Λογισμικό και Προγραμματισμός Συστημάτων Υψηλής Επίδοσης

Ακαδημαικό Έτος 2018-2019

Πετράκης Κωσταντίνος AM:5878

Πετράκης Ευάγγελος AM:6339

Ομάδα Χρηστών 18

Περιγραφή συστήματος

Όλα τα πειράματα υλοποιήθηκαν σε φορητό υπολογιστή με κάρτα γραφικών NVIDIA® GeForce® GT 560M με τεχνολογία CUDA™ με τα εξής χαρακτηριστικά:

ποσότητα μνήμης : Αποκλειστική μνήμη VRAM 1.536 MB. Η διαθέσιμη μνήμη γραφικών μπορεί να επεκταθεί με τη χρήση της μνήμης συστήματος, μέσω της τεχνολογίας TurboCache™: έως 4.095 MB με εγκατεστημένη μνήμη συστήματος 8 GB όπως εμφανίζεται στον πίνακα ελέγχου της NVIDIA (με προεγκατεστημένο λειτουργικό σύστημα 64-bit).

τύπος μνήμης : GDDR5 Video RAM (συνδυασμός μνήμης DDR Video RAM και μνήμης συστήματος)

συνδεδεμένος δίαυλος : PCI Express®

Η κάρτα είναι αρχιτεκτονικής Fermi, και έχει Compute Capability 2.1. Παραθέτουμε τα φυσικά όρια της κάρτας με βάση τα οποία θα εξηγούμε σε κάθε περίπτωση τον τρόπο που παραλληλοποιήσαμε τον αντίστοιχο υπολογιστικό πυρήνα ώστε να πετύχουμε μέγιστη δυνατή απόδοση.

Physical Limits for GPU Compute Capability:	**2,1**
Threads per Warp	32
Max Warps per Multiprocessor	48
Max Thread Blocks per Multiprocessor	8
Max Threads per Multiprocessor	1536
Maximum Thread Block Size	1024
Registers per Multiprocessor	32768
Max Registers per Thread Block	32768
Max Registers per Thread	63
Shared Memory per Multiprocessor (bytes)	49152
Max Shared Memory per Block	49152
Register allocation unit size	128
Register allocation granularity	warp
Shared Memory allocation unit size	128
Warp allocation granularity	2

Παρατήρηση: Επισημαίνεται πως οι καταχωρητές της κάρτας είναι των 32-bit. Επομένως επειδή κάθε στοιχείο τύπου double απαιτεί 64 bits, για την αποθήκευση του χρειαζόμαστε 2 καταχωρητές των 32-bit. Με βάση αυτήν την παρατήρηση γίνεται ο υπολογισμός των απαιτούμενων καταχωρητών σε όλα τα παρακάτω.

Όλα τα προγράμματα υλοποιήθηκαν με την χρήση του Visual Studio 2015 και CUDA 8.0. Για την συνάρτηση χρονομέτρησης έχουμε τροποποιήσει τον κώδικα ώστε να δουλεύουν τα προγράμματα σε Windows.

Ασκηση 1

Ο υπολογιστικός πυρήνας για το συγκεκριμένο πρόβλημα είναι ο έξης:

```
__global__ void Convolution(double *a, double *b, int ni, int nj)
{
	int Col = blockDim.x*blockIdx.x + threadIdx.x;
	int Row = blockDim.y*blockIdx.y + threadIdx.y;
	double c11, c12, c13, c21, c22, c23, c31, c32, c33;

	c11 = +0.2;  c21 = +0.5;  c31 = -0.8;
	c12 = -0.3;  c22 = +0.6;  c32 = -0.9;
	c13 = +0.4;  c23 = +0.7;  c33 = +0.10;

	if ((Row>0)&& (Col>0)&& (Row < ni-1) && (Col < nj-1)) {

		b[Row*nj + Col] = c11 * a[(Row - 1)*nj + (Col - 1)] + c12 *
a[(Row + 0)*nj + (Col - 1)] + c13 * a[(Row + 1)*nj + (Col - 1)]
			+ c21 * a[(Row - 1)*nj + (Col + 0)] + c22 * a[(Row + 0)*nj
+ (Col + 0)] + c23 * a[(Row + 1)*nj + (Col + 0)]
			+ c31 * a[(Row - 1)*nj + (Col + 1)] + c32 * a[(Row + 0)*nj
+ (Col + 1)] + c33 * a[(Row + 1)*nj + (Col + 1)];

	}
}
```

Όπου ni και nj οι διαστάσεις του πίνακα στον οποίο θα εκτελέσουμε την συνέλιξη.
Εκτελούμε κλήση του υπολογιστικού πυρήνα ως εξής:

```
unsigned int BLOCK_SIZE_PER_DIM = 16;

	unsigned int numBlocksX = (NJ - 1) / BLOCK_SIZE_PER_DIM + 1;
	unsigned int numBlocksY = (NI - 1) / BLOCK_SIZE_PER_DIM + 1;


	dim3 dimBlock(BLOCK_SIZE_PER_DIM, BLOCK_SIZE_PER_DIM, 1);
	dim3 dimGrid(numBlocksX, numBlocksY, 1);

	// Launch a kernel on the GPU with one thread for each element.
	Convolution << <dimGrid, dimBlock>> >(dev_a, dev_b, NI, NJ);
```

Σημείωση: Ο πλήρης κώδικας για την άσκηση φαίνεται στο επισυναπτόμενο αρχείο που αντιστοιχεί στην άσκηση 1.

Όπου χρησιμοποιούμε δισδιάστατο πλέγμα με τον αριθμό των blocks σε κάθε διάσταση να υπολογίζεται όπως έχουμε δει και στις διαφάνειες του μαθήματος ανάλογα με το μέγεθος του προς επεξεργασία πίνακα.

Σε κάθε SM(streaming multiprocessor) θεωρητικά μπορώ να έχω μέχρι και 1536 νήματα. Επειδή όμως όπως φαίνεται από τον κώδικα μας κάθε thread χρησιμοποιεί 9 στοιχεία του

πίνακα A για να αποθηκεύσει το αποτέλεσμα σε 1 στοιχείο του πίνακα B(όπου οι A και B τύπου double) χρειάζεται 20 καταχωρητές των 32-bit. Επιπλέον κάθε νήμα χρησιμοποιεί 9 σταθερές c για τον υπολογισμό του αντίστοιχου στοιχείου του πίνακα B άρα απαιτεί επιπλέον 18 καταχωρητές των 32-bit.Τέλος κάθε νήμα απαιτεί 2 καταχωρητές για την αποθήκευση των δεικτών θέσης Row και Col. Δηλαδή συνολικά κάθε νήμα απαιτεί 20+18+2=40 καταχωρητές.

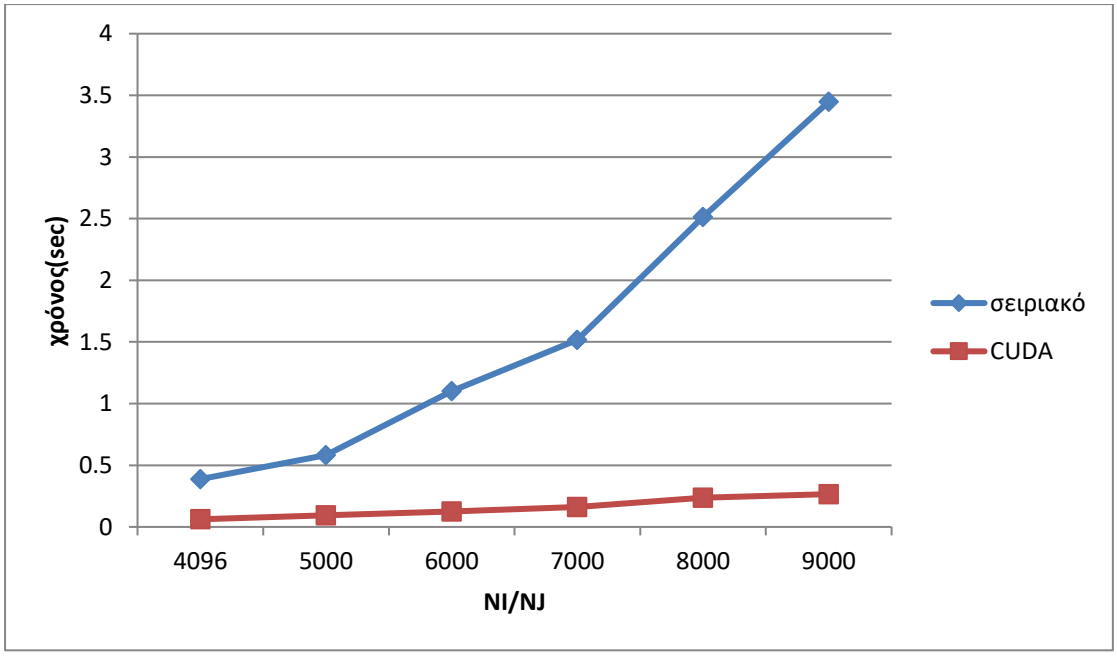
Επειδή κάθε SM χρησιμοποιεί μέχρι και 32.768 καταχωρητές, αυτό σημαίνει ότι μπορούμε να έχουμε μέχρι και 32.768/40=819 ενεργά νήματα σε κάθε SM. Με βάση αυτό λοιπόν δεν μπορούμε να χρησιμοποιήσουμε μέγεθος block 32x32 διότι κάθε block θα είχε 1024 νήματα και θα απαιτούσε 1024*40=40.960 καταχωρητές. Γι αυτό τον λόγο επιλέγουμε να χρησιμοποιήσουμε block 16x16 (256 νήματα). Τώρα κάθε SM μπορεί να περιέχει μέχρι και 3 blocks (3*256*40=30.720 καταχωρητές) άρα έχουμε 3*256=768 ενεργά νήματα, δηλαδή έχουμε 24 ενεργά warps ανά SM που αντιστοιχεί στο μισό της μέγιστης χωρητικότητας του SM της κάρτας μας που είναι 48 ενεργά warps/SM. Δηλαδή μόνο από αυτό περιμένουμε μέγιστη απόδοση 50%. (Σημειώνουμε ότι αν δεν μας περιόριζαν οι καταχωρητές με block 16x16 ιδανικά θα μπορούσαμε να έχουμε 6 blocks/SM δηλαδή 48 warps/SM που είναι και το μέγιστο δυνατό).

Με βάση τα παραπάνω λοιπόν λάβαμε τις παρακάτω μετρήσεις για διάφορα μεγέθη του μητρώου A.

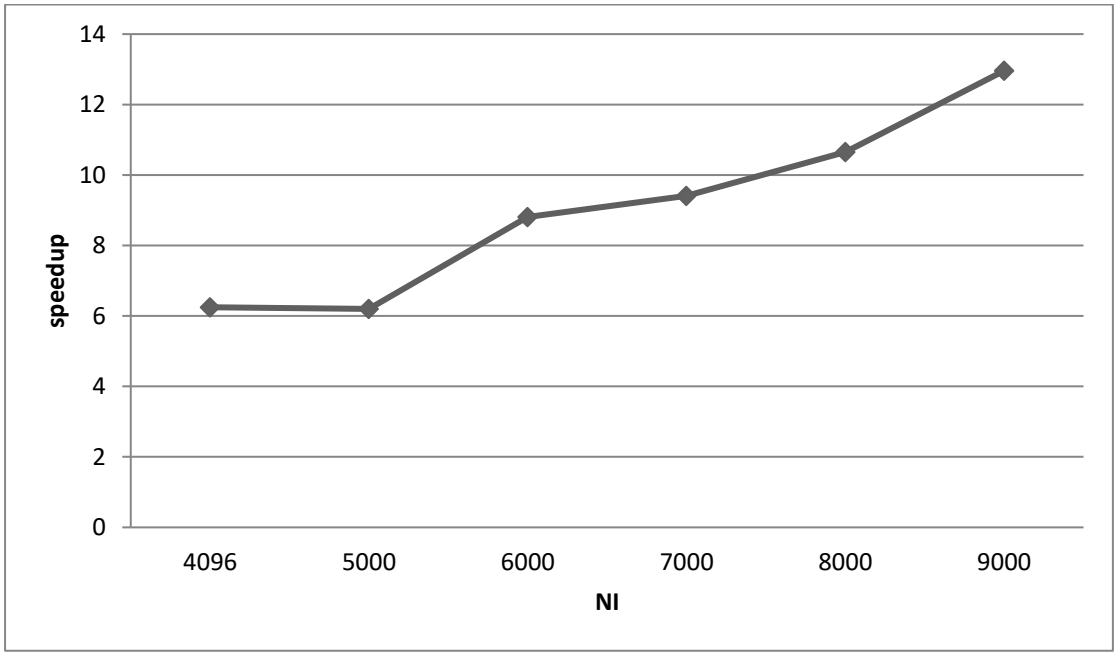
Μέγεθος NIxNJ	**Σειριακό πρόγραμα(sec)**	**Με χρήση CUDA(sec)**	**speedup**	**GPU utilization**
4096x4096	0,387760	0,062000	6,25	8%
5000x5000	0,583060	0,094000	6,20	5,4%
6000x6000	1,1013223	0,125000	8,81	10,5%
7000x7000	1,515584	0,161000	9,41	17,4%
8000x8000	2,514449	0,236000	10,65	21,3%
9000x9000	3,448880	0,266000	12,96	25%

Για μέγεθος πίνακα 9000x9000 είμαστε πολύ κοντά στην μέγιστη χωρητικότητα της κάρτας μας που είναι 1536MB. Αυτό γιατί θέλουμε να αποθηκεύσουμε τους πίνακες A και B στη μνήμη της κάρτας μας δηλαδή χρειάζεται να αποθηκεύσουμε 2*(9000x9000)=2*(81.000.000)=162.000.000 στοιχεία τύπου double. Αυτό ισοδυναμεί σε 162.000.000*8=1.296.000.000 bytes= 1.265.625 KB= 1236 MB που είναι πολύ κοντά στην χωρητικότητα της κάρτας μας.(Για πίνακες10.000x10.000 προκύπτει σφάλμα στη συνάρτηση cudaMalloc)

Παραθέτουμε γράφημα με τους χρόνους εκτέλεσης:



Ακολουθεί γράφημα που δείχνει την χρονοβελτίωση που πετυχαίνουμε με το να παραλληλοποιήσουμε το πρόγραμμα με χρήση της GPU συναρτήσει του μεγέθους των πινάκων.



Βλέπουμε ότι όσο αυξάνονται οι διαστάσεις του πίνακα τόσο αυξάνεται και η χρονοβελτίωση που πετυχαίνω με χρήση GPU.

Η χαμηλή αξιοποίηση της GPU που παρατηρούμε εδώ οφείλεται στα όσα εξηγήσαμε πιο πάνω για τους καταχωρητές που απαιτεί κάθε νήμα αλλά και στο γεγονός ότι τα νήματα προσπελαύνουν την καθολική μνήμη για τα στοιχεία των πινάκων εισόδου. Επειδή για τον υπολογισμό του κάθε στοιχείου του πίνακα B απαιτούνται 9 στοιχεία του πίνακα A έχουμε

πολύ μεγάλες απαιτήσεις σε bandwidth (πολύ περισσότερο από 150GB/s που είναι διαθέσιμα) και έτσι οι προσπελάσεις στην μνήμη περιορίζουν δραστικά την απόδοση της κάρτας όπως φαίνεται και από τον πίνακα πιο πάνω.

Ασκηση 2

Ο υπολογιστικός πυρήνας για αυτήν την άσκηση είναι:

```
__global__ void trans_norm_vector(double* A, double* x, double* tmp, double *y,
int nx, int ny)
{
	unsigned int tid = threadIdx.x + blockIdx.x*blockDim.x;
	__shared__ double x_s[BLOCK_SIZE];
	__shared__ double tmp_s[BLOCK_SIZE];
		double sum = 0.0;
		double res = 0.0;

	for (int m = 0; m < ((ny + BLOCK_SIZE - 1) / BLOCK_SIZE); m++) {
		if ((m*BLOCK_SIZE + threadIdx.x) < ny) x_s[threadIdx.x] =
x[threadIdx.x + m*BLOCK_SIZE];
		else x_s[threadIdx.x] = 0.f;
		__syncthreads();

		for (int e = 0; e < BLOCK_SIZE; e++) {
			//To A*x ginetai kata stiles-pio grigoro
			sum += A[tid + (e + BLOCK_SIZE*m)*nx] * x_s[e];
		}
		__syncthreads();
	}
	if (tid < nx) tmp[tid] = sum;

	for (int m = 0; m < ((ny + BLOCK_SIZE - 1) / BLOCK_SIZE); m++) {
		if ((m*BLOCK_SIZE + threadIdx.x) < ny) tmp_s[threadIdx.x] =
tmp[threadIdx.x + m*BLOCK_SIZE];
		else tmp_s[threadIdx.x] = 0.f;
		__syncthreads();

		for (int e = 0; e < BLOCK_SIZE; e++) {
			res += A[tid + (e + BLOCK_SIZE*m)*nx] * tmp_s[e];
		}
		__syncthreads();
	}
	if (tid < nx) y[tid] = res;
```

και εκτελούμε την συνάρτηση πυρήνα με μονοδιάστατο πλέγμα ως εξής:

```
dim3 dimBlock(BLOCK_SIZE);
dim3 dimGrid((NX+BLOCK_SIZE-1)/BLOCK_SIZE);
gettimeofday(&cpu_start, NULL);
trans_norm_vector << <dimGrid, dimBlock >> >(dev_a, dev_x, dev_tmp, dev_y, NX,
NY);
```

Όπως ζητείται εκτελούμε πρώτα το γινόμενο A*x και στην συνέχεια πολλαπλασιάζουμε το διάνυσμα που προκύπτει με τον A^T για να πάρουμε το διάνυσμα του αποτελέσματος y. Σημειώνουμε πως πρέπει να δεσμεύσουμε χώρο στην μνήμη της κάρτας μας για το μητρώο A(φορτώνεται μια φορά) το ενδιάμεσο διάνυσμα tmp και το τελικό διάνυσμα y.

