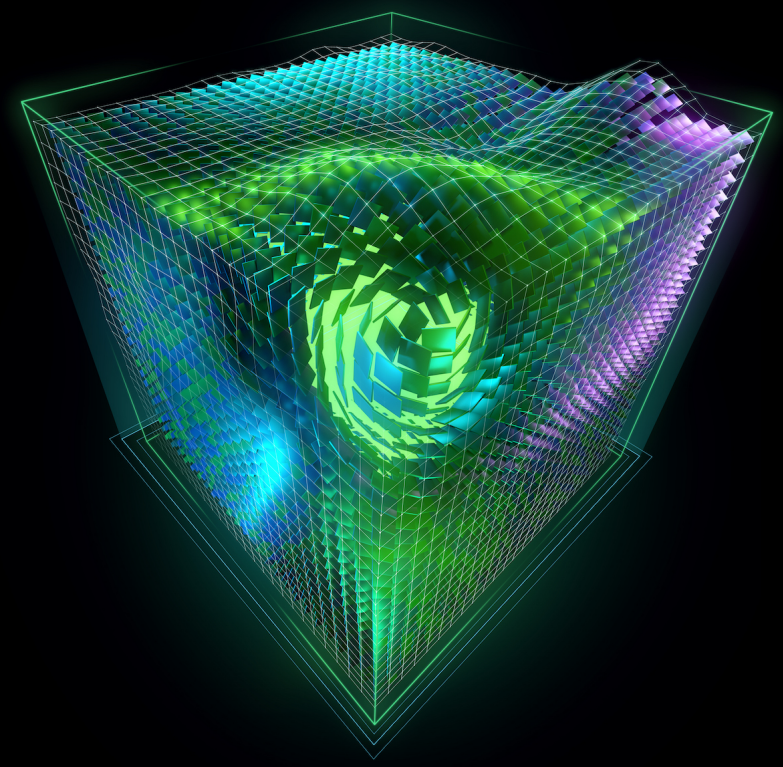
ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
UNIVERSITY OF PATRAS

Πολυτεχνικη Σχολη
Τμημα Μηχανικων Ηλεκτρονικων Υπολογιστων & Πληροφορικης

Λογισμικό και Προγραμματισμός Συστημάτων Υψηλής Επίδοσης

Εργασία Χειμερινού Εξαμήνου 2018-2019



Πέτρος Αγγελής, 6268
pangelis@ceid.upatras.gr

Δαμιανός Ντούμη - Σιγάλας, 6157
nsigalas@ceid.upatras.gr

Ανδρέας Τσαλίδης, 6239
tsalidis@ceid.upatras.gr

Περιεχόμενα

0. Τεχνικά Χαρακτηριστικά

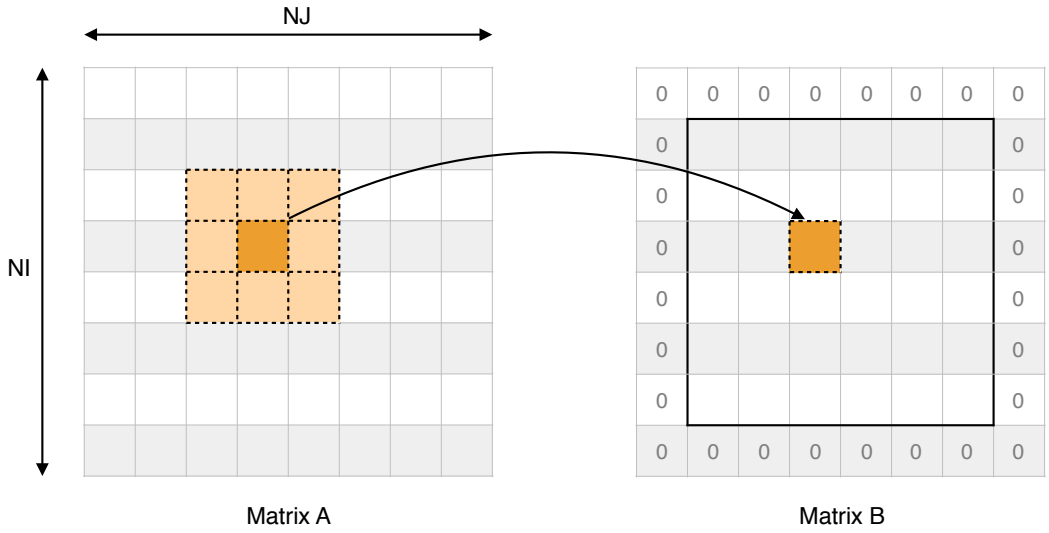
Η εκτέλεση των προγραμμάτων μας έγινε χρησιμοποιώντας τους υπολογιστικούς πόρους που μας διετέθησαν στα πλαίσια του μαθήματος. Συγκεκριμένα η κάρτα γραφικών που αξιοποιήσαμε περιγράφεται από τα παρακάτω τεχνικά χαρακτηριστικά, με **Compute Capability 2.0**. Επιπρόσθετα η CPU που είχαμε στη διάθεση μας ήταν ενας Intel Xeon E5530 @ 2.40GHz με 8 πυρήνες και 8KB L3 cache με 12GB διαθέσιμης μνήμης RAM. Στα πλαίσια της εργασίας δεν αξιοποιείται κάποια δυνατότητα παραλληλίας στη CPU.

```
Device 0: "Tesla C2075"
  CUDA Driver Version / Runtime Version          6.0 / 6.0
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 5375 MBytes (5636554752 bytes)
  (14) Multiprocessors, ( 32) CUDA Cores/MP:     448 CUDA Cores
  GPU Clock rate:                                1147 MHz (1.15 GHz)
  Memory Clock rate:                             1566 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65535),
                                                 3D=(2048, 2048, 2048)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           66 / 0
```

Για την χρονομέτρηση των υπολογιστικών πυρήνων (kernels) χρησιμοποιούμε τα κατάλληλα cuda events που μας παρέχονται από τις βιβλιοθήκες της CUDA. Αν χρησιμοποιούσαμε κάποια λύση που βασίζεται σε βιβλιοθήκες που μας προσφέρει ο host, χωρίς την χρήση κάποιου επιπλέον ρητού συγχρονισμού θα μετράγαμε τον χρόνο εκκίνησης του υπολογιστικού πυρήνα και όχι τον χρόνο εκτέλεσής του, καθώς η εκτέλεση του kernel στην GPU είναι ασύγχρονη ως προς την υπόλοιπη ροή του προγράμματος στη CPU [1]. Τέλος, τον παραδοτέο κώδικα συνοδεύουν κάποια απλουστευμένα Makefiles, ανεξάρτητα για κάθε ερώτημα, για την πιο εύκολη μεταγλώττισή του.

1. 2D Convolution

Στο πρώτο ερώτημα της εργασίας καλούμαστε να υλοποιήσουμε στη GPU την 2D συνέλιξη ενός μητρώου A. Για τον υπολογισμό της νέας τιμής στη θέση (i,j) χρησιμοποιείται η τιμή της τρέχουσας θέσης καθώς και οι τιμές των οχτώ γειτονικών της, κάθε μία εκ των οποίων συνεισφέρει με ένα συγκεκριμένο βάρος στο τελικό αποτέλεσμα. Στο μητρώο B, το οποίο αποθηκεύει τις υπολογισθείσες τιμές, για τις περιφερειακές (οριακές) θέσεις του πίνακα δεν γίνεται κάποιος υπολογισμός με αποτέλεσμα να συμπληρώνονται με μηδενικά.



Πως όμως εξασφαλίζουμε την ορθότητα των υπολογισμών στη GPU;

Ενα ερώτημα που προκύπτει μετά την παραλληλοποίηση του κώδικα είναι αν έχει γίνει με σωστό τρόπο. Το γεγονός ότι εκτελείται χωρίς σφάλματα δεν μας εγγυάται την ορθότητα των υπολογισμών. Για την εξακρίβωσή της απαιτείται να καταστρώσουμε μια τεχνική ελέγχου η οποία θα μας διασφαλίζει ότι τελικά το μητρώο που υπολογίστηκε είναι το ζητούμενο. Σε μικρού μεγέθους προβλήματα ο οπτικός έλεγχος είναι εφικτός αφού μπορούμε να εκτυπώσουμε στην οθόνη τα αποτελέσματα και να τα αντιπαραβάλουμε με αυτά που προκύπτουν από τον αρχικό κώδικα. Με αυτόν τον τρόπο είναι πιο εύκολο να έχουμε μια πρώτη εικόνα του υπολογισμού μας η οποία μας επιτρέπει να ελέγξουμε με ευκολία τις οριακές περιπτώσεις, όπως τι συμβαίνει στα άκρα του μητρώου.

Για μεγαλύτερου μεγέθους προβλήματα αποφασίσαμε να ακολουθήσουμε μια τεχνική παρόμοια με αυτήν που παρουσιάστηκε στο μάθημα της Παράλληλης Επεξεργασίας κατά το προηγούμενο ακαδημαϊκό έτος. Συγκεκριμένα διατρέχουμε το μητρώο και αποθηκεύουμε είτε δειγματοληπτικά είτε το σύνολο των τιμών (όταν το μέγεθός του μας το επιτρέπει) σε ένα αρχείο. Κάνουμε το ίδιο και για τον αρχικό κώδικα και έπειτα τα συγκρίνουμε τα δύο αρχεία με χρήση της `numdiff`. Θεωρούμε ότι οι υπολογισμοί μας είναι ορθοί όταν οι τιμές που περιέχουν είναι ίσες (λαμβάνοντας υπόψιν ένα άνω όριο απόλυτου σφάλματος).

Επιλογή μεγέθους block και grid

Το επόμενο θέμα που μας απασχόλησε είναι ποια είναι τα βέλτιστα μεγέθη block και grid για τα οποία παίρνουμε την καλύτερη απόδοση; Αφού το πρόβλημα μας περιέχει μητρώα δύο διαστάσεων, θα χρησιμοποιήσουμε δισδιάστατα block και grid. Τι μέγεθος όμως θα ορίσουμε σε κάθε μία από τις διαστάσεις για καθένα από αυτά τα δύο στοιχεία; Κάποια στοιχεία που πρέπει να ληφθούν υπόψιν περιγράφονται παρακάτω:

- Το μέγεθος κάθε block δεν πρέπει να ξεπερνά τα 1024 threads με μέγιστες διαστάσεις (1024, 1024, 64).
- Κάθε block δεν μπορεί να καταναλώνει περισσότερους από 32k καταχωρητές.

- Κάθε block δεν μπορεί να καταναλώνει περισσότερο από 48kb shared memory.
- Ιδανικά, ο αριθμός των thread ανα block πρέπει να είναι πολλαπλάσιο του warp size (32 threads). Η Tesla C2075 έχει 1.536 thread slots per SM. Αφού 1.536 = 32 × 48, έχουμε #thread slots = warp size × #warps per block. Αν έχω 32 threads ανα block, 1.536 / 32 = 48 blocks / SM το μέγιστο.
- Κάθε SM της GPU ιδανικά πρέπει να έχει αρκετά ενεργά warps ώστε να προσεγγίζουμε την μέγιστη διεκπεραιωτική ικανότητα (throughput).

Γίνεται φανερό ότι δεν υπάρχει ένας ντετερμινιστικός τρόπος να καθορίσουμε το ιδανικό μέγεθος block και grid αφού αυτά εξαρτώνται τόσο από περιορισμούς στο hardware που θέτει η εκάστοτε κάρτα όσο και στην φύση του κώδικα μας. Για τον λόγο αυτό έχοντας υπόψιν τις παραπάνω παραμέτρους πειραματιζόμαστε δοκιμάζοντας διαφορετικούς συνδυασμούς από τον "χώρο αναζήτησης" μας μέχρι να βρούμε για ποια μεγέθη το πρόγραμμα μας συμπεριφέρεται με ικανοποιητικό τρόπο. Παρακάτω παρατίθενται κάποιες ενδεικτικές μετρήσεις από την διαδικασία αυτή οι οποίες επιβεβαιώνουν την διαίσθηση μας.

Αποτελέσματα

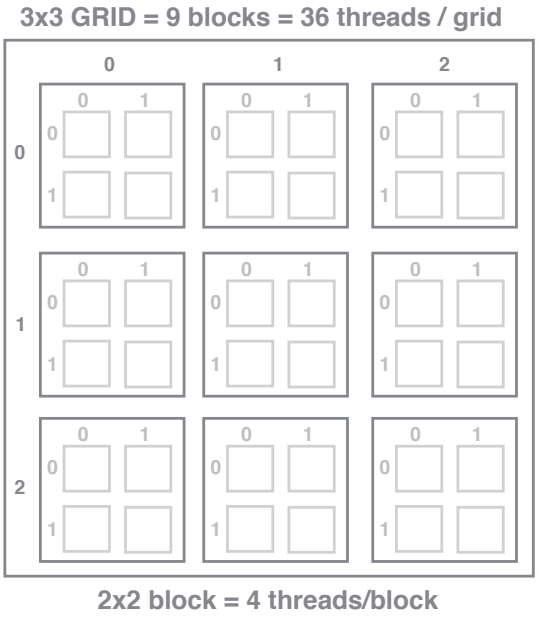
NI x NJ	**4096 x 4096**				**8192 x 8192**							
block (x,y)		32	32			32	32		16	16	8	8
grid (x,y)		128	128			256	256		512	512	1024	1024
	CPU	**GPU**			**CPU**	**GPU**			**GPU**		**GPU**	
	0,377982	0,005917			1,511	0,03295			0,02326		0,033457	
	0,378036	0,005935			1,511	0,03295			0,02327		0,033463	
	0,384716	0,005922			1,536	0,03296			0,02333		0,033477	
	0,37807	0,005911			1,512	0,03293			0,02327		0,033438	
	0,378022	0,005943			1,511	0,03297			0,02329		0,033474	
average (sec)	**0,3793652**	**0,0059256**			**1,516**	**0,03295**			**0,02329**		**0,0334618**	
speedup		64,0				46,0			**65,1**		45,3	

NI x NJ	**6000x8000**								
block (x,y)		16	16		10	10		16	16
grid (x,y)		512	1024		800	600		500	375
	CPU	**GPU**			**GPU**			**GPU**	
	1,029803	0,016912			0,025204			0,014521	
	1,029727	0,016828			0,025208			0,014549	
	1,030155	0,016834			0,025159			0,014541	
	1,030539	0,016884			0,025179			0,014543	
	1,030419	0,016902			0,025238			0,014557	
average (sec)	**1,0301286**	**0,016872**			**0,0251976**			**0,0145422**	
speedup		61,1			40,9			70,8	

Ιδιαίτερο ενδιαφέρον παρουσιάζει η περίπτωση στην οποία το μέγεθος του μητρώου είναι 6000X8000, ένα grid αρκετά μεγαλύτερο από το μητρώο δίνει σημαντικά καλύτερη απόδοση από ένα άλλο grid με διαστάσεις που ενώ ταιριάζουν ακριβώς στο αρχικό μητρώο, δεν έχουν μέγεθος block πολλαπλάσιο του warp size. Σε κάθε περίπτωση καλύτερη απόδοση επιτυγχάνουμε όταν για 16×16 block χρησιμοποιούμε το μικρότερο δυνατό grid. Αφού λοιπόν καταλήξαμε ότι στις περισσότερες περιπτώσεις block μεγέθους 16×16 μας δίνουν την καλύτερη συμπεριφορά μπορούμε να εκφράσουμε αλγοριθμικά τον υπολογισμό του απαιτούμενου μεγέθους για το grid ανάλογα με το μέγεθος της εισόδου (NI, NJ) ως εξής:

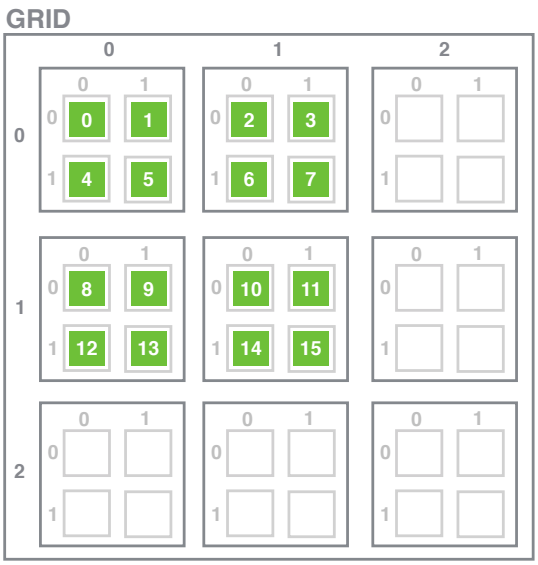
```
// given block size
int block_dim_x = 16;
int block_dim_y = 16;

// calculate grid dimensions
int grid_dim_x = ceil( (float) NJ / block_dim_x );
int grid_dim_y = ceil( (float) NI / block_dim_y );
```



4x4 Matrix

	0	1	2	3
0	0,0	0,1	0,2	0,3
1	1,0	1,1	1,2	1,3
2	2,0	2,1	2,2	2,3
3	3,0	3,1	3,2	3,3



Στο παραπάνω σχήμα φαίνεται ο τρόπος με τον οποίο αντιστοιχούμε ένα 4x4 μητρώο σε δεδομένων διαστάσεων grid, μέσω της μονοδιάστατης αναπαράστασής του στην μνήμη. Στην πραγματικότητα βέβαια τα πράγματα είναι διαφορετικά αφού δεν χρησιμοποιούμε προκαθορισμένες διαστάσεις για το grid αλλά ο αριθμός των block καθορίζεται δυναμικά ώστε να είναι ο ελάχιστος απαραίτητος σε κάθε διάσταση.

2. $y = A^T \cdot A \cdot x$

Στο ζητούμενο αυτό καλούμαστε να υλοποιήσουμε στη GPU την παρακάτω πράξη:

$$y = A^T \cdot A \cdot x$$

όπου:
A: μητρώο διαστάσεων NX × NY,
x: διάνυσμα (διαστάσεων NY × 1),
$\mathbf{A^T}$: ανάστροφο του A (NY × NX) και
y: το ζητούμενο διάνυσμα διαστάσεων NY × 1

Στην δοθείσα σειριακή εκδοχή παρατηρούμε ότι ο υπολογισμός του αποτελέσματος γίνεται σταδιακά σε έναν βρόγχο for όπου σε κάθε βήμα υπολογίζεται το εσωτερικό/στικτό γινόμενο (dot product) της τρέχουσας γραμμής που βρίσκεται ο υπολογισμός και στην συνέχεια με βάση αυτό ενημερώνεται το διάνυσμα y, κάνοντας ουσιαστικά την αντίστοιχη διαδικασία.

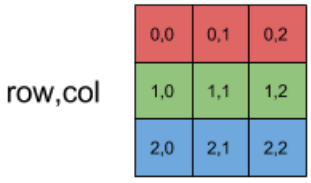
Για να μεταφέρουμε την λειτουργία αυτή σε CUDA αξιοποιώντας λειτουργίες όπως η shared memory, προχωρήσαμε στην αναδιοργάνωση των υπολογισμών ως εξής:

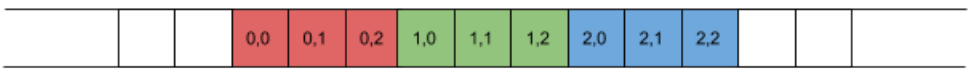
$$\left(A^T \cdot (A \cdot x)\right) = y$$

Αρχικά προκειμένου να γλυτώσουμε υπολογισμούς πρώτη γίνεται η πράξη $(A \cdot x)$ από την οποία προκύπτει ένα διάνυσμα-στήλη μεγέθους NX στοιχείων. Επειτα πολλαπλασιάζεται ο ανάστροφος A^T με το διάνυσμα που υπολογίστηκε στο προηγούμενο βήμα. Ετσι προκύπτει το ζητούμενο διάνυσμα y μεγέθους NY στοιχείων. Στο σημείο αυτό τίθενται δύο κύριοι προβληματισμοί μας σχετικά με το πώς θα αντιμετωπίσουμε την διαδικασία αυτή. Ο πρώτος είναι ότι επειδή θα έχουμε ήδη μεταφέρει το μητρώο A στην μνήμη της GPU θα πρέπει να το εκμεταλλευτούμε τόσο ως το κανονικό όσο ως και το ανάστροφό του. Ακόμα και αν κάναμε την αναστροφή του στην GPU πέρα από σπατάλη χρόνου θα είχαμε και σπατάλη χώρου, ο οποίος όσο μεγαλώνει το μέγεθος της εισόδου γίνεται όλο και πιο περιορισμένος. Το δεύτερο σημείο που μας απασχόλησε αφορά ποιες δυνατότητες και με ποιον τρόπο θα τις χρησιμοποιήσουμε ώστε να αξιοποιήσουμε τα πλεονεκτήματα του παράλληλου υπολογισμού που μας προσφέρει η κάρτα γραφικών.

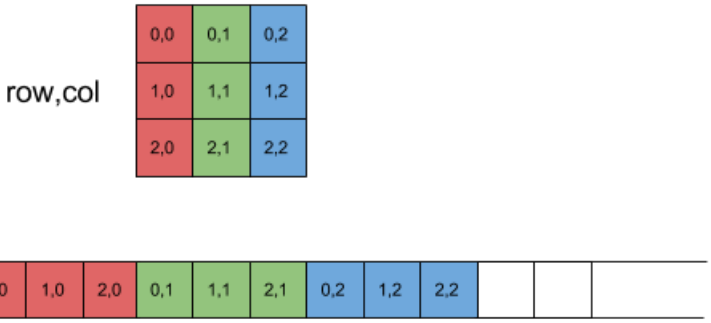
Μητρώο A και ανάστροφο $\mathbf{A^T}$

Για τον υπολογισμό του αποτελέσματος χρειάζεται να διατρέξουμε τόσο το μητρώο A όσο και το ανάστροφό του. Ο τρόπος με τον οποίο αρχικοποιούμε το A αντιστοιχεί στην κατά-γραμμές (row-major) αναπαράσταση του στην μνήμη όπου διαδοχικές γραμμές αποθηκεύονται σε διαδοχικές θέσεις μνήμης όπως φαίνεται και στο σχήμα παρακάτω.





Αντίθετα μία κατά στήλες (column-major) αναπαράσταση του μητρώου στην μνήμη θα ήταν η αντίστοιχη:



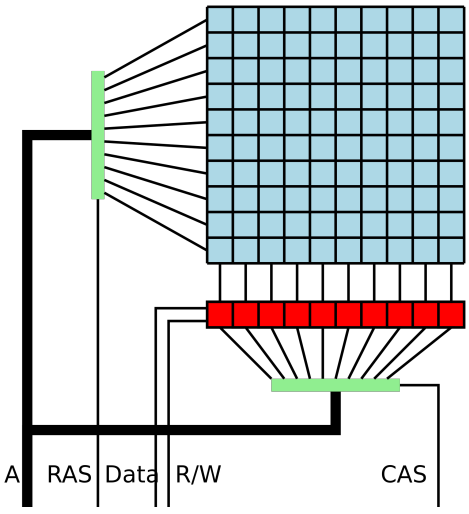
Σε κάθε περίπτωση εμείς στην υλοποίησή μας διαθέτουμε τα δεδομένα μας στην εξής μορφή:

$$\text{Matrix A} = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix} \qquad \text{Matrix A}^{\mathsf{T}} = \begin{pmatrix} 0 & 3 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \end{pmatrix}$$

```
A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8}
```

Ετσι λοιπόν προκειμένου να αποφύγουμε την διπλή αποθήκευση του μητρώου στην μνήμη, εκμεταλλευόμαστε την παρουσία του A ώστε διατρέχοντας το με τον κατάλληλο τρόπο να προσπευλαύνουμε το ανάστροφό του. Η πρακτική αυτή συνδέεται στενά με την έννοια του coalescing που περιγράφεται παρακάτω.

Memory Coalescing / Row-Major & Column-Major Traversal



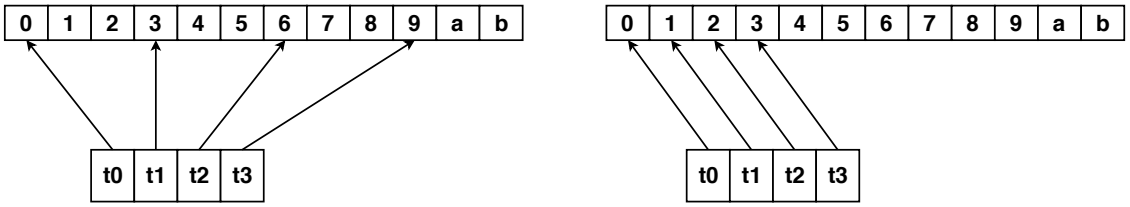
Τυπική αρχιτεκτονική DRAM bank

Εξ αιτίας της κατασκευής των DRAM, όπου ο τρισδιάστατος, οργανωμένος σε banks φυσικός χώρος διευθύνσεων, αντιστοιχίζεται σε έναν γραμμικό λογικό χώρο, όταν προσπελαύνεται μία θέση μνήμης στην πραγματικότητα γίνονται διαθέσιμα τα δεδομένα περισσότερων από μία διαδοχικών διευθύνσεων. Λαμβάνοντας υπόψιν ότι όλα τα thread σε ένα warp εκτελούν την ίδια εντολή, αν οργανώσουμε τον κώδικα μας με τέτοιο τρόπο ώστε τα threads κάθε χρονική στιγμή να προσπελαύνουν συνεχόμενες θέσεις μνήμης τότε εκμεταλλευόμενοι την παραπάνω συμπεριφορά μπορούμε να πετύχουμε σημαντική βελτίωση της επίδοσης του προγράμματός μας. Η τεχνική αυτή ονομάζεται memory coalescing [2]. Ας εξετάσουμε το παρακάτω παράδειγμα:

```
2D array:                    1D mapping in memory:
0  1  2  3                   0 1 2 3 4 5 6 7 8 9 a b
4  5  6  7                   element (r,c) maps to (r*4+c)
8  9  a  b

row-major traversal:                    column-major traversal:
thread 0:  0, 1, 2                      thread 0:  0, 4, 8
thread 1:  3, 4, 5                      thread 1:  1, 5, 9
thread 2:  6, 7, 8                      thread 2:  2, 6, a
thread 3:  9, a, b                      thread 3:  3, 7, b
----------time---------->               ----------time---------->
```

