Εργασία 2 στην ενότητα Πολυεπεξεργαστικά Υπολογιστικά Συστήματα: Υλοποίηση του παιχνιδιού Conway Game of Life

Τρίτη 5 Δεκεμβρίου, 2017

Ομάδα Εργασίας

αα	Ονοματεπώνυμο	Αριθμός Μητρώου	Email ή τηλέφωνο
1.	ΖΕΡΚΕΛΙΔΗΣ ΔΗΜΗΤΡΙΟΣ	Π14046	dimzerkes@gmail.com

1. Εισαγωγή

Η εφαρμογή περιλαμβάνει την υλοποίηση του παιχνιδιού Conway Game of Life. στο μοντέλο προγραμματισμού CUDA.

2. Σύντομη περιγραφή της εφαρμογής

To Conway Game of Life (https://en.wikipedia.org/wiki/Conway's_Game_of_Life) είναι ένα cellular automaton που προτάθηκε από τον μαθηματικό John Horton Conway το 1970.

Είναι παιχνίδι χωρίς παίκτες όπου η εξέλιξη του παιχνιδιού καθορίζεται από την αρχική κατάσταση και δεν απαιτείται αλληλεπίδραση με τον παίκτη. Ο παίκτης πρέπει απλά να ορίσει την αρχική κατάσταση του αυτόματου.

Το παιχνίδι αποτελείται από ένα σύμπαν (universe), ένα 2Δ πλέγμα κελιών, όπου κάθε κελί είναι σε κατάσταση: alive ή dead. Σε κάθε γενιά (generation), ένα κελί «πεθαίνει», «επιζεί» ή «γεννιέται» ανάλογα με την κατάστασή του και την κατάσταση των γειτονικών του κελιών. Για παράδειγμα, στην βασική έκδοση του παιχνιδιού ισχύουν οι εξής κανόνες:

- Ένα ζωντανό κελί με 2 ή 3 ζωντανούς γείτονες επιζεί
- Ένα ζωντανό κελί πεθαίνει όταν έχει 0 ή 1 ζωντανό γείτονα (λόγω μοναξιάς) και περισσότερους από 3 ζωντανούς γείτονες (λόγω υπερπληθυσμού)
- Ένα πεθαμένο κελί με ακριβώς 3 ζωντανούς γείτονες γεννιέται (ως αναπαραγωγή)

Οι κανόνες εφαρμόζονται ταυτόχρονα σε όλα τα κελιά του πλέγματος για να παράγουν την επόμενη γενιά.

Υπάρχουν πολλές παραλλαγές του παιχνιδιού:

- Διαφορετικές συνθήκες για την επιβίωση, τον θάνατο και την γέννηση
- Αριθμός καταστάσεων ανά κελί
- Αριθμός διαστάσεων, π.χ. 3D
- Διάταξη των κελιών, π.χ. εξαγωνικό δικτύωμα
- Αριθμός και βαρύτητα γειτόνων

Στην παρούσα εργασία θα υλοποιήσετε μια παραλλαγή με τους εξής κανόνες:

- Κάθε κελί αλληλεπιδρά με τα γειτονικά του σε μια ακτίνα 5 κελιών (οριζόντια, κάθετα & διαγώνια). Άρα το κελί βρίσκεται στο κέντρο ενός τετραγώνου διαστάσεων 11x11 (έχει 120 γείτονες).
- Ένα ζωντανό κελί με 34 έως 58 ζωντανούς γείτονες επιζεί.
- Ένα ζωντανό κελί πεθαίνει όταν έχει λιγότερους από 34 (λόγω μοναξιάς) και περισσότερους από 58 ζωντανούς γείτονες (λόγω υπερπληθυσμού).
- Ένα πεθαμένο κελί με 34 έως 45 ζωντανούς γείτονες γεννιέται (ως αναπαραγωγή).

Το πρόγραμμα που θα υλοποιήσετε θα πρέπει να εκτελεί τα παρακάτω βήματα:

1. Η CPU θα παράγει το σύμπαν (ένα 2Δ πλέγμα κελιών διαστάσεων MxM) αρχικοποιώντας ψευδοτυχαία την κατάσταση των κελιών.
2. Η CPU θα αντιγράφει τον πίνακα στην μνήμη της GPU.
3. Η GPU θα υπολογίζει την επόμενη γενιά και τον αριθμό των «ζωντανών» και «πεθαμένων» κελιών
4. Η GPU θα στέλνει στην CPU τα αποτελέσματα (τους 2 μετρητές)
5. Η CPU θα απεικονίζει τα αποτελέσματα (μετρητές) στην οθόνη
6. Η GPU θα στέλνει το νέο σύμπαν στην CPU και η CPU θα το απεικονίζει στην οθόνη
7. Επανάληψη των βημάτων 3-6

Σημείωση: Το βήμα 6 είναι προαιρετικό, ώστε να απεικονίσετε τα αποτελέσματα του παιχνιδιού (την εξέλιξη του σύμπαντος) στην οθόνη σας. Είναι πιθανόν, η υλοποίηση του βήματος 6 να επηρεάσει αρνητικά την επιτάχυνση του συστήματος λόγω της μεταφοράς δεδομένων από την GPU στην CPU.

Τεχνικές βελτιστοποίησης:

- Θα πρέπει να παραλληλοποιήσετε τόσο τον υπολογισμό της νέας γενιάς όσο και τον υπολογισμό των 2 μετρητών.
- Θα πρέπει να υλοποιήσετε 2 εκδόσεις του προγράμματος: στην $1^{η}$ έκδοση μην χρησιμοποιήσετε shared memory και στην $2^{η}$ έκδοση χρησιμοποιήστε shared memory. Να αναλύσετε πόσο συμβάλλει η χρήση της shared memory στην επιτάχυνση της εκτέλεσης της εφαρμογής.
- Για τον υπολογισμό των 2 μετρητών να χρησιμοποιήσετε atomic operations.
- Να υπολογίσετε τον χρόνο εκτέλεσης του πρόγραμματος: (α) όταν εκτελείται ακολουθιακά (δηλαδή στην CPU) και (β) όταν εκτελείται παράλληλα (στην GPU).

3. Σύσταση Ομάδων – Παράδοση Εργασίας

- ☑ Κάθε ομάδα εργασίας μπορεί να αποτελείται μέχρι 2 φοιτητές. Συμπληρώστε τα στοιχεία των μελών της ομάδας στον πίνακα της πρώτης σελίδας αυτής της εκφώνησης (όνομα, αριθμός μητρώου και τουλάχιστον ένα email επικοινωνίας ή τηλέφωνο για κάθε ομάδα).
- ☑ Θα πρέπει να παραδώσετε: (α) τον κώδικα του προγράμματος σε C (για την ακολουθιακή εκτέλεση), (β) τον κώδικα του προγράμματος σε Cuda (για την παράλληλη εκτέλεση) και (γ) σύντομη τεκμηρίωση του κώδικα και σχολιασμό των αποτελεσμάτων.
- ☑ Η παράδοση της εργασίας πρέπει να γίνει μέχρι την **Κυριακή 4 Φεβρουαρίου, 2018**.
- ☑ Θα οριστεί ημερομηνία για την εξέταση και της εργασίας 1 και της εργασίας 2.
- ☑ Η παράδοση της εργασίας θα γίνει σε ηλεκτρονική μορφή στην ιστοσελίδα του μαθήματος.

Τεκμηρίωση Εργασίας

(αρχίστε από εδώ την τεκμηρίωση της εργασίας σας – μονό διάστιχο – 10pt γραμματοσειρά).

ΚΩΔΙΚΑΣ CPU

Ξεκινάμε με τον κώδικα στην CPU.

```
int worldX, worldY;

printf("Please enter the width of the array : ");
scanf("%d", &worldX);

printf("Please enter the height of the array : ");
scanf("%d", &worldY);
```

Εισαγωγή διαστάσεων πίνακα.(προτείνεται δύναμη του 2)

```
int population = worldX * worldY;
int* world = (int*)malloc(sizeof(int*) * population);
int* count = (int*)malloc(sizeof(int*) * population);
int* state = (int*)malloc(sizeof(int*) * population);
int alive = 0;
```

Εδώ υπολογίζουμε τον πληθυσμό και έπειτα ελευθερώνουμε χώρο στη μνήμη για τους πίνακες world,count,state.

```
// Random initial polulation
srand(time(NULL));
for (int row = 0; row < worldY; row++)
        for (int col = 0; col < worldX; col++){
                int rand_val = rand() % 2;
                world[row*worldX + col] = rand_val;
                if (rand_val) alive += 1;
        }

cout << alive;
```

Θέτουμε πληθυσμό τυχαία και βγάζουμε και τους ζωντανούς την ίδια στιγμή.

Τώρα μεταφέρουμε τα στοιχεία του πίνακα world στον πίνακα state.

```
memcpy(state, world, sizeof(int*) * population);
```

Θέτουμε μια μεταβλητή lowest ίση με τον μεγαλύτερο ακέραιο .

```
int lowest = INT_MAX;
```

Παρακάτω είναι ο υπολογισμός των γειτόνων:

```
        while (alive > 0){

                // Calculate alