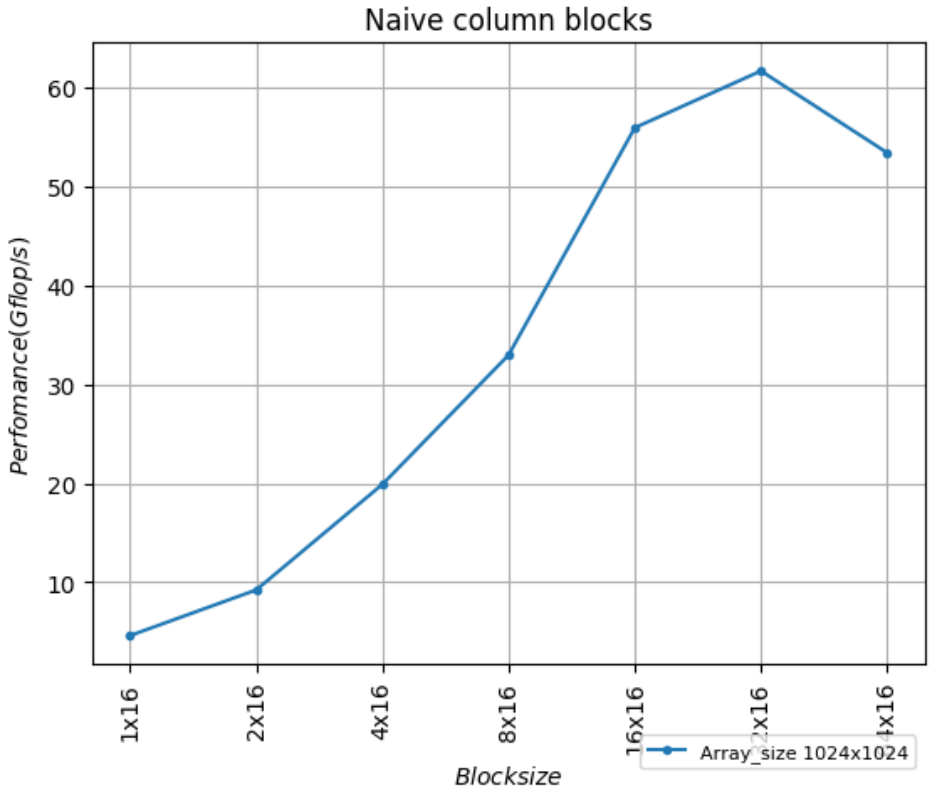
Γραφικές παραστάσεις χρόνου και επίδοσης με μεγαλύτερες τιμές στον άξονα x.





Γραφικές παραστάσεις χρόνου και επίδοσης με μεγαλύτερες τιμές στον άξονα y.





Με μία πρώτη ματιά θα μπορούσε κανείς να πει πως τα δύο παραπάνω σενάρια μετρήσεων δεν μπορούν συγκριθούν καθώς οι διαστάσεις είναι διαφορετικές και στις περισσότερες περιπτώσεις στο δεύτερο σενάριο μετρήσεων νήματα ανά block είναι πολύ λιγότερα από ότι στο πρώτο σενάριο . Υπενθυμίζουμε όμως έχουμε επιλέξει τις καλύτερες μετρήσεις για κάθε διάσταση x (δηλαδή η 1x16 είχε καλύτερη επίδοση από την 1x256) . Για τα παραπάνω δύο σενάρια μετρήσεων λοιπόν εύκολα μπορεί να παρατηρήσει κανείς ότι το πρώτο σενάριο (δηλαδή αυτό με την μεγαλύτερη διάσταση στον άξονα x) έχει πολύ καλύτερη επίδοση από το δεύτερο . Αυτό πιθανώς να οφείλεται στο ότι οι προσβάσεις στον πίνακα A δεν συνενώνονται . Παράλληλα , παρατηρούμε από τις παραπάνω γραφικές παραστάσεις ότι τα καλύτερα αποτελέσματα τ παίρνουμε για αριθμό νημάτων πολλαπλάσιο του 32 . Το γεγονός αυτό είναι απολύτως φυσιολογικό καθώς και τα wraps είναι πολλαπλάσια του 32 . Ως εκ τούτου αξιοποιείται το γεγονός ότι οι προσβάσεις στην μνήμη για τον πίνακα B συνενώνονται .

Τέλος , για τον υπολογισμό του occupancy των SMs θα χρησιμοποιήσουμε το cuda occupancy calculator το οποίο μπορεί κανείς να βρει στον παρακάτω σύνδεσμο : https://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls .

Για την χρήση του εργαλείου αυτού χρειάζεται να γνωρίζουμε 3 πράγματα . Την αρχιτεκτονική της κάρτας γραφικών , τον αριθμό των νημάτων ανά μπλοκ καθώς και τον αριθμό των registers ανά νήμα . Τα πρώτα δύο είναι στοιχεία που μπορούν να συμπληρωθούν εύκολα καθώς για την αρχιτεκτονική της κάρτας γραφικών γνωρίζουμε ότι είναι Fermi(2.0 compute capability) ενώ για τον αριθμό των νημάτων ανά μπλοκ γνωρίζουμε ότι προκύπτει από το γινόμενο των διαστάσεων του μπλοκ . Για τον αριθμό των καταχωρητών ανά μπλοκ χρησιμοποιήσαμε το flag ptxas info=-v έτσι ώστε να μας επιστραφούν σχετικές πληροφορίες κατά το nvcc compilation . Τα αποτελέσματα αυτού , για ΟΛΟΥΣ τους πυρήνες που γράψαμε φαίνονται παρακάτω :

```
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z17dmm_gpu_coalescedPKfS0_Pfmmm'
for 'sm_20'
ptxas info    : Function properties for _Z17dmm_gpu_coalescedPKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 18 registers, 80 bytes cmem[0]
ptxas info    : Compiling entry function '_Z14dmm_gpu_cublasPKfS0_Pfmmm'
for 'sm_20'
ptxas info    : Function properties for _Z14dmm_gpu_cublasPKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 2 registers, 80 bytes cmem[0]
ptxas info    : Compiling entry function '_Z13dmm_gpu_naivePKfS0_Pfmmm' for
'sm_20'
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, 80 bytes cmem[0]
ptxas info    : Compiling entry function '_Z13dmm_gpu_shmemPKfS0_Pfmmm' for
'sm_20'
ptxas info    : Function properties for _Z13dmm_gpu_shmemPKfS0_Pfmmm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
```

```
ptxas info    : Used 19 registers, 80 bytes cmem[0]

ptxas info    : 0 bytes gmem
ptxas info    : 0 bytes gmem
```

Από τα παραπάνω αποτελέσματα συμπεραίνουμε ότι για το naïve kernel έχουμε 16 καταχωρητές , για το coalesced έχουμε 18 καταχωρητές ενώ για το tiled έχουμε 19 καταχωρητές . Τέλος , το occupancy εξαρτάται και από το shared memory . Βέβαια , για την naïve υλοποίηση το μέγεθος της μνήμης αυτή ισούται με μηδέν . Στην συνέχεια παραθέτουμε σε πίνακα το occupancy των SMs για διάφορους αριθμούς νημάτων .

#Threads	***Occupancy (%)***
1	17
2	17
4	17
8	17
16	17
32	17
64	33
96	50
128	67
256	100
512	100
768	100
800	52
832	54
864	56
896	58
928	60
960	63
992	65
1024	67

Από τα παραπάνω παρατηρούμε ότι για την naïve έκδοση μέγιστο occupancy των SMs επιτυγχάνουμε για αριθμούς νημάτων από 164 έως και 768 νήματα ανά μπλοκ. Το συμπέρασμα αυτό συνάδει αρκετά με τα αποτελέσματα των μετρήσεων που παρουσιάσαμε στις παραπάνω γραφικές παραστάσεις .

Διαστάσεις βέλτιστου block → 128x2 !

Coalesced υλοποίηση: Στην συνέχεια , στην προσπάθεια μας να βελτιστοποιήσουμε την συμπεριφορά της naïve υλοποίησης του DMM θεωρήσαμε χρήσιμο να δοκιμάσουμε να επιτύχουμε συνένωση τον προσβάσεων του πίνακα A στην μνήμη . Για να το πετύχουμε αυτό αρχικά πήραμε τον ανάστροφο του πίνακα A (transpose) και στην συνέχεια στον πυρήνα υπολογισμού ενός στοιχείου του C αλλάξαμε τα memory accesses στον πίνακα A έτσι ώστε κάθε νήμα να μην κάνει access διαδοχικά στοιχεία αλλά στοιχεία που απέχουν μεταξύ τους κατά M. Ουσιαστικά , για την συνένωση των προσβάσεων στην μνήμη για έναν πίνακα προσπαθούμε για τα νήματα του ίδιου wrap να φέρνουμε τα δεδομένα με ένα memory access . Αυτό επιτυγχάνεται με την τεχνική που περιγράψαμε παραπάνω καθώς κάθε νήμα για να φέρει ένα στοιχείο του πίνακα A (το οποίο χρειάζεται σε κάθε iteration του for loop που εκτελεί) στην ουσία δεν φέρνει ένα στοιχείο αλλά ένα chunk από στοιχεία . Στην naïve υλοποίηση τα υπόλοιπα στοιχεία ήταν άχρηστα ενώ στην υλοποίηση αυτή τα στοιχεία που έρχονται σε ένα chunk αξιοποιούνται από τα υπόλοιπα νήματα του wrap (υπενθυμίζουμε ότι τα νήματα του ίδιου wrap βρίσκονται για τον ίδιο κύκλο ρολογιού στην ίδια εντολή) . Για την καλύτερη εξήγηση των παραπάνω παραθέτουμε και ένα παράδειγμα της αλλαγής που πραγματοποιήσαμε σε σχέση με την naïve υλοποίηση .