Θεωρούμε ότι έχουμε ένα 3X4 μητρώο αποθηκευμένο στην μνήμη σειριακά, με οργάνωση κατά γραμμή. Αν επίσης διαθέτουμε 4 thread καθένα εκ των οποίο σε κάθε βήμα προσπελαύνει μία θέση του πίνακα, τότε κάθε thread θα αναλάβει συνολικά την επεξεργασία τριών στοιχείων. Αν λοιπόν σε κάθε thread αντιστοιχήσουμε τα στοιχεία που του αναλογούν με έναν row-major τρόπο τότε κάθε χρονική στιγμή θα καταλήξουν να μην προσπελαύνουν διαδοχικές θέσεις μνήμης. Αντίθετα αν κάθε thread κινείται στα στοιχεία μιας στήλης (column-major traversal) τότε θα πετύχουμε το επιθυμητό αποτέλεσμα. Η συμπεριφορά αυτή γίνεται πιο εύκολα κατανοητή παρατηρώντας το παρακάτω σχήμα στο οποίο αριστερα απεικονίζονται οι θέσεις μνήμης που προσπελαύνουν τα 4 threads, στο πρώτο βήμα του υπολογισμού, αν χρησιμοποιείται row-major traversal ενω δεξιά η αντίστοιχη συμπεριφορά για column-major traversal.



Η αξιοποίηση αυτής της τεχνικής βρίσκει άμεση εφαρμογή στο πρόβλημά μας κατά τον πολλαπλασιασμό του μητρώου A^T με το διάνυσμα (Ax). Εξαιτίας του τρόπου με τον οποίο έχουμε αποθηκευμένο το μητρώο στην μνήμη, διατρέχοντας τα στοιχεία του A υποθέτοντας οργάνωση κατά στήλες ουσιαστικά επιτυγχάνουμε την προσπέλαση του ανάστροφού του με coalesced τρόπο.

Shared Memory

Η προσπέλαση της shared memory είναι αρκετές τάξεις μεγέθους ταχύτερη σε σχέση με αυτήν της global memory επομένως μπορούμε να την αξιοποιήσουμε για να βελτιώσουμε την απόδοση της εφαρμογής μας. Βέβαια με τη χρήση της προκύπτουν δύο ζητήματα που πρέπει να λάβουμε υπόψιν. Αρχικά πρέπει να μεταφέρουμε στην κοινόχρηστη μνήμη μόνο τα δεδομένα που θα χρησιμοποιηθούν παραπάνω από μία φορές (ιδανικά πολλές περισσότερες) καθώς και να τροποποιήσουμε τον αλγόριθμό μας ώστε να τα χρησιμοποιεί σε μία φάση ώστε να αποφύγουμε τις εναλλασσόμενες μεταφορές των ίδιων στοιχείων. Σε περίπτωση που ένα στοιχείο χρησιμοποιείται μόνο μία φορά είναι ανώφελο να το μεταφέρουμε στην shared memory καθώς στην πράξη θα προσθέσουμε επιπλέον επιβάρυνση για τη μεταφορά και τη χρήση του.

Επίσης η shared memory είναι διαθέσιμη σε επίπεδο block. Δηλαδή μόνο τα threads του ίδιου block έχουν πρόσβαση στα κοινά δεδομένα. Το δεύτερο ζήτημα που προκύπτει είναι ότι κάθε block έχει διαθέσιμα 49152 bytes (48KB) κοινόχρηστης μνήμης με αποτέλεσμα να πρέπει να είμαστε προσεκτικοί να μην ξεπεράσουμε το όριο αυτό. Στον δικό μας αλγόριθμο εφαρμόζουμε τη χρήση της shared memory κατά τον πολλαπλασιασμό μητρώου διανύσματος.

Σύνοψη υλοποίησης

Η υλοποίηση μας αποτελείται από δύο υπολογιστικούς πυρήνες (kernels):

1. `__global__ void matvec_kernel_row_major(...)`

   Στον kernel αυτόν υλοποιείται η πράξη του πολλαπλασιασμού μητρώου διανύσματος και χρησιμοποιείται για την εκτέλεση της πρώτης πράξης, δηλαδή της (Ax). Για την προσπέλαση των στοιχείων του διανύσματος x γίνεται χρήση της shared memory καθώς τα στοιχεία του χρησιμοποιούνται με μεγαλύτερη συχνότητα και επαναλαμβανόμενα για τον υπολογισμού του αποτελέσματος. Το μητρώο A βρίσκεται αποθηκευμένο στην global memory και κάθε thread πραγματοποιεί προσπέλαση στοιχείων με row-major τρόπο (υποθέτοντας οργάνωση στη μνήμη κατά γραμμές), με αποτέλεσμα οι προσπελάσεις της global memory να μην ειναι coalesced.

   Για την κλήση του υπολογιστικού πυρήνα έχει δημιουργηθεί μία συνάρτηση-wrapper η οποία αναλαμβάνει να υπολογίσει το απαιτούμενο grid size ανάλογα το μέγεθος του προβλήματος, την κλήση του kernel καθώς και την χρονομέτρησή του. Η συνάρτηση αυτή είναι η: `__host__ float matvec_ROW_MAJOR(...)`

2. `__global__ void matvec_kernel_column_major(...)`

   Στον kernel αυτόν υλοποιείται η πράξη του πολλαπλασιασμού μητρώου διανύσματος και χρησιμοποιείται για την εκτέλεση της πρώτης πράξης, δηλαδή της $A^T(Ax)$ όπου το (Ax) πλέον αποτελεί ένα διάνυσμα ως το αποτέλεσμα της προηγούμενης πράξης. Για την προσπέλαση των στοιχείων του διανύσματος x γίνεται χρήση της shared memory καθώς τα στοιχεία του χρησιμοποιούνται με μεγαλύτερη συχνότητα και επαναλαμβανόμενα για τον υπολογισμού του αποτελέσματος. Το μητρώο A βρίσκεται αποθηκευμένο στην global memory και κάθε thread πραγματοποιεί προσπέλαση στοιχείων με column-major τρόπο, υποθέτωντας οργάνωση στη μνήμη κατά στήλες, ώστε να γίνεται η πράξη με το ανάστροφο του μητρώου A. Αυτό έχει ως αποτέλεσμα οι προσπελάσεις της global memory να ειναι coalesced.

   Αντίστοιχα έχει δημιουργηθεί η συνάρτηση-wrapper: `__host__ float matvec_COL_MAJOR(...)`

cuBLAS - gemv

Προκειμένου να αξιολογήσουμε την απόδοση της δικής μας υλοποίησης αποφασίσαμε να δοκιμάσουμε την εκτέλεση των ίδιων πράξεων με χρήση της υλοποίησης της BLAS που παρέχει η CUDA, δηλαδή την cuBLAS. Συγκεκριμένα με χρήση της level 2 συνάρτησης gemv (generalized matrix vector multiplication) εκτελέσαμε τους πολλαπλασιασμούς μητρώου-διανύσματος όπου απαιτούνται.

Αποτελέσματα

Παρακάτω παρατίθενται οι μέσοι χρόνοι όπως προέκυψαν μετά την εκτέλεση του προγράμματος 10 φορές για κάθε διαφορετικό μέγεθος εισόδου. Το μέγεθος σε MByte που αναφέρεται αφορά μόνο τον χώρο που καταλαμβάνει στη μνήμη το μητρώο A και αποτελεί μία ένδειξη του μεγέθους του προβλήματος.

Problem Size			Execution Time (seconds)		
NX	**NY**	**Mbytes**	**CPU**	**GPU - custom**	**GPU - cuBLAS**
2000	3000	45,8	0,075025	0,003604	0,000625
4096	4096	128,0	0,209489	0,012213	0,001500
16000	16000	1953,1	3,219159	0,161162	0,023177
16000	32000	3906,3	6,760477	0,320441	0,047232
35840	17408	4760,0	8,733589	0,371951	0,056956
32000	32000	7812,5	46,850101	not enough memory - for a single transaction	

Στον πίνακα που ακολουθεί αναλύονται οι χρόνοι εκτέλεσης του υπολογισμού ανα kernel για τα διάφορα μεγέθη εισόδου:

execution time per kernel for one instance						
NX	**NY**	**Kernel 1 - Ax**		**Kernel 2 - AT(Ax)**		**Total**
2000	3000	0,002857	79,3%	0,000746	20,7%	0,003603
4096	4096	0,010577	86,2%	0,001698	13,8%	0,012275
16000	16000	0,131900	81,8%	0,029262	18,2%	0,161162
16000	32000	0,262370	81,9%	0,058071	18,1%	0,320441
35840	17408	0,304047	81,7%	0,067905	18,3%	0,371952
		average	82,2%		17,8%	

Συμπεράσματα

NX	NY	GPU Speedup	cuBLAS Speedup
2000	3000	20,8	120,0
4096	4096	17,2	139,7
16000	16000	20,0	138,9
16000	32000	21,1	143,1
35840	17408	23,5	153,3

Υπολογίζοντας τα μετρούμενα speedup τόσο της υλοποίησής μας όσο και της cuBLAS μπορούμε να οδηγηθούμε σε κάποια συμπεράσματα. Αρχικά ενώ έχουμε καταφέρει να πάρουμε αρκετά καλή βελτίωση της απόδοσης μέσω της δικής μας υλοποίησης γίνεται φανερό ότι χρήση της cuBLAS σε μία πιθανή εφαρμογή θα ήταν επιβεβλημένη, καθώς επιτυγχάνει εξαιρετική επιτάχυνση, “εκτός ανταγωνισμού”.

Μπορούν να γίνουν οι εξής παρατηρήσεις:

1. Οταν οι διαστάσεις του μεγέθους του μητρώου A αποτελούν δυνάμεις του 2 ή πολλαπλάσια τους τότε έχουμε καλύτερα αποτελέσματα, από την άποψη μας ότι ο αλγόριθμος μας είναι πιο αποδοτικός. Ενας λόγος για τον οποίο μπορεί να συμβαίνει αυτό είναι ότι δημιουργούνται ομοιόμορφα grid και blocks στα οποία όλα τα thread αναλαμβάνουν αξιοποιούνται για τον υπολογισμό με αποτέλεσμα το μεγαλύτερο occupancy.
2. Παρατηρούμε ότι ο δεύτερος kernel είναι σημαντικά ταχύτερος καθώς καταλαμβάνει μόνο το 10% του συνολικού χρόνου κατά μέσο όρο, αν και φαινομενικά εκτελεί τον ίδιο υπολογισμό. Αυτό συμβαίνει καθώς πέρα από την χρήση της shared memory για την αποθήκευση του διανύσματος x εκμεταλλεύεται και την τεχνική του coalescing όταν προσπελαύνει τα στοιχεία του A^T από την global memory.

Περαιτέρω Βελτίωση

Είναι φανερό ότι υπάρχουν μεγάλα περιθώρια βελτίωσης της απόδοσης της εφαρμογής μας τα οποία όμως ίσως ξεφεύγουν από τα πλαίσια της παρούσας εργασίας. Θα μπορούσαμε να τα συνοψίσουμε στα εξής σημεία:

- Θα πρέπει να επικεντρωθούμε στην βελτίωση του πρώτου υπολογιστικού πυρήνα ο οποίος προσπελαύνει την global memory με μη-coalesced τρόπο με αποτέλεσμα την σημαντική επιβάρυνση του απαιτούμενου χρόνου. Μια ιδέα είναι να μεταφέρουμε τμήματα του μητρώου A στην shared memory κατα την διάρκεια

του υπολογισμού υλοποιώντας ουσιαστικά έναν **tiled shared memory** αλγόριθμο πολλαπλασιασμού μητρώου διανύσματος. Μία τέτοια προσέγγιση προτείνεται στην εργασία [3], την οποία αν και καταφέραμε και εντάξαμε στην υλοποίηση μας με πολύ καλά αποτελέσματα για τετραγωνικά μητρώα με διαστάσεις δυνάμεις του 2, δεν δούλευε για όλα τα μεγέθη εισόδου. Μια λύση θα ήταν να τροποποιήσουμε τον κώδικά μας ώστε να μετατρέπει το μητρώο εισόδου σε τετραγωνικό με μέγεθος κάθε διάστασης την κοντινότερη δύναμη του 2 για την μεγαλύτερη από τις 2 αρχικές.

- Το δεύτερο σημείο επικεντρώνεται στην παρατήρηση οτι η cuBLAS επιτυγχάνει εξαιρετικά αποτελέσματα. Καθώς πρόκειται για μια closed source βιβλιοθήκη της nVidia δεν έχουμε πρόσβαση στον κώδικα της ώστε να προσπαθήσουμε τον κατανοήσουμε τον τρόπο λειτουργίας της. Για τον λόγο αυτό αναζητήσαμε να βρούμε open source υλοποιήσεις της BLAS για GPU και να δούμε με ποιον τρόπο αντιμετωπίζουν την gemv. Πράγματι εντοπίσαμε την kblas-gpu η οποία υλοποιεί τις level 2 συναρτήσεις της blas. Αν είχαμε περισσότερο χρόνο θα μπορούσαμε να μελετήσουμε και αυτήν την προσέγγιση για να δούμε αν θα πετυχαίναμε περαιτέρω βελτίωση.