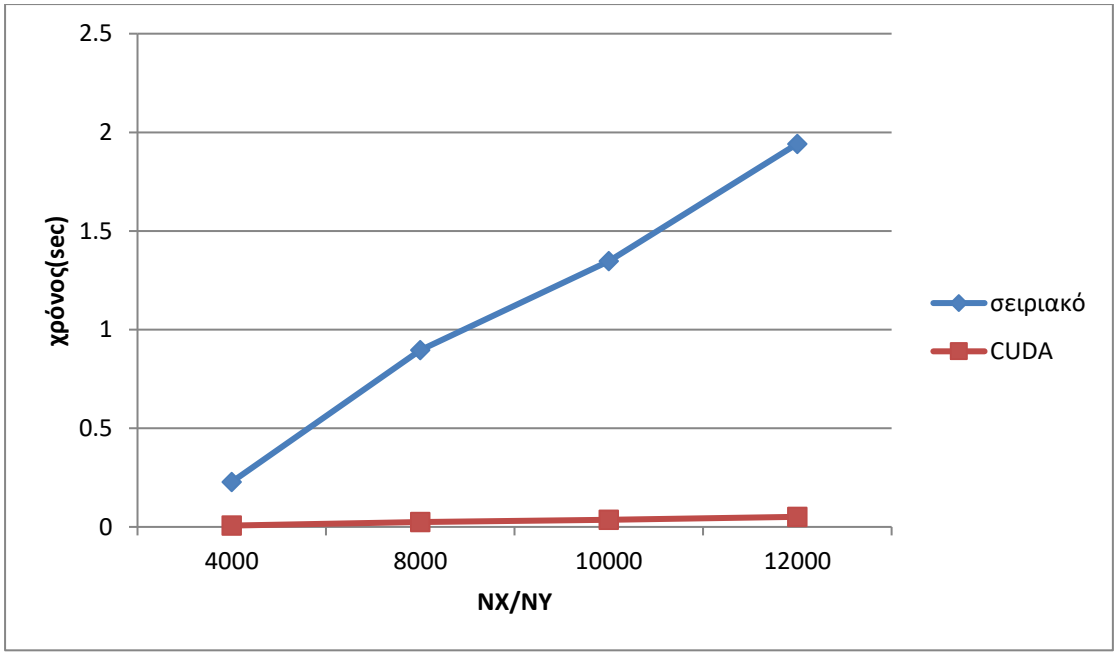
Ο υπολογιστικός πυρήνας αποτελείται από 2 κυρίως βρόγχους οι οποίοι εκτελούν από ένα γινόμενο μητρώο με διάνυσμα ο καθένας. Στον πρώτο βρόγχο υπολογίζουμε το tmp=A*x και στον επόμενο το τελικό αποτέλεσμα y=A^T*tmp. Για τον πολλαπλασιασμό μητρώου επι διάνυσμα αξιοποιούμε την κοινή μνήμη ως εξής: Φορτώνουμε στην κοινή μνήμη 'κομμάτια' του διανύσματος x μεγέθους BLOCK_SIZE και χρησιμοποιούμε τα στοιχεία του διανύσματος που έχουμε στην κοινή μνήμη.(Όπως αναλύθηκε στην θεωρία χρησιμοποιούμε πολλά threads μαζί για την συνεργατική φόρτωση των στοιχείων του x στην κοινή μνήμη). Επισημαίνουμε εδώ ότι δεν χρειάζεται να χρησιμοποιήσουμε κοινή μνήμη για τα στοιχεία του A καθώς αυτά θα χρησιμοποιηθούν μόνο μια φορά στο γινόμενο A*x, οπότε δεν γλυτώνουμε σε κόστος μεταφοράς μιας και θα τα φέρουμε από την μνήμη μία φορά ούτως η άλλως. Δηλαδή εδώ χρησιμοποιούμε την κοινή μνήμη σαν cache καθώς τα στοιχεία του διανύσματος x θα χρησιμοποιηθούν πολλές φορές.