Έστω πίνακας 4x4 με τα εξής στοιχεία :

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Στην cuda έχουμε row-major αποθήκευση επομένως τα δεδομένα αποθηκεύονται στην μνήμη ως : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 .

Τα παραπάνω δεδομένα αποθηκεύονται στο thread όπως φαίνεται παρακάτω :

Thread 1	1	2	3	4
Thread 2	5	6	7	8
Thread 3	9	10	11	12
Thread 4	13	14	15	16

Φυσικά , η παραπάνω κατανομή δεν βοηθάει καθώς για παράδειγμα το thread 1 χρειάζεται τα στοιχεία 1,5,9,13 όμως έχει στην διάθεση του τα στοιχεία 1,2,3,4.

Παίρνοντας λοιπόν τον ανάστροφο του παραπάνω πίνακα έχουμε :

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

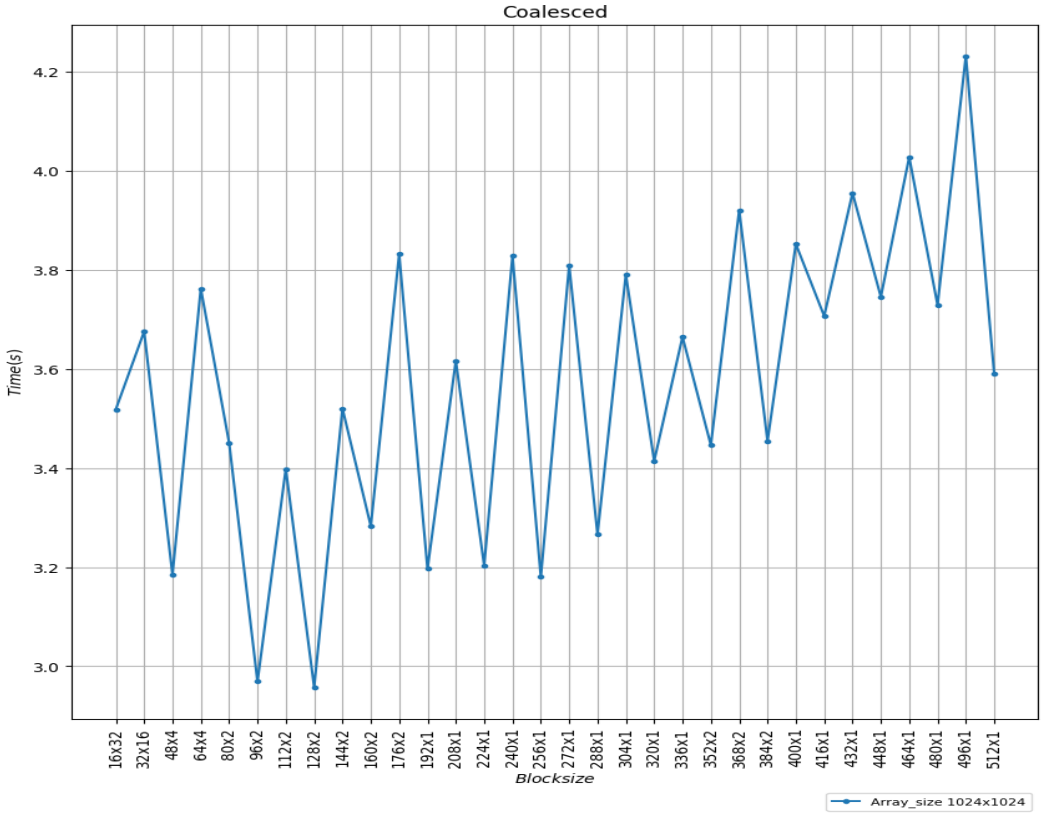
Τα στοιχεία αποθηκεύονται στη μνήμη ως: 1, 5, 9, 13, 6, 10, 14, 3, 7, 11, 15, 4, 8, 12, 16

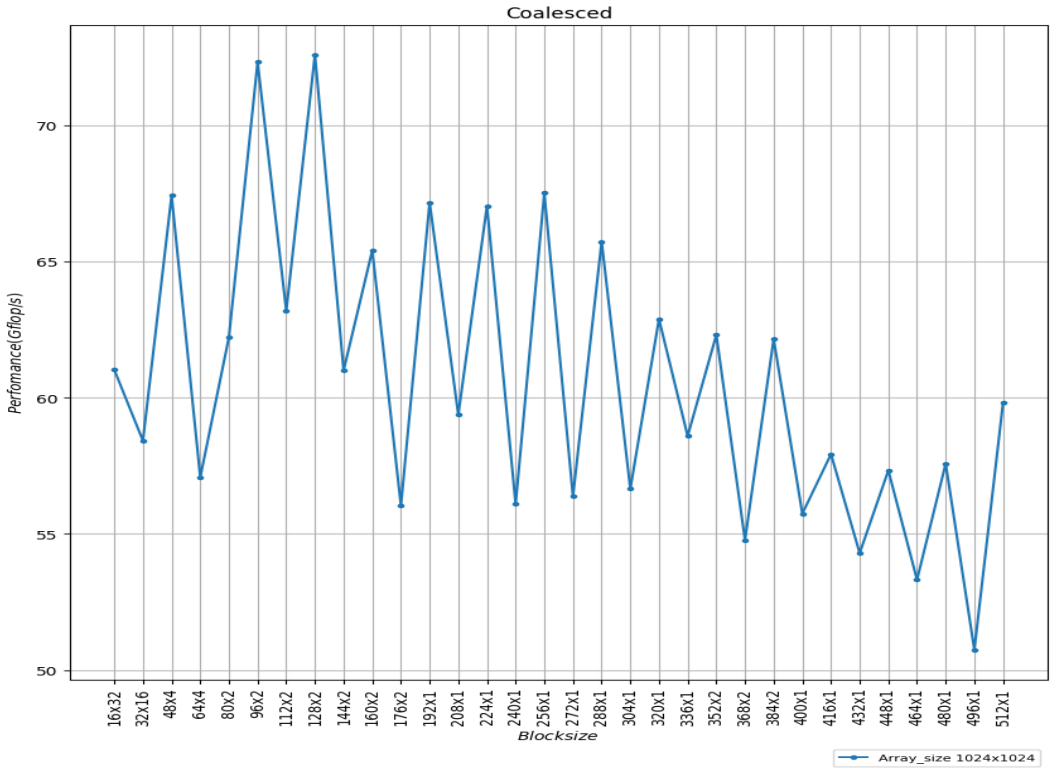
Και στα thread ως:

Thread 1	1	5	9	13
Thread 2	2	6	10	14
Thread 3	3	7	11	15
Thread 4	4	8	12	16

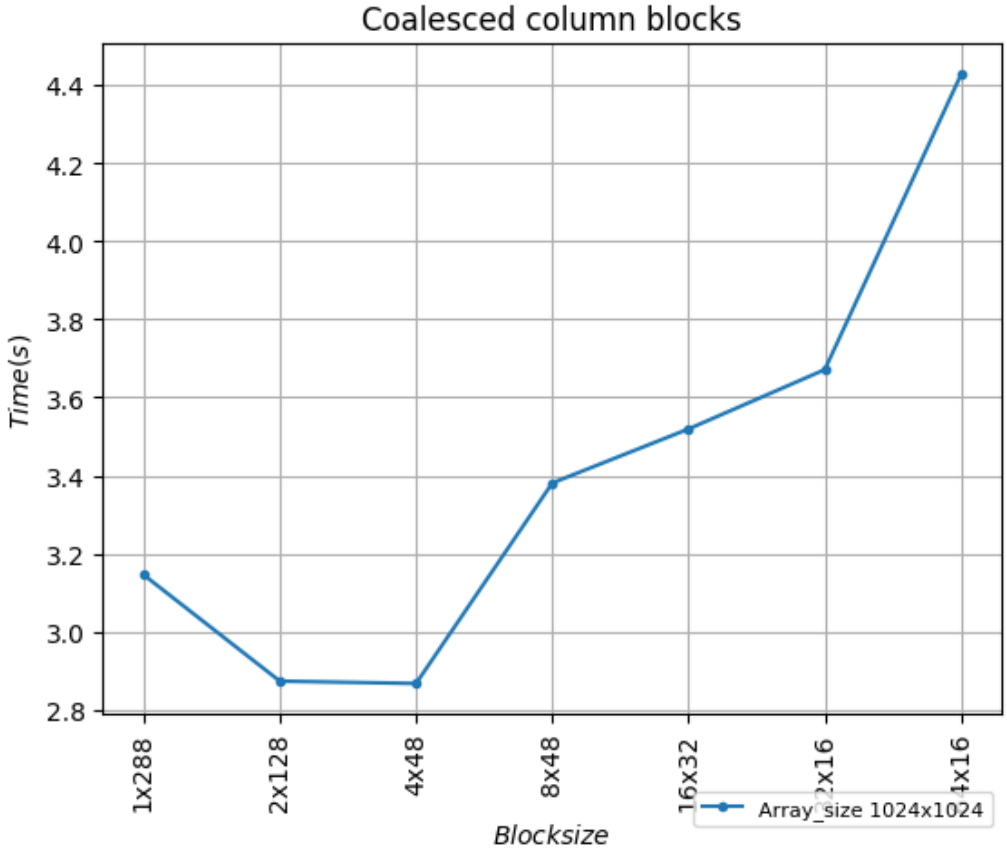
Με βάση το παραπάνω λοιπόν υλοποιείται η συνένωση των προσβάσεων στην μνήμη . Παράλληλα , απαραίτητη προϋπόθεση για την επιτυχία της παραπάνω υλοποίησης είναι το πλήθος των νημάτων να είναι πολλαπλάσιο του 32 (δηλαδή του wrap size) . Στην συνέχεια , παραθέτουμε γραφικές παραστάσεις για τα σενάρια μετρήσεων που περιγράψαμε και παραπάνω .

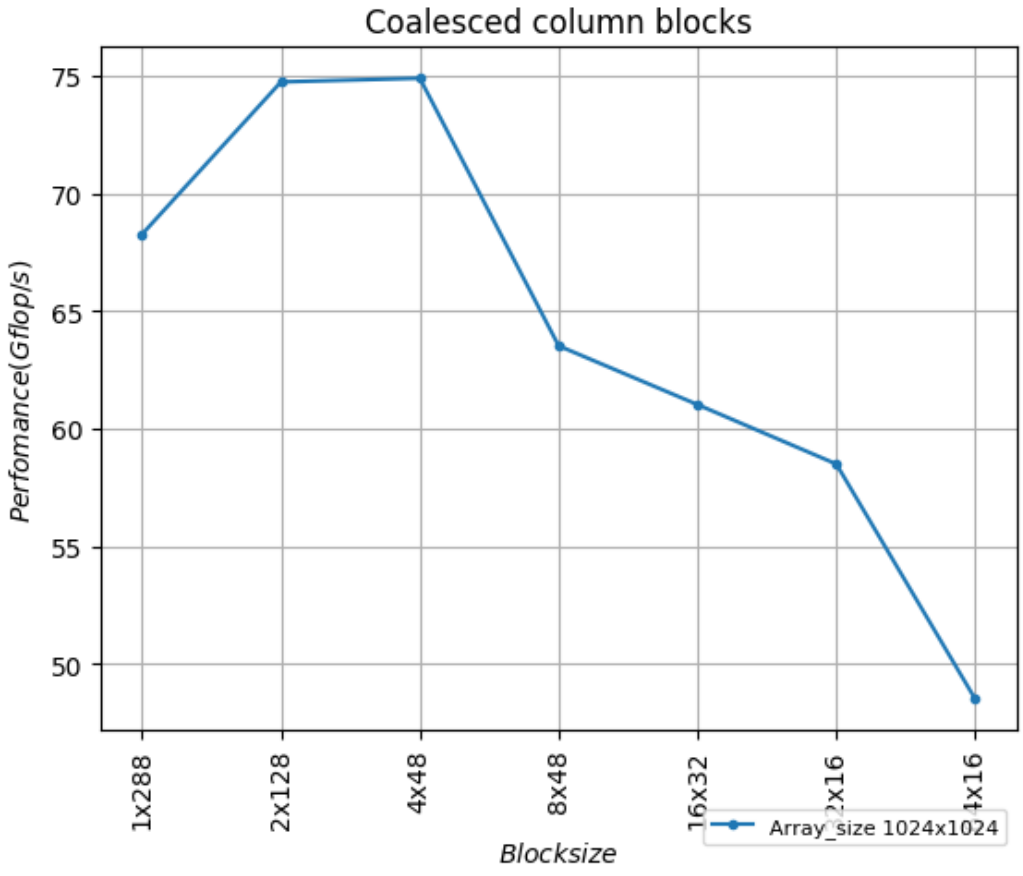
Γραφικές παραστάσεις χρόνου και επίδοσης με μεγαλύτερες τιμές στον άξονα x.



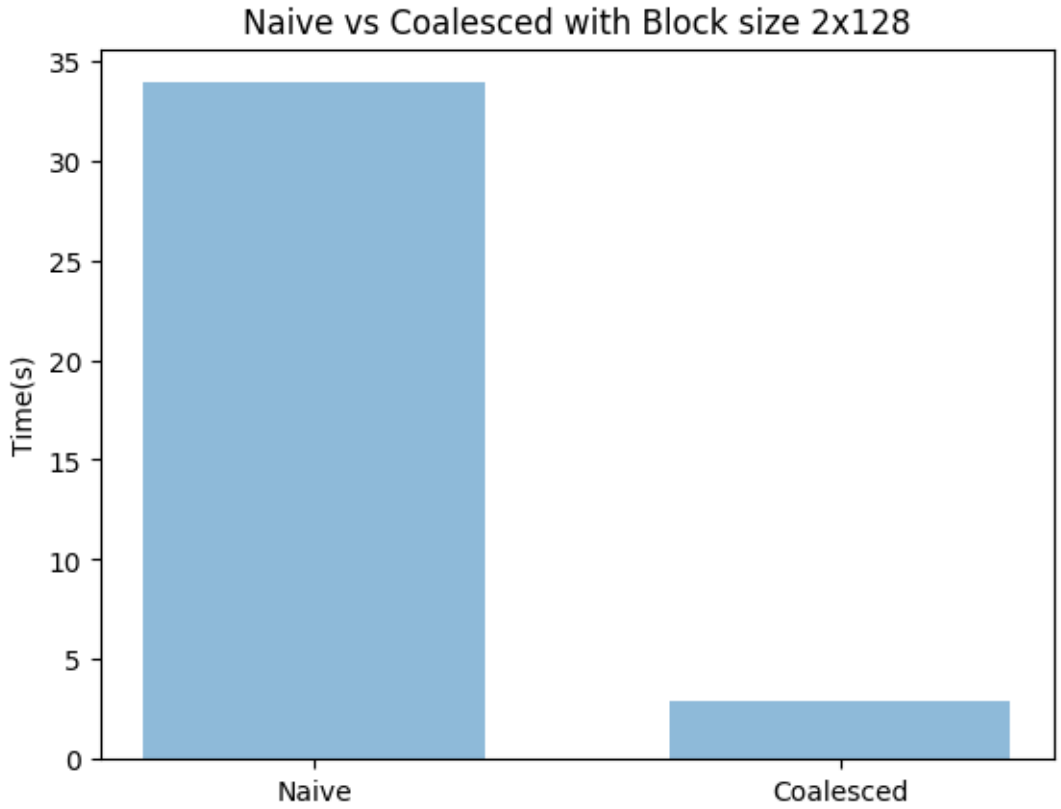


Γραφικές παραστάσεις χρόνου και επίδοσης με μεγαλύτερες τιμές στον άξονα y.





Το πρώτο πράγμα που παρατηρούμε από τις παραπάνω γραφικές παραστάσεις είναι η επίδοση είναι αρκετά κακή για αριθμό νημάτων που δεν είναι πολλαπλάσιο του 32 . Μάλιστα , σε σχέση με την naïve υλοποίηση παρατηρούμε και χειρότερη επίδοση για τις διαστάσεις αυτές . Φυσικά , το γεγονός αυτό είναι απόλυτα φυσιολογικό καθώς όπως προείπαμε για επιτυχημένη συνένωση προσβάσεων θέλουμε το πλήθος των νημάτων να είναι πολλαπλάσιο του 32 και έτσι για τις περιπτώσεις που αυτό δεν ισχύει έχουμε αντίθετα αποτελέσματα . Το δεύτερο πράγμα το οποίο παρατηρούμε είναι μία μικρή βελτίωση επίδοσης για κάποιες διαστάσεις . Το πιο σημαντικό που παρατηρούμε όμως είναι ότι οι τραγικά κακές επιδόσεις που είχαμε στον naïve για το δεύτερο σενάριο μετρήσεων έχουν πλέον εξαλειφθεί και φτάνουμε πλέον στο σημείο το δεύτερο σενάριο μετρήσεων να μας δίνει τα καλύτερα αποτελέσματα επίδοσης . Για να κάνουμε πιο σαφές το παραπάνω παραθέτουμε μία ενδεικτική γραφικής σύγκρισης αποτελεσμάτων μεταξύ του naïve και του coalesced πυρήνα .



Από την παραπάνω γραφική είναι πλέον πολύ εμφανείς η χαοτική διαφορά σε επίδοση . Τέλος , με χρήση και πάλι του occupancy calculator θα υπολογίσουμε την χρησιμοποίηση των SMs για διαφορετικούς αριθμούς νημάτων . Υπενθυμίζουμε ότι για τον συγκεκριμένο πυρήνα έχουμε registers = 18 και shared memory = 0.

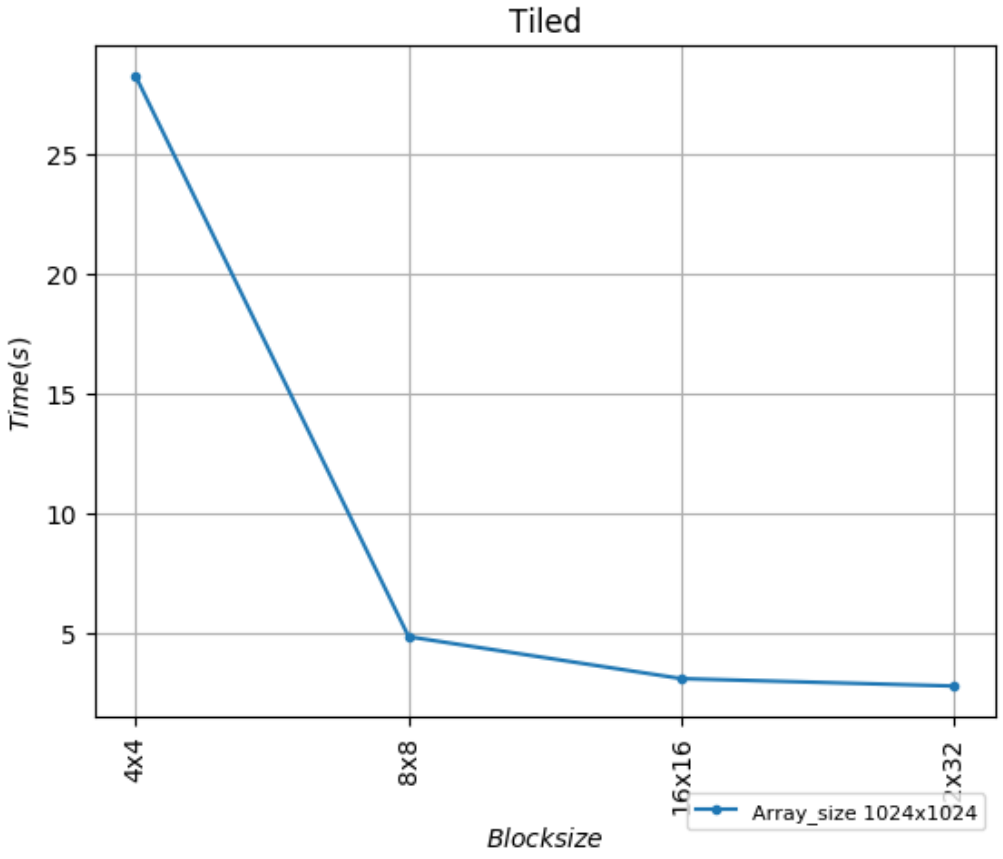
#Threads	*Occupancy (%)*
1	17
2	17
4	17
8	17
16	17
32	17
64	33
96	50
128	67
256	100
512	100
768	100
800	52
832	54
864	56
896	58
928	60
960	63
992	65
1024	67

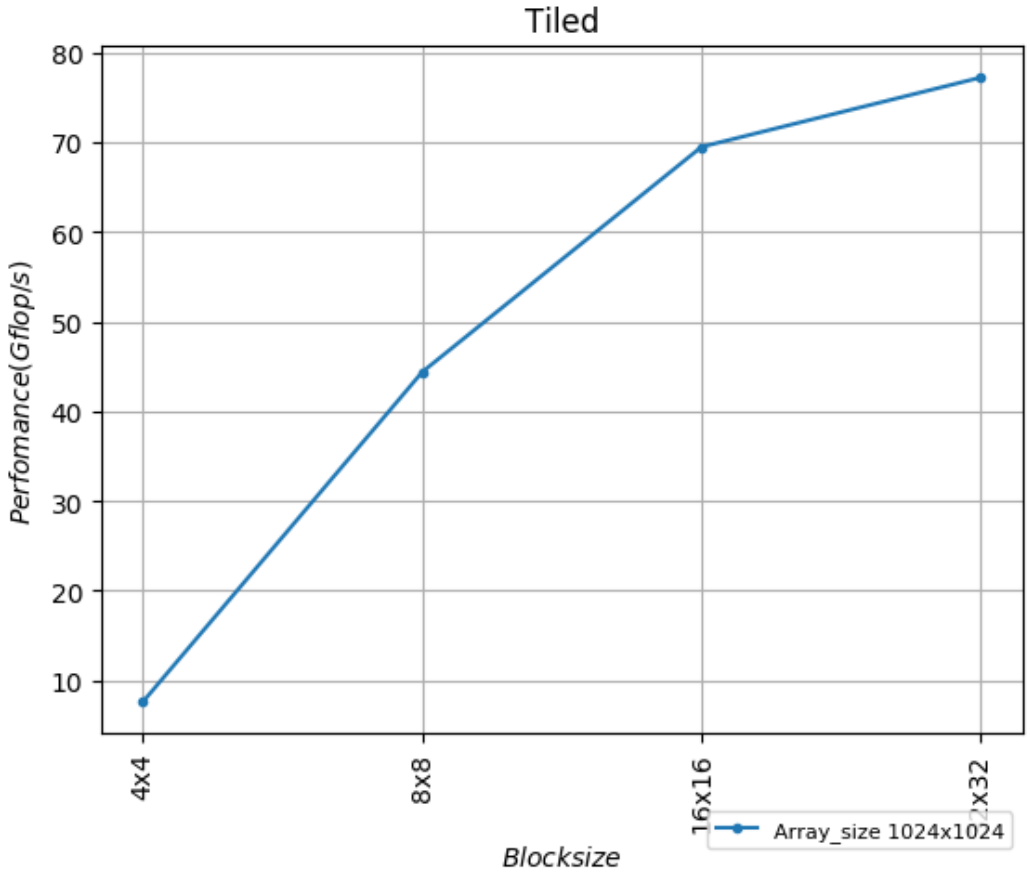
Ο παραπάνω πίνακας στην ουσία ταυτίζεται με το occupancy του naïve kernel . Επιπρόσθετα , παρατηρούμε ότι την καλύτερη επίδοση την παίρνουμε σε περιοχές με μέγιστο SM occupancy , γεγονός το οποίο είναι απόλυτα αναμενόμενο .

Διαστάσεις βέλτιστου block → 4x48 !

Tiled υλοποίηση: Στην συνέχεια, στην προσπάθεια μας για περαιτέρω βελτιστοποίηση των παραπάνω υλοποιήσαμε έναν tiled πυρήνα στον οποίο τα στοιχεία των πινάκων A,B γίνονται prefetch στην τοπική μνήμη (shared memory) των SMs . Παράλληλα , και στην έκδοση αυτή εξασφαλίζεται συνένωση των προσβάσεων στην μνήμη . Για την υλοποίηση αυτή δοκιμάσαμε μόνο τετραγωνικά blocks καθώς οι διαστάσεις των tiles είναι υποχρεωτικά τετραγωνικές . Στην συνέχεια , παρουσιάζουμε γραφικές παραστάσεις των μετρήσεων που εξάγαμε από την εκτέλεση του παραπάνω πυρήνα για διαστάσεις πινάκων M=N=K=1024.

Γραφικές παραστάσεις χρόνου και επίδοσης για τετραγωνικές διαστάσεις block.





Από τις παραπάνω γραφικές παραστάσεις , παρατηρούμε μικρή βελτίωση σε σχέση με την προηγούμενη έκδοση . Ο λόγος που η βελτίωση αυτή είναι μικρή πιθανώς να οφείλεται στο περιορισμένο compute capability που έχουν οι κάρτες γραφικών (παλιές εκδόσεις) . Παράλληλα , όπως σχολιάζουμε παρακάτω σημαντική μείωση της επίδοσης επιφέρει και το δυναμικό allocate των shared πινάκων .

Τέλος , με χρήση και πάλι του occupancy calculator θα υπολογίσουμε την χρησιμοποίηση των SMs για διαφορετικούς αριθμούς νημάτων . Υπενθυμίζουμε ότι για τον συγκεκριμένο πυρήνα έχουμε registers = 19 . Στην υλοποίηση αυτή το SM occupancy επηρεάζεται και από την shared memory per block που υπάρχει πλέον .

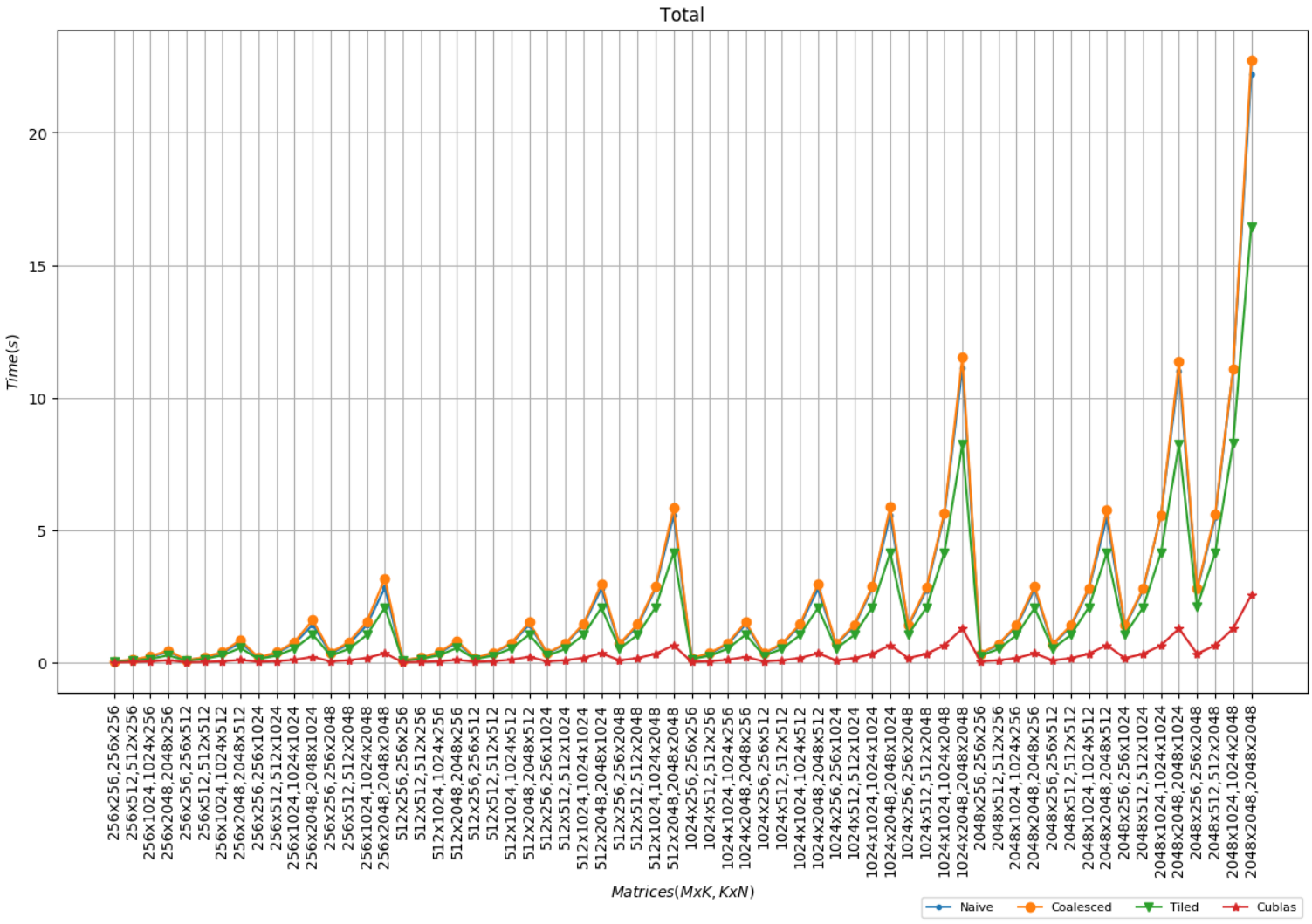
#Threads	Shared Memory (Bytes)	Occupancy(%)
16	128	17
64	512	33
256	2048	100
1024	8192	67

Διαστάσεις βέλτιστου block → 32x32 !

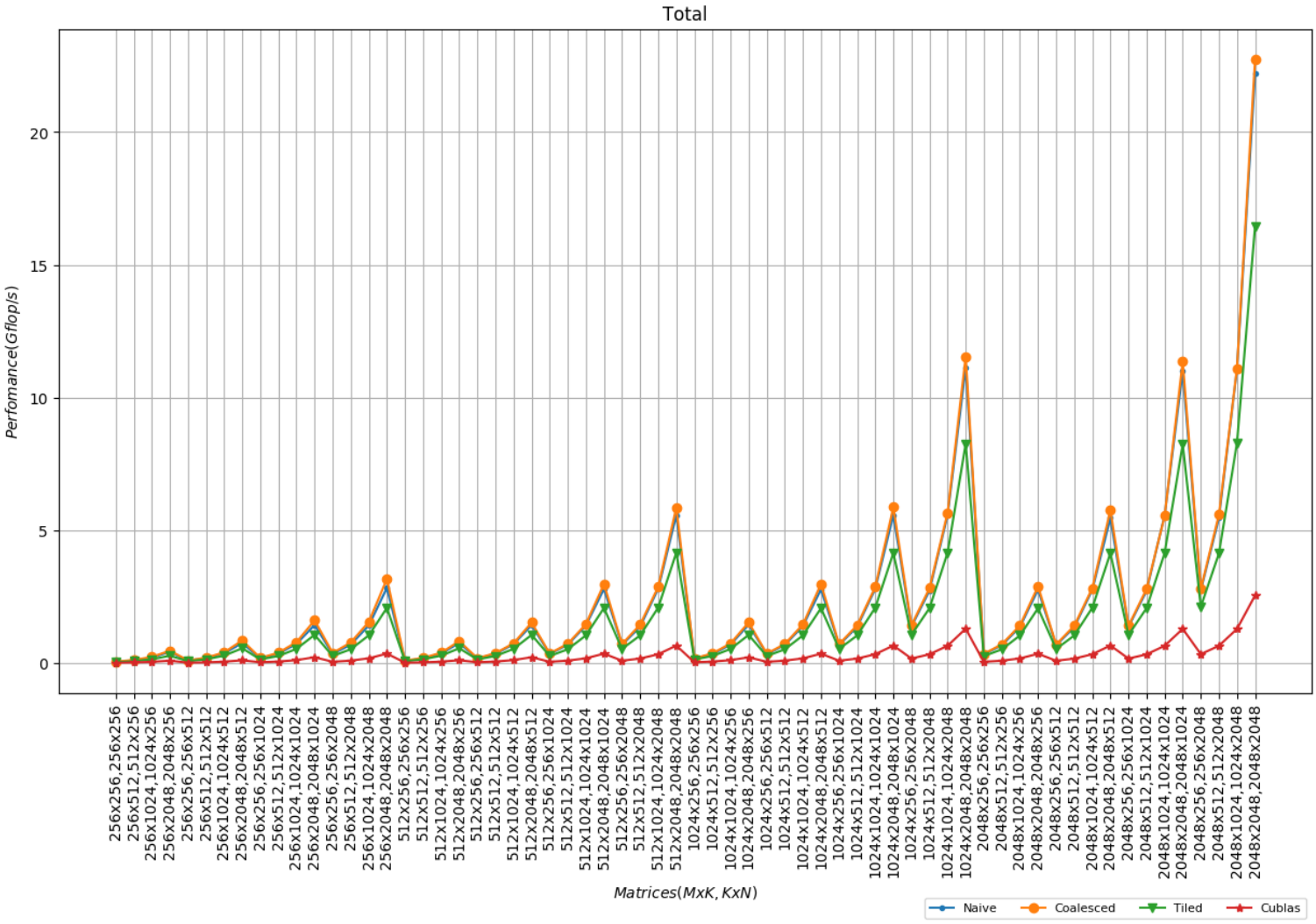
Χρήση της βιβλιοθήκης cuBLAS : Τέλος , τροποποιήσαμε κατάλληλα τον κώδικα μας ώστε να χρησιμοποιήσομε την συνάρτηση cublasSgemm() της βιβλιοθήκης cuBLAS για την υλοποίηση του ζητούμενου πολλαπλασιασμού . Η βιβλιοθήκη cuBLAS αποτελεί υλοποίηση για κάρτες γραφικών της NVIDIA . Σημαντική δυσκολία της συγκεκριμένης υλοποίησης αποτέλεσε το γεγονός ότι η cuBLAS ουσιαστικά είναι γραμμένη σε Fortran . Η Fortran ακολουθεί column-major αποθήκευση των δεδομένων σε αντίθεση με την cuda. Ως εκ τούτου προβληματιστήκαμε αρκετά για τον τρόπο με τον οποίο θα δώσουμε τους πίνακες έτσι ώστε να πάρουμε το σωστό αποτέλεσμα . Εν τέλει καταλήξαμε στο ότι θα περάσουμε τους πίνακες στην κανονική αρχική τους μορφή και θα δηλώσουμε στην συνάρτηση πως οι πίνακες βρίσκονται σε column-major μορφή (CUBLAS_OP_N) . Στην συνέχεια , για να πάρουμε το σωστό αποτέλεσμα αρκεί να δώσουμε πρώτα τον πίνακα B και μετά τον πίνακα A . Αυτό , θα μας δώσει σαν αποτέλεσμα τον πίνακα C . Παράλληλα , η cublasSgemm() υλοποιεί την πράξη C=aop(A)op(B) + bC . Έτσι , δώσαμε στο όρισμα a την τιμή 1.0 ενώ στο όρισμα b την 0.0 . Η cublasSgemm() αναμένουμε να έχει πολύ καλύτερα αποτελέσματα από τις παραπάνω υλοποιήσεις . Τα αποτελέσματα αυτά θα τα δούμε παρακάτω όπου θα συγκρίνουμε συνολικά τις υλοποιήσεις μας. Τέλος , θα θέλαμε να σχολιάσουμε ότι για την υλοποίηση του DMM αρκεί μία κλήση συνάρτησης και ένα initialization του cublas handle . Το γεγονός αυτό σε συνδυασμό με τα πολύ καλά αποτελέσματα κάτι την βιβλιοθήκη αυτή ένα εξαιρετικό εργαλείο.

Συνολική σύγκριση υλοποιήσεων : Στο σημείο αυτό θα συγκρίνουμε όλες τις παραπάνω υλοποιήσεις παίρνοντας για τις 3 πρώτες τα καλύτερα block sizes όπως προέκυψαν από τις δοκιμές μας παραπάνω . Για την cublasSgemm() το block size δεν επηρεάζει την επίδοση καθώς δεν έχουμε έλεγχο πάνω σε αυτές τις παραμέτρους . **Πριν παραθέσουμε τις γραφικές μας πρέπει να τονίσουμε ένα σημαντικό trick που βελτίωσε αισθητά την απόδοση του tiled** . Προηγουμένως , είχαμε δυναμικό allocate των πινάκων που ήταν shared στα tiles καθώς δεν γνωρίζαμε εξ αρχής το μέγεθος τους (εφόσον πειραματιζόμασταν με το μέγεθος του block size – tile width) . Στο σημείο αυτό έχουμε αποφασίσει όμως ότι βέλτιστο αποτελεί το block size 32x32 . Το γεγονός αυτό μας δίνει το δικαίωμα να δηλώσουμε στατικά τους πίνακες localA και localB στους οποίους γίνονται prefetch τα δεδομένα των A,B . Το γεγονός αυτό οδήγησε σε σημαντική αύξηση της επίδοσης της tiled υλοποίησης . Στην συνέχεια λοιπόν, παραθέτουμε γραφικές παραστάσεις για M,N,K ∈ [256,512,1024,2048] !

Γραφική παράσταση του χρόνου σε seconds !



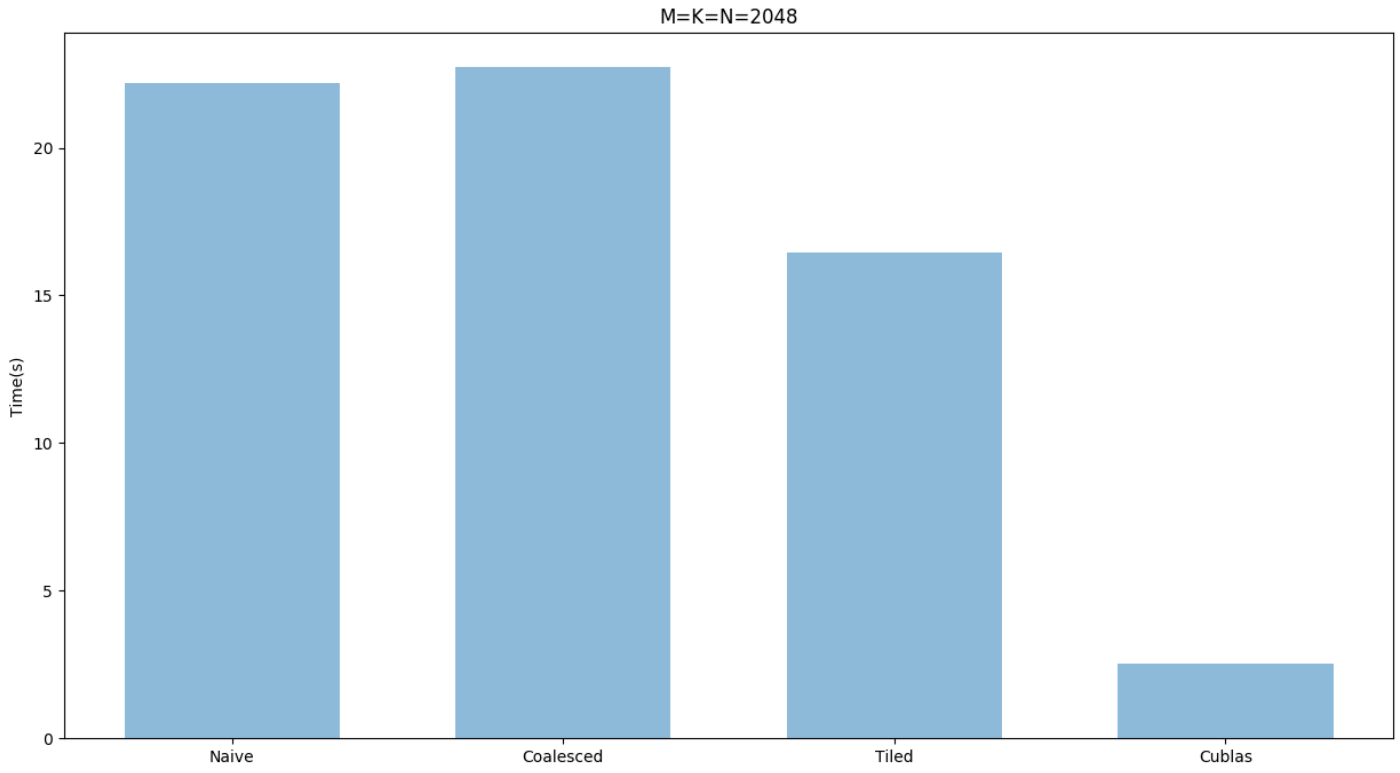
Γραφική παράσταση του performance σε Glops/s !



Από τις παραπάνω γραφικές παραστάσεις παρατηρούμε ότι όλες οι υλοποιήσεις εμφανίζουν παρόμοια συμπεριφορά ως προς τις διάφορες διαστάσεις πινάκων . Δηλαδή , στα σημεία όπου ένας πυρήνας εμφανίζει μείωση της επίδοσης ανάλογη συμπεριφορά εμφανίζουν και οι άλλοι τρεις πυρήνες . Το γεγονός αυτό , είναι απόλυτα λογικά καθώς οι διαστάσεις των πινάκων επηρεάζουν τις διαστάσεις του grid μας .Παράλληλα , παρατηρούμε ότι η Cublas υλοποίηση είναι σταθερά καλύτερη όλων. Το γεγονός αυτό ήταν ως ένα βαθμό αναμενόμενο καθώς η βιβλιοθήκη Cublas είναι highly optimized με βάση το εκάστοτε hardware που χρησιμοποιείται . Αντίθετα , οι δικές μας υλοποιήσεις ακόμα και αν έγιναν προσπάθειες για βελτιστοποίηση των επιδόσεων δεν μπορούν να φτάσουν σε τόσο σύντομο χρονικό διάστημα της βιβλιοθήκης . Μάλιστα , δεν είναι τυχαίο ότι είναι highly recommended στην προγραμματιστική κοινότητα να χρησιμοποιεί την βιβλιοθήκη αυτή και ότι στην πραγματικότητα η πλειοψηφία την χρησιμοποιεί για τον αλγόριθμο DMM . Στην συνέχεια παρατηρούμε ότι η tiled έκδοση πυρήνα είναι

σταθερά καλύτερη των coalesced και naïve εκδόσεων . Το γεγονός αυτό είναι επίσης αναμενόμενο καθώς η tiled υλοποίηση είναι αρκετά πιο sophisticated από την naïve . Επιπρόσθετα , η tiled υλοποίηση όχι μόνο επιτυγχάνει συνένωση των προσβάσεων στην μνήμη αλλά και επιτυγχάνει πολύ καλύτερο locality με το prefect των πινάκων στην τοπική μνήμη. Τέλος , παρατηρούμε ότι η coalesced υλοποίηση σε κάποιες περιπτώσεις εμφανίζει καλύτερη και σε άλλες χειρότερη επίδοση από την naïve υλοποίηση . Η συμπεριφορά αυτή όπως αναφέραμε και παραπάνω είναι συνάρτηση δύο παραγόντων . Πρώτων στο ότι το to # threads πρέπει να είναι πολλαπλάσιο του 32 και δεύτερον στο ότι η κάρτα γραφικών του εργαστηρίου είναι αρκετά παλιά και δεν επιτυγχάνει βέλτιστο optimization.

Στην συνέχεια , για καλύτερη σύγκριση και εποπτεία των διαφορετικών υλοποιήσεων θα παραθέσουμε και ένα bar plot για M=N=K=2048 (μέγιστες διαστάσεις πινάκων για τις μετρήσεις μας) .



Φυσικά το παραπάνω bar plot απλά κάνει ακόμα πιο εμφανής αυτά που σχολιάσαμε παραπάνω !

Κώδικες :

dmm_main.cu

```
/*
 *  dmm_main.cu -- DMM front-end program.
 *
 *  Copyright (C) 2019, Computing Systems Laboratory (CSLab)
 *  Copyright (C) 2019, Athena Elafrou
 */

#include "alloc.h"
#include "dmm.h"
#include "error.h"
#include "gpu_util.h"
#include "mat_util.h"
#include "timer.h"
#include <cuda.h>
#include <stdio.h>
#include <stdlib.h>
#include <cublas_v2.h>

#ifndef VALUES_MAX
#define VALUES_MAX MAKE_VALUE_CONSTANT(1.)
#endif

#ifndef EPS
#define EPS MAKE_VALUE_CONSTANT(1.e-6)
#endif

#ifndef NR_ITER
#define NR_ITER 100
#endif


static void check_result(const value_t *const *test, const value_t *const
*orig,
                         const size_t M, const size_t N) {
  printf("Checking ... ");
  bool ret = mat_equals(test, orig, M, N, EPS);
  if (ret) {
    printf("PASSED!\n");
  } else {
    printf("FAILED!\n");
  }
}

static void report_results(xtimer_t *timer, const size_t M, const size_t N,
                           const size_t K) {
  double elapsed_time = timer_elapsed_time(timer);
  size_t flops = 2 * M * N * K * NR_ITER;

  printf(" Elapsed time: %lf s", elapsed_time);
  printf(" Performance:  %lf Gflop/s\n", flops * 1.e-9 / elapsed_time);
}

static void print_usage() {
```

```
  printf("Usage: [GPU_KERNEL=<kernel_no>] [GPU_BLOCK_SIZE=<size>] "
         "%s <M> <N> <K>\n",
         program_name);
  printf("GPU_KERNEL defaults to 0\n");
  printf("GPU_BLOCK_SIZE defaults to 256\n");
  printf("Available kernels [id:name]:\n");
  size_t i;
  for (i = 0; i < GPU_KERNEL_END; ++i) {
    printf("\t%zd:%s\n", i, gpu_kernels[i].name);
  }
}

int main(int argc, char **argv) {
  set_program_name(argv[0]);
  if (argc < 4) {
    warning(0, "too few arguments");
    print_usage();
    exit(EXIT_FAILURE);
  }

  size_t M = atoi(argv[1]);
  if (!M)
    error(0, "invalid argument: %s", argv[1]);
  size_t N = atoi(argv[2]);
  if (!N)
    error(0, "invalid argument: %s", argv[2]);
  size_t K = atoi(argv[3]);
  if (!K)
    error(0, "invalid argument: %s", argv[3]);

  /* Read block size and kernel to launch from the environment */
  const char *env_gpu_kernel = getenv("GPU_KERNEL");
  const char *env_gpu_block_sizex = getenv("GPU_BLOCK_SIZEX");
  const char *env_gpu_block_sizey = getenv("GPU_BLOCK_SIZEY");
  int kernel = (env_gpu_kernel) ? atoi(env_gpu_kernel) : GPU_NAIVE;
  int block_sizex = (env_gpu_block_sizex) ? atoi(env_gpu_block_sizex) : 32;
  int block_sizey = (env_gpu_block_sizey) ? atoi(env_gpu_block_sizey) : 32;
  size_t orig_M = M;
  size_t orig_N = N;
  size_t orig_K = K;

  if((block_sizey*block_sizex) > 1024) block_sizey = 1024/block_sizex;

  int grid_sizex = ceil((N*1.0)/block_sizex); // FILLME: compute the grid
size
  int grid_sizey = ceil((M*1.0)/block_sizey);
  size_t shmem_size = 0;

  // create the cublas handle
  cublasHandle_t handle;
  cublasCreate(&handle);
  float alpha = 1.0;
  float beta = 0.0;
  cublasStatus_t t;

  /*
   *  FILLME: you can optionally adjust appropriately (increase
   *          only) the matrix size here if that helps you with your
   *          kernel code, e.g., to avoid divergent warps.
```

```
   */


  printf("M: %zd", orig_M);
 // printf("Adjusted dimension M: %zd\n", M);
  printf(" N: %zd", orig_N);
 // printf("Adjusted dimension N: %zd\n", N);
  printf(" K: %zd", orig_K);
 // printf("Adjusted dimension K: %zd\n", K);

  /*
   * Allocate the structures.
   *
   * Initialization to zero is crucial if you adjusted the matrix
   * size.
   */

  value_t **A = (value_t **)calloc_2d(M, K, sizeof(**A));
  if (!A)
    error(1, "alloc_2d failed");

  value_t **A_t = (value_t **)calloc_2d(K, M, sizeof(**A_t));
  if(!A_t)
    error(1,"alloc_2d_failed");

  value_t **B = (value_t **)calloc_2d(K, N, sizeof(**B));
  if (!B)
    error(1, "alloc_2d failed");

  value_t **C = (value_t **)calloc_2d(M, N, sizeof(**C));
  if (!C)
    error(1, "alloc_2d failed");

#ifdef _CHECK_
  value_t **C_serial = (value_t **)calloc_2d(M, N, sizeof(**C_serial));
  if (!C_serial)
    error(1, "alloc_2d failed");
#endif

  /* Initialize */
  srand48(0);
  mat_init_rand(A, orig_M, orig_K, VALUES_MAX);
  mat_init_rand(B, orig_K, orig_N, VALUES_MAX);

  /* Setup timers */
  xtimer_t timer;

  /*
   *  FILLME: Set up the blocks, grid and shared memory depending on
   *          the kernel. Make any transformations to the input
   *          matrix here.
   */

  /*
    In order to achieve memory coalesce we will use
    the transposed array of A.
    The above is meant only for kernels 1,2,3 !
    For kernel 0 (NAIVE) we will use the initial array A.
    We will deep copy array A into A_t if needed in order
```

```
    to maintain initial A for results verification .

  */

  /* GPU allocate A */
  value_t *gpu_A = (value_t *)gpu_alloc(M * K * sizeof(*gpu_A));
  if (!gpu_A)
    error(0, "gpu_alloc failed: %s", gpu_get_last_errmsg());

  if (kernel == 1 ) {
    //Coalesced

    //Deep copy of array A into A_t (also transpose it)
    for(int i=0 ; i<K; i++)
       for(int j=0; j<M; j++)
          A_t[i][j] = A[j][i];

    // Copy A to GPU
    if (copy_to_gpu(A_t[0], gpu_A, M * K * sizeof(*gpu_A)) < 0)
            error(0, "copy_to_gpu failed: %s", gpu_get_last_errmsg());

  }

  else {
    //Naive or CUBLAS or tiled

    // Copy A to GPU
    if (copy_to_gpu(A[0], gpu_A, M * K * sizeof(*gpu_A)) < 0)
            error(0, "copy_to_gpu failed: %s", gpu_get_last_errmsg());

  }

  if( kernel == 2 ) {

    // tiled matrix multiplication

    shmem_size = 2*block_sizey*block_sizex*sizeof(value_t);
  }


  dim3 gpu_block(block_sizex,block_sizey);  // FILLME: set up the block
dimensions
  dim3 gpu_grid(grid_sizex,grid_sizey);   // FILLME: set up the grid
dimensions

  //printf(">>>> Begin of record <<<<\n");
  printf(" Block size: %dx%d", gpu_block.x, gpu_block.y);
  printf(" Grid size : %dx%d", gpu_grid.x, gpu_grid.y);
  printf(" Shared memory size: %ld bytes", shmem_size);

  /* GPU allocations */
  value_t *gpu_B = (value_t *)gpu_alloc(K * N * sizeof(*gpu_B));
  if (!gpu_B)
    error(0, "gpu_alloc failed: %s", gpu_get_last_errmsg());

  value_t *gpu_C = (value_t *)gpu_alloc(M * N * sizeof(*gpu_C));
  if (!gpu_C)
    error(0, "gpu_alloc failed: %s", gpu_get_last_errmsg());
```

```
  /* Copy data to GPU */
  if (copy_to_gpu(B[0], gpu_B, K * N * sizeof(*gpu_B)) < 0)
    error(0, "copy_to_gpu failed: %s", gpu_get_last_errmsg());

  /* Reset C and copy it to GPU */
  mat_init(C, M, N, MAKE_VALUE_CONSTANT(0.0));
  if (copy_to_gpu(C[0], gpu_C, M * N * sizeof(*gpu_C)) < 0)
    error(0, "copy_to_gpu failed: %s", gpu_get_last_errmsg());

  if (kernel >= GPU_KERNEL_END)
    error(0, "the requested kernel does not exist");

  //printf("GPU kernel version: %s\n", gpu_kernels[kernel].name);

  /* Execute and time the kernel */
  timer_clear(&timer);
  timer_start(&timer);
  for (size_t i = 0; i < NR_ITER; ++i) {
    if(kernel==3)
    //t =
cublasSgemm(handle,CUBLAS_OP_N,CUBLAS_OP_N,M,N,K,&alpha,gpu_A,M,gpu_B,K,&be
ta,gpu_C,M);
        t =
cublasSgemm(handle,CUBLAS_OP_N,CUBLAS_OP_N,N,M,K,&alpha,gpu_B,N,gpu_A,K,&be
ta,gpu_C,N);
    else
    gpu_kernels[kernel].fn<<<gpu_grid, gpu_block, shmem_size>>>(gpu_A,
gpu_B,
                                                                gpu_C,
M, N, K);
#ifdef _DEBUG_
    cudaError_t err;
    if ((err = cudaGetLastError()) != cudaSuccess)
      error(0, "gpu kernel failed to launch: %s", gpu_get_errmsg(err));
    if(kernel==3 && t!= CUBLAS_STATUS_SUCCESS) {
          if (t == CUBLAS_STATUS_INVALID_VALUE)
       printf("CUBLAS invalid params\n");
      else if (t == CUBLAS_STATUS_EXECUTION_FAILED)
       printf("CUBLAS execution failed\n");
    }
#endif
    cudaThreadSynchronize();
  }
  timer_stop(&timer);

  /* Copy result back to host and check */
  if (copy_from_gpu(C[0], gpu_C, M * N * sizeof(*gpu_C)) < 0)
    error(0, "copy_from_gpu failed: %s", gpu_get_last_errmsg());

#ifdef _CHECK_
  /* Compute serial */
  dmm_serial(A, B, C_serial, orig_M, orig_N, orig_K);
  check_result(C, C_serial, orig_M, orig_N);
#endif

  report_results(&timer, orig_M, orig_N, orig_K);
  //printf(">>>> End of record <<<<\n");

  /* Free resources on host */
```

```
  free_2d((void **)A);
  free_2d((void **)A_t);
  free_2d((void **)B);
  free_2d((void **)C);
#ifdef _CHECK_
  free_2d((void **)C_serial);
#endif

  /* Free resources on GPU */
  gpu_free(gpu_A);
  gpu_free(gpu_B);
  gpu_free(gpu_C);

  // Destroy the cublas handle
  cublasDestroy(handle);

  return EXIT_SUCCESS;
}
```

dmm_gpu.cu

```
/*
 *  dmm_gpu.cu -- Template for DMM GPU kernels
 *
 *  Copyright (C) 2019, Computing Systems Laboratory (CSLab)
 *  Copyright (C) 2019, Athena Elafrou
 */

#include "dmm.h"
#include <stdio.h>
#include <cublas_v2.h>

/*
 *  Utility function to get the thread ID within the
 *  global working space.
 */
__device__ int get_global_tid() {
  return (gridDim.x * blockIdx.y + blockIdx.x) * blockDim.x * blockDim.y +
         blockDim.x * threadIdx.y + threadIdx.x;
}

/*
 *  Utility function to get the thread ID within the
 *  local/block working space.
 */
__device__ int get_local_tid() {
  return blockDim.x * threadIdx.y + threadIdx.x;
}

/*
 *  Naive kernel
 */

__global__ void dmm_gpu_naive(const value_t *A, const value_t *B, value_t
*C,
                              const size_t M, const size_t N, const size_t
K) {

   value_t sum = 0;
```

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

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

        C[(row*N)+column] = sum ;
   }
}

/*
 *  Coalesced memory acceses
 */

__global__ void dmm_gpu_coalesced(const value_t *A, const value_t *B,
value_t *C,
                 const size_t M, const size_t N, const size_t K) {

   value_t sum = 0;
   int row = blockIdx.y * blockDim.y +threadIdx.y;
   int column = blockIdx.x * blockDim.x + threadIdx.x;

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


}

/*
 *  Use of shared memory
 */

__global__ void dmm_gpu_shmem(const value_t *A, const value_t *B, value_t
*C,
                              const size_t M, const size_t N, const size_t
K) {
  /*
   * FILLME: fill the code for the shared memory kernel.
   */

   extern __shared__ value_t localA[];
   //__shared__ value_t localA[1024];

   int bx,by,tx,ty,row,column;
   int Tile;
   value_t sum = 0 ;

   Tile = blockDim.x ;
   bx = blockIdx.x , by = blockIdx.y ;
   tx = threadIdx.x , ty = threadIdx.y ;
   row = by*Tile + ty , column = bx*Tile + tx ;
   localA[ty*Tile+tx] = 0.0 , localA[Tile*Tile+ty*Tile+tx] = 0.0 ;

   for (int i = 0 ; i<(((K-1)/(Tile))+1) ; ++i ) {
```

```
    if( row < M && (i*Tile+tx) < K )
       localA[ty*Tile+tx] = A[(i*Tile+tx)+row*K];
    else localA[ty*Tile+tx] = 0.0 ;

    if( column < N && (i*Tile+ty) < K )
       localA[Tile*Tile+ty*Tile+tx] = B[(i*Tile+ty)*N+column];
    else localA[Tile*Tile+ty*Tile+tx] = 0.0 ;

    __syncthreads();

    for(int j = 0 ; j < Tile ; ++j)
       sum +=localA[ty*Tile+j]*localA[Tile*Tile+j*Tile+tx];

    __syncthreads();
   }

   if(row < M && column < N)
    C[row*N+column] = sum ;

}

/*
 Cublas matrix multiplication
 CublasSqemm performs C=aop(A)op(B) + bC
*/

__global__ 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) {
  /*
   * Dummy function just to declare cublas .
   */
}
```

dmm.h

```
/*
 *  dmm.h -- Declarations and definitions related to the DMM
 *           multiplication kernels.
 *
 *  Copyright (C) 2019, Computing Systems Laboratory (CSLab)
 *  Copyright (C) 2019, Athena Elafrou
 */

#ifndef DMM_H
#define DMM_H

#include "common.h"
#include <stdbool.h>
#include <stddef.h>

__BEGIN_C_DECLS

void dmm_serial(const value_t *const *A, const value_t *const *B, value_t
**C,
                const size_t M, const size_t N, const size_t K);

__END_C_DECLS

#ifdef __CUDACC__
#define __MAKE_KERNEL_NAME(name) dmm_gpu##name
#define MAKE_KERNEL_NAME(name) __MAKE_KERNEL_NAME(name)

#define DECLARE_GPU_KERNEL(name)
\
  __global__ void MAKE_KERNEL_NAME(name)(const value_t *A, const value_t
*B,   \
                                         value_t *C, const size_t M,
\
                                         const size_t N, const size_t K)

#define SHMEM_PER_BLOCK 8 * 1024

typedef void (*dmm_kernel_t)(const value_t *A, const value_t *B, value_t
*C,
                             const size_t M, const size_t N, const size_t
K);

typedef struct {
  const char *name;
  dmm_kernel_t fn;
} gpu_kernel_t;

enum { GPU_NAIVE = 0, GPU_COALESCED, GPU_SHMEM, GPU_CUBLAS, GPU_KERNEL_END
};

DECLARE_GPU_KERNEL(_naive);
DECLARE_GPU_KERNEL(_coalesced);
DECLARE_GPU_KERNEL(_shmem);
DECLARE_GPU_KERNEL(_cublas);
```

```
static gpu_kernel_t gpu_kernels[] = {
    {
        .name = "naive", .fn = MAKE_KERNEL_NAME(_naive),
    },

    {
        .name = "coalesced", .fn = MAKE_KERNEL_NAME(_coalesced),
    },

    {
        .name = "shmem", .fn = MAKE_KERNEL_NAME(_shmem),
    },

    {
        .name = "cublas", .fn = MAKE_KERNEL_NAME(_cublas),
    },

};

#endif /* __CUDACC__ */
#endif /* DMM_H */
```