3. Covariance

Η ανάλυση των περισσότερων μεθόδων και τεχνικών βελτιστοποίησης που χρησιμοποιήσαμε στα πλαίσια της εργασίας έχει ήδη αναφερθεί στα προηγούμενα ερωτήματα. Στη συνέχεια γίνεται μία σύντομη επισκόπηση του τρόπου με τον οποίο έγινε η παραλληλοποίηση στη GPU του κώδικα του τρίτου ερωτήματος.

Στο συγκεκριμένο ερώτημα υπολογίζεται το μητρώο συνδιακύμανσης ενός αρχικού μητρώου μεγέθους M X N. Ο υπολογισμός αυτός γίνεται σε τρία βήματα:

1. **Mean Calculation:** Για κάθε στήλη του μητρώου A υπολογίζεται ο μέσος όρος των στοιχείων της στήλης.
2. **Matrix Centering:** Από κάθε στοιχείο κάθε στήλης του μητρώου αφαιρεί τον μέσο όρο της αντίστοιχης στήλης που υπολογίστηκε στο προηγούμενο βήμα.
3. **Covariance Matrix Calculation:** Πολλαπλασιάζεται το μητρώο που υπολογίστηκε στο προηγούμενο βήμα με το ανάστροφό του. Επειδή το αποτέλεσμα που προκύπτει είναι συμμετρικό μητρώο, αρκεί ο υπολογισμός είτε του άνω είτε του κάτω τριγωνικού μέρους του.

CUDA-naïve implementation

Αρχικά προχωρήσαμε στην απλούστερη δυνατή μορφή παραλληλοποίησης του κώδικα που μας δίνεται, δημιουργώντας έναν υπολογιστικό πυρήνα για κάθε ένα από τα βήματα του υπολογισμού όπως αυτά αναφέρθηκαν παραπάνω. Επίσης δημιουργήθηκε και μία συνάρτηση wrapper η οποία αναλαμβάνει να υπολογίσει τις κατάλληλες παραμέτρους και να καλέσει κάθε kernel.

1. `__global__ void mean_kernel(double *mean_d, double *data_d)`
2. `__global__ void reduce_kernel(double *mean_d, double *data_d)`
3. `__global__ void covar_kernel(double *symmat_d, double *data_d)`
4. `__host__ void calculate_on_GPU(...) // wrapper function`

Επόμενο βήμα είναι ο έλεγχος της ορθότητας των υπολογισμών ώστε να διαπιστώσουμε ότι το πρόγραμμα μας παράγει τα επιθυμητά αποτελέσματα. Για τον λόγο αυτό πέρα από την εγγραφή των αποτελεσμάτων στα αρχεία cpu.out και gpu.out έχει δημιουργηθεί η συνάρτηση *compareResults()* η οποία συγκρίνει μεταξύ τους τα στοιχεία που παράχθηκαν στην CPU και την GPU και μετρά πόσα από αυτά διαφέρουν πάνω από ένα προκαθορισμένο ποσοστό σφάλματος. Η ιδέα αυτή προέρχεται από τον τρόπο με τον οποίο γίνεται η σύγκριση των αποτελεσμάτων στην υλοποίηση των PolyBench benchmarks για GPU όπως αυτά φαίνονται εδώ και στην εργασία [4].

Βελτιστοποίηση

Αφού επιβεβαιώσαμε την ορθότητα των υπολογισμών έγινε η βελτιστοποίηση του παράλληλου κώδικα. Επιτύχαμε πολύ μεγάλη εξοικονόμηση χρόνου με την χρήση καταχωρητών για την αποθήκευση των ενδιάμεσων αποτελεσμάτων. Με αυτόν τον τρόπο αντί να προσπελαύνεται πολλαπλές φορές η global memory απο κάθε thread, χρησιμοποιείται μία τοπική μεταβλητή για την αποθήκευση του αποτελέσματος και στην συνέχεια αυτό εγγράφεται στην global memory όταν έχει ολοκληρωθεί ο υπολογισμός του.

Αποτελέσματα

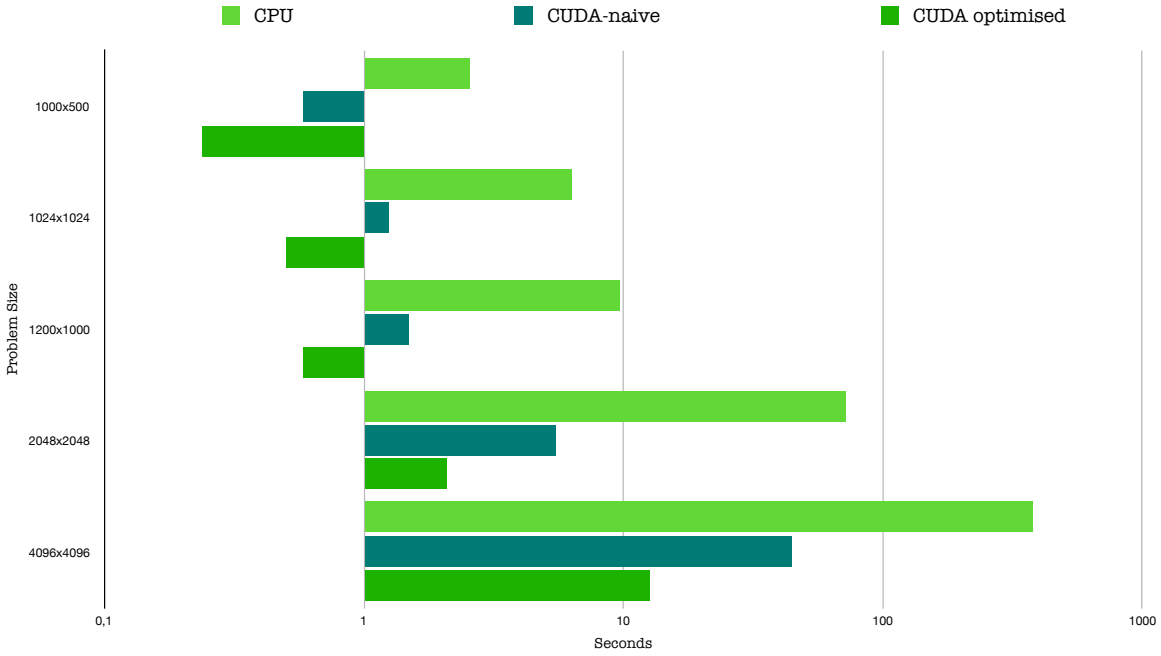
Παρακάτω συγκεντρώνονται τα αποτελέσματα επαναλαμβανόμενων εκτελέσεων του προγράμματος για διαφορετικά μεγέθη εισόδου.