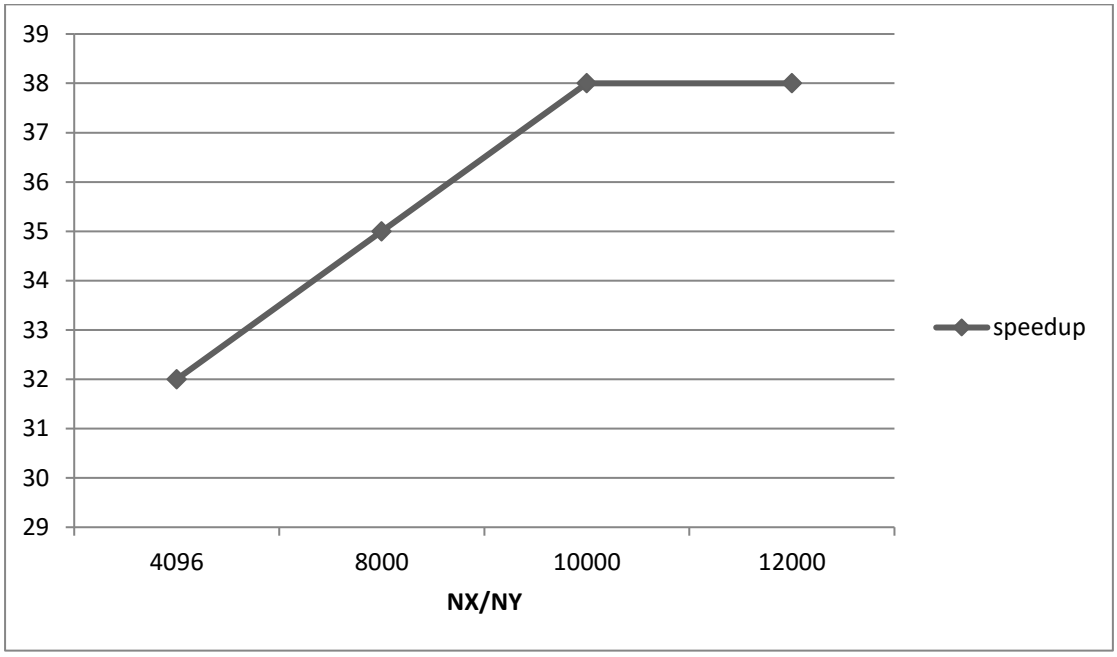
Επίσης λαμβάνοντας υπ' όψιν ότι το μητρώο A είναι αποθηκευμένο κατά γραμμές στην καθολική μνήμη της κάρτας, αντί να χρησιμοποιούμε κάθε thread για το γινόμενο BLOCK_SIZE στοιχείων μιας γραμμής με το αντίστοιχο τμήμα του x στην shared memory, οργανώνουμε το πλέγμα με τέτοιο τρόπο ώστε κάθε thread να χειρίζεται μια στήλη του πίνακα(στήλες του A^T=γραμμές A). Δηλαδή κάθε thread εκτελεί m(όπου m το πλήθος των blocks στο πλέγμα) γινόμενα μεγέθους BLOCK_SIZE στήλης επί το τμήμα του x στη κοινή μνήμη. Με αυτόν τον τρόπο διαμοιρασμού τα νήματα προσπελαύνουν συνεχόμενα στοιχεία της καθολικής μνήμης, αξιοποιούμε δηλαδή το Coalescing. Αυτό αν και δεν φαίνεται στην αναφορά έχει σημαντική επίδραση στον χρόνο μεταφοράς των δεδομένων από την καθολική μνήμη.

Με όμοια λογική εκτελούμε και το δεύτερο γινόμενο y=A^T*tmp για τον υπολογισμό του αποτελέσματος. Δεν μπορούμε να αξιοποιήσουμε το διάνυσμα x στην κοινή μνήμη για να εκτελέσουμε και τους 2 υπολογισμούς σε ένα βρόγχο καθώς χρειαζόμαστε ολόκληρο το διάνυσμα tmp για τον υπολογισμό y=A^T*tmp.

Παραθέτουμε πίνακα μετρήσεων χρόνου εκτέλεσης και την επιτάχυνση που πετυχαίνουμε και γράφημα των αντίστοιχων μεγεθών.

Μέγεθος NXxNY	**Σειριακό πρόγραμα(sec)**	**Με χρήση CUDA(sec)**	**speedup**	**GPU utilization**
4096x4096	0,227867	0,007000	32	33%
8000x8000	0,896451	0,025000	35	33%
10000x10000	1,347171	0,035451	38	66%
12000x12000	1,941804	0,051100	38	67%





Παρατηρούμε πως ακόμα και χωρίς 100% αξιοποίηση της κάρτας μας η χρονική επιτάχυνση που παίρνουμε είναι τεράστια.

Για όλες τις παραπάνω μετρήσεις χρησιμοποιήσαμε BLOCK_SIZE=256 ώστε να έχουμε τα μέγιστα δυνατά νήματα ανά SM(6 blocks *256thread/block=1536threads/SM). Με μεγέθη block 128 και 512 παρατηρήθηκε παρόμοια χρονική συμπεριφορά. Μόνο για BLOCK_SIZE<32 είχαμε χειρότερους χρόνους και χαμηλή αξιοποίηση της κάρτας πράγμα αναμενόμενο αφού το μέγεθος του warp είναι 32 νήματα , χρησιμοποιώντας λιγότερα ανά μπλοκ υποβαθμίζουμε την υπολογιστική ικανότητα της κάρτας(Warp divergence).

Ο περιοριστικός παράγοντας στην αξιοποίηση της κάρτας είναι όπως περιγράψαμε και στην άσκηση 1 ο αριθμός των καταχωρητών που χρησιμοποιεί το κάθε νήμα. Το χρησιμοποιούμενο μέγεθος της κοινής μνήμης εδώ για τα μπλοκ του διανύσματος x που

αποθηκεύουμε είναι BLOCK_SIZE*8=256*8=2048 bytes και δεν μας περιορίζει καθώς κάθε SM διαθέτει 48KB shared memory.

Ασκηση 3

Ο υπολογιστικός πυρήνας γι αυτήν την άσκηση είναι:

```
__global__ void covariance(double *data, double *mean, double *symmat, int m,
int n)
{
	int bx = blockIdx.x;
	int by = blockIdx.y;
	int tx = threadIdx.x;
	int ty = threadIdx.y;
	unsigned int Col = bx*blockDim.x + tx;
	unsigned int Row = by*blockDim.y + ty;

	__shared__ double data_s1[BLOCK_SIZE][BLOCK_SIZE];
	__shared__ double data_s2[BLOCK_SIZE][BLOCK_SIZE];

	double float_n = 3214212.01;
	while (Col < n) {
		double result = 0;
		for (int i = 0; i < m; i++)
			result += data[i*n + Col];
		mean[Col] = result/float_n;
		Col += gridDim.x*blockDim.x;
	}
	while (Col < n) {
		for (int i = 0; i < m; i++)
			data[i*n + Col] -= mean[Col];
		Col += gridDim.x*blockDim.x;
	}

	double Pvalue = 0.0;

	for (int l = 0; l <((n+BLOCK_SIZE-1)/ BLOCK_SIZE); l++) {
		// Colaborative loading of tiles into shared memory
		data_s1[ty][tx] = data[Row * n + l * BLOCK_SIZE + tx];
		data_s2[ty][tx] = data[Col + (l * BLOCK_SIZE + ty) * m];
		__syncthreads();
		for (int k = 0; k < BLOCK_SIZE; k++) {
			if ((l*BLOCK_SIZE + k) > m) break;
			Pvalue += data_s1[ty][k] * data_s2[k][tx];

		}
		  __syncthreads();

	}

	symmat[Row*n + Col] = Pvalue;


}
```

Και το πλέγμα στην GPU είναι:

```
dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE);
	dim3 dimGrid((M + BLOCK_SIZE - 1) / BLOCK_SIZE, (N + BLOCK_SIZE - 1) /
BLOCK_SIZE);
	gettimeofday(&cpu_start, NULL);
```

```
        covariance << <dimGrid, dimBlock >> >(dev_data, dev_mean, dev_symmat, M,
N);
```

Το πρώτο ζητούμενο εδώ είναι να υλοποιήσουμε το άθροισμα κάθε στήλης του μητρώου. Επειδή στη C τα δισδιάστατα μητρώα αποθηκεύονται κατά γραμμές στην μνήμη(row-major order) για να υλοποιήσουμε το άθροισμα στηλών εδώ αναθέτουμε σε κάθε thread το άθροισμα κάθε στήλης .Επιδιώκουμε δηλαδή αυτό που γνωρίζουμε από την θεωρία δηλαδή, ότι threads στο ίδιο warp είναι επιθυμητό να προσπελάζουν γειτονικά στοιχεία της καθολικής μνήμης. Με τον τρόπο αυτό λοιπόν τα threads πράγματι προσπελαύνουν συνεχόμενες θέσεις της καθολικής μνήμης δηλαδή αξιοποιούμε το Coalescing και έχουμε μεγάλη χρονική βελτίωση. Δεν χρειάζεται να χρησιμοποιήσουμε κοινή μνήμη ούτε και thread reduction για την άθροιση στηλών καθώς δεν έχουμε κανένα όφελος απ αυτό. Το άθροισμα στηλών με τον τόπο που το υλοποιούμε εδώ διατηρεί πλήρες αξιοποίηση του bandwidth της κάρτας μας(μέσω του Coalescing).Σημειώνουμε πως αν ο υπολογισμός απαιτούσε άθροισμα γραμμών θα είχαμε όφελος με το να χρησιμοποιήσουμε κοινή μνήμη. Στη συνέχεια αφού ολοκληρωθεί ο υπολογισμός του αθροίσματος κάθε στήλης αποθηκεύουμε το αποτέλεσμα σε ένα μονοδιάστατο διάνυσμα γραμμή mean μεγέθους N, όσες και οι στήλες του μητρώου, (σημειώνουμε πως απαιτείται να δεσμεύσουμε χώρο στην καθολική μνήμη της κάρτας και γι αυτό το διάνυσμα) και αφαιρούμε από κάθε στοιχείο κάθε στήλης το αντίστοιχο στοιχείο του διανύσματος. Και γι αυτόν τον υπολογισμό αξιοποιούμε μόνο την μία διάσταση του πλέγματος.

Για τον πολλαπλασιασμό του μητρώου με τον ανάστροφο του αξιοποιήσαμε και την κοινή μνήμη του κάθε block. Υλοποιούμε τον πολλαπλασιασμό του μητρώου με τον ανάστροφο του όπως υποδεικνύεται στις διαφάνειες του μαθήματος. Χωρίζουμε δηλαδή το μητρώο σε τετραγωνικά TILES μεγέθους BLOCK_SIZExBLOCK_SIZE, τα οποία φορτώνουμε στην κοινή μνήμη και εκτελούμε τον πολλαπλασιασμό των TILES ώστε να αξιοποιήσουμε την ταχύτητα προσπέλασης της κοινής μνήμης. Ο πολλαπλασιασμός με χρήση των TILES γίνεται με τον ίδιο τρόπο που απεικονίζεται στις διαφάνειες μόνο που τώρα και παίρνουμε και τα 2 TILES από τον ίδιο πίνακα στην καθολική μνήμη. Επισημαίνουμε πως έχουμε τροποποιήσει τον κώδικα των διαφανειών καθώς τώρα εκτελούμε πολλαπλασιασμό του μητρώου με τον ανάστροφο του. Με τον τρόπο που έχουμε υλοποιήσει τον πολλαπλασιασμό επειδή μεταφέρουμε στην κοινή μνήμη 2 TILES (κομμάτια του πίνακα data) του ίδιου πίνακα κάθε φορά υπάρχουν κάποιες επαναλήψεις(του βρόγχου l στον κώδικα) στις οποίες κρατάμε στην κοινή μνήμη το ίδιο τμήμα του μητρώου data και στα 2 TILES. Θα μπορούσαμε ίσως να χειριστούμε αυτήν την περίπτωση με κάποιες συνθήκες if στον βρόγχο αλλά θα αυξανόταν η πολυπλοκότητα του κώδικα , και θα δούμε πιο κάτω ότι ακόμη και με αυτή τη μικρή σπατάλη κοινής μνήμης ο κώδικας μας έχει πολύ καλή χρονική συμπεριφορά και απόδοση.

Χρησιμοποιούμε BLOCK_SIZE=16 διότι έτσι έχουμε 16x16=256 threads ανά block δηλαδή μπορούμε να έχουμε μέχρι και 6 ενεργά block ανά SM (6*256=1535 νήματα) . Δηλαδή με αυτήν την ανάθεση εκμεταλλευόμαστε πλήρως τις δυνατότητες του SM. Αλλά παρόλο που με μεγέθη block 8 ή 32 δεν πετυχαίνουμε την ίδια αξιοποίηση της κάρτας η χρονική βελτίωση είναι και πάλι πολύ μεγάλη.

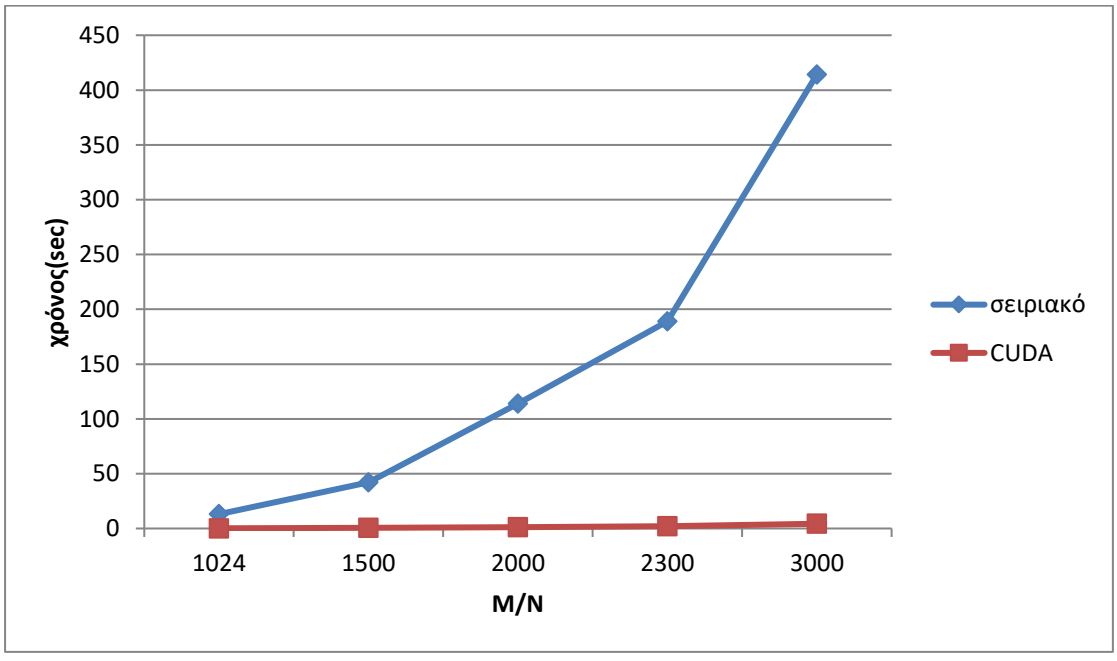
Επίσης δεν έχουμε αξιοποιήσει την πληροφορία που μας δίνεται ότι το μητρώο που προκύπτει είναι συμμετρικό και έχουμε υπολογίσει το μητρώο του αποτελέσματος σαν ένα κανονικό μητρώο.(Περισσότερα στη σημείωση στο τέλος).

Παραθέτουμε πίνακα μετρήσεων χρόνου εκτέλεσης και την επιτάχυνση που πετυχαίνουμε καθώς και την αξιοποίηση της GPU:

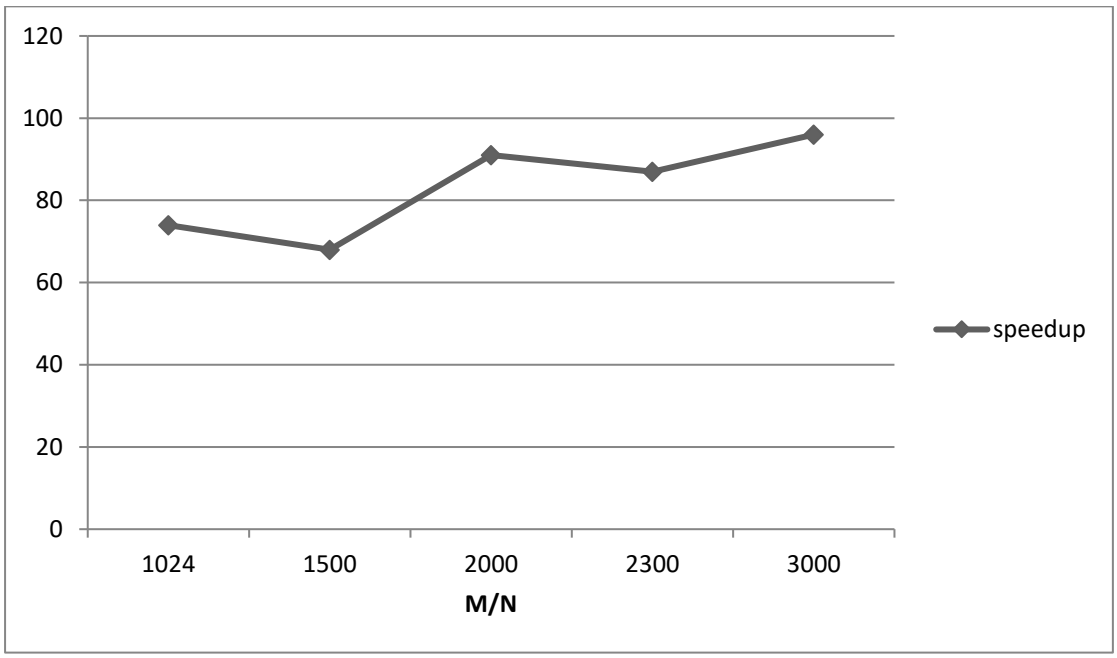
Μέγεθος MxN	**Σειριακό πρόγραμα(sec)**	**Με χρήση CUDA(sec)**	**speedup**	**GPU utilization**
1024x1024	13,081656	0,176000	74	44%
1500x1500	42,213009	0,613000	68	31,6%
2000x2000	113,819523	1,250000	91	77,5%
2300x2300	189,047289	2,168000	87	99,7%
3000x3000	414,228562	4,314880	96	100%

Δεν παραθέσαμε μετρήσεις για μεγαλύτερα μεγέθη καθώς το σειριακό πρόγραμμα είχε υπερβολικά μεγάλο χρόνο εκτέλεσης.

Ακολουθούν γραφήματα του χρόνου συναρτήσει της εισόδου καθώς και την επιτάχυνσης που πετυχαίνουμε

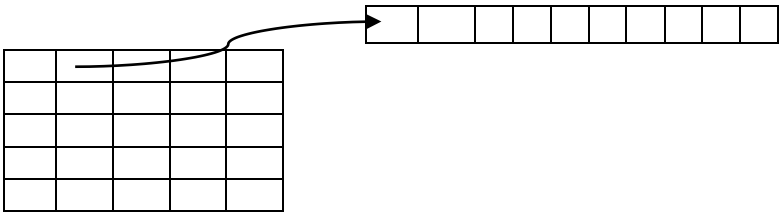


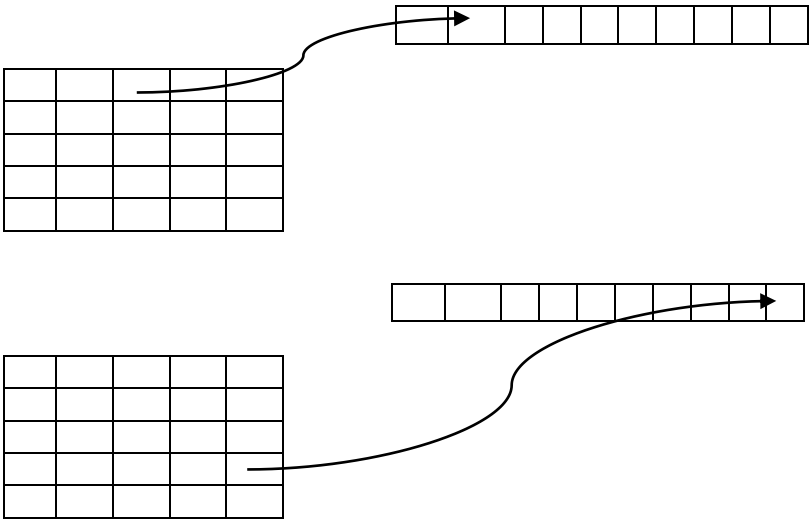
Επειδή οι χρονικές διαφορές είναι πολύ μεγάλες δεν έχουμε πλήρη εικόνα για τα πρώτα στιγμιότυπα του προβλήματος από το γράφημα. Ωστόσο μπορούμε να δούμε την τεράστια διαφορά στους χρόνους εκτέλεσης όσο μεγαλώνουν οι διαστάσεις του μητρώου.



Παρατηρούμε τεράστια βελτίωση με την χρήση GPU. Η εφαρμογή εκτελείται μέχρι και 90 φορές πιο γρήγορα με την χρήση GPU για μεγαλύτερα στιγμιότυπα. Επίσης εδώ έχουμε αξιοποίηση της κάρτας που αγγίζει το 100%. Αυτό γιατί η συγκεκριμένη εφαρμογή επιδέχεται παραλληλοποίησης με χρήση CUDA σε μεγάλο ποσοστό. Η φύση του υπολογισμού είναι τέτοια που εκτελείται πολύ καλύτερα σε αρχιτεκτονικές GPU από ότι σε CPU. Επίσης στον συγκεκριμένο υπολογιστικό πυρήνα έχουμε πολλές πράξεις ανά μεταφορά από την καθολική μνήμη για αυτό και έχουμε τόσο καλή απόδοση ακόμη και με τις αστοχίες που περιέχει ο κώδικας μας όπως εξηγήσαμε πιο πάνω.

Σημείωση: Επιχειρήσαμε να αποθηκεύσουμε το τελικό αποτέλεσμα σε ένα μονοδιάστατο πίνακα που θα αντιστοιχούσε στο άνω τριγωνικό μέρος του μητρώου αποτελέσματος(χωρίς την διαγώνιο), αλλά λαμβάναμε λάθη στην διευθυνσιοδότηση του διανύσματος. Ο τρόπος που επιχειρήσαμε να το πετύχουμε αυτό ήταν ο εξής:





Όπου έστω symmat το μητρώο του αποτελέσματος και sym_vector το διάνυσμα που περιέχει το άνω τριγωνικό μέρος του μητρώου. Ο ψευδοκώδικας για την παραπάνω αντιστοίχηση είναι:

idx = blockDim.x ∗ blockIdx.x + threadIdx.x

i = idx/N

j = idx%N

if i < j and i < N and j < N then

k1 = i × N − i × (i + 1) /2 + j − i

k2 = j × N + i

sym_vector[k1 − 1] = symmat[k2]

end if

Βρήκαμε τον παραπάνω τρόπο ανάθεσης στο άρθρο:

Fast-GPU-PCC: A GPU-Based Technique to Compute Pairwise Pearson's Correlation Coefficients for Time Series Data-fMRI Study.

Taban Eslami and Fahad Saeed * ID Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008, USA; taban.eslami@wmich.edu * Correspondence: fahad.saeed@wmich.edu; Tel.: +1-269-276-3156 Received: 2 March 2018; Accepted: 17 April 2018; Published: 20 April 2018

<u>ΣΗΜΕΙΩΣΗ:</u> Στην αναφορά έχουμε συμπεριλάβει ατομικά τα αρχεία .cu για την κάθε άσκηση, αλλά και τους πλήρεις φακέλους του Visual Studio για την κάθε άσκηση διότι δεν γνωρίζουμε την πλατφόρμα στην όποια θα δοκιμαστούν οι κώδικες.