Problem Size		Average Execution Time (seconds)			Speedup	
M	N	CPU	CUDA-naïve	CUDA optimised	CPU vs CUDA	CUDA vs CUDA
1000	500	2,550265	0,587833	0,239721	10,6	2,5
1024	1024	6,318746	1,239252	0,499820	12,6	2,5
1200	1000	9,712513	1,481306	0,584743	16,6	2,5
2048	2048	71,620694	5,502618	2,082896	34,4	2,6
4096	4096	380,505193	44,264532	12,603627	30,2	3,5

Το πρώτο speedup που υπολογίζεται αφορά την επιτάχυνση που δίνει ο βελτιωμένος κώδικας σε σχέση με την σειριακή εκτέλεση στη CPU. Το δεύτερο speedup αναφέρεται στην επιτάχυνση που έδωσαν οι βελτιστοποιημένοι cuda kernels σε σχέση με την naive εκδοχή τους.

Συμπεράσματα

Το πρώτο που παρατηρούμε είναι ότι η χρήση καταχωρητών αντί για προσπέλαση της global memory είναι απο μόνη της ικανή να μας δώσει πολύ μεγάλη βελτίωση του χρόνου εκτέλεσης. Επίσης γίνεται φανερό ότι όσο μεγαλώνει το μέγεθος του προβλήματος, τόσο μεγαλύτερο speedup επιτυγχάνεται που όμως μετά από ένα σημείο παραμένει σχετικά σταθερό.



4. Google Cloud Platform

Ως τελευταίο μέρος της εργασίας και περισσότερο για λόγους πειραματισμού, αποφασίσαμε να τρέξουμε τον κώδικα μας στην πλατφόρμα Google Cloud Platform αξιοποιώντας την πρόσβαση που μας παρέχει σε πιο σύγχρονο εξοπλισμό. Παρακάτω παρατίθενται τα τεχνικά χαρακτηριστικά του virtual machine που δημιουργήσαμε:

```
======== CPU ========
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    2
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 45
Model name:            Intel(R) Xeon(R) CPU @ 2.60GHz
Stepping:              7
CPU MHz:               2600.000
BogoMIPS:              5200.00
Hypervisor vendor:     KVM
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K

======== RAM ========
MemTotal:       16425608 kB

======== GPU ========
Device 0: "Tesla V100-SXM2-16GB"
  CUDA Driver Version / Runtime Version          10.0 / 10.0
  CUDA Capability Major/Minor version number:    7.0
  Total amount of global memory:                 16130 MBytes (16914055168 bytes)
  (80) Multiprocessors, ( 64) CUDA Cores/MP:     5120 CUDA Cores
  GPU Max Clock rate:                            1530 MHz (1.53 GHz)
  Memory Clock rate:                             877 Mhz
  Memory Bus Width:                              4096-bit
  L2 Cache Size:                                 6291456 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536),
                                                 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
```

```
Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
Run time limit on kernels:                     No
Integrated GPU sharing Host Memory:            No
Support host page-locked memory mapping:       Yes
Alignment requirement for Surfaces:            Yes
Device has ECC support:                        Enabled
Device supports Unified Addressing (UVA):      Yes
Device supports Compute Preemption:            Yes
Supports Cooperative Kernel Launch:            Yes
Supports MultiDevice Co-op Kernel Launch:      Yes
Device PCI Domain ID / Bus ID / location ID:   0 / 0 / 4
```

Αποτελέσματα

Παρακάτω παρατίθενται οι μετρήσεις που προέκυψαν από την εκτέλεση του κώδικα των δύο τελευταίων ερωτημάτων στο νέο περιβάλλον. Σε αυτό το σημείο πρέπει να αναφερθεί ότι δεν προχωρήσαμε σε βελτιστοποίηση του προγράμματος ώστε να ανταποκρίνεται καλύτερα στις πιθανές απαιτήσεις που θέτει η εκτέλεσή του σε μία αρκετά μεταγενέστερη κάρτα γραφικών, παραμόνο τρέξαμε τον ίδιο κώδικα για να δούμε πως θα συμπεριφερθεί στο νέο υλικό.

$y = A^T \cdot A \cdot x$						
Problem Size		**Execution Time (seconds)**			**Speedup**	
NX	NY	CPU	GPU - custom	GPU - cublas	CPU/GPUcustom	CPU/GPUBlas
4096	4096	0,147874	0,001020	0,000196	145,0	754,5
16000	16000	2,270194	0,009133	0,002410	248,6	942,0
16000	32000	4,556810	0,017551	0,004807	259,6	948,0
35840	17408	5,506848	0,021803	0,005807	252,6	948,3
65536	25600	14,889877	0,076916	0,015550	193,6	957,5

Covariance Matrix					
Problem Size		**Execution Time (seconds)**		**Speedup**	
M	**N**	**CPU**	**CUDA optimised**	**CPU/CUDA**	**ceid vs. google (gpus)**
1000	500	2,278675	0,045577	50,0	5,3
1024	1024	11,469601	0,122616	93,5	4,1
1200	1000	14,485506	0,142839	101,4	4,1
2048	2048	110,992461	0,473674	234,3	4,4
4096	4096	987,765404	2,449274	403,3	5,1

Με μία πρώτη ματιά παρατηρούμε ότι και μόνο η μετάβαση σε μία πιο ισχυρή κάρτα γραφικών είναι ικανή να μας δώσει πολλαπλάσιο speedup από αυτό που πετυχαίναμε. Επίσης είναι φανερό ότι η κυριότερη διαφορά έγκειται στην επεξεργαστική ισχύ της GPU και όχι της CPU αφού σε πολλές περιπτώσεις ο αρχικός επεξεργαστής κατέληγε σε μικρότερους χρόνους εκτέλεσης. Τέλος, πολλά από τα αρχικά συμπεράσματα ισχύουν και σε αυτήν την περίπτωση καθώς και εδώ βλέπουμε την υπεροχή της cuBLAS σε σχέση με την δική μας υλοποίηση ενώ παρατηρούμε ότι σε γενικές γραμμές, όσο μεγαλύτερο είναι το μέγεθος του προβλήματος τόσο καλύτερο speedup παίρνουμε.

Αναφορές

[1] Mark Harris, “How to Implement Performance Metrics in CUDA C/C++ | NVIDIA Developer Blog,” Nov. 2012. [Online]. Available: https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/

[2] “How to Access Global Memory Efficiently in CUDA C/C++ Kernels,” Jan. 2013. [Online]. Available: https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/

[3] “Matrix-Vector Multiplication Using Shared and Coalesced Memory Access,” Jun. 2012. [Online]. Available: https://github.com/uysalere/cuda-matrix-vector-multiplication/raw/master/vmp.pdf

[4] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, “Auto-tuning a high-level language targeted to gpu codes,” in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–10. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6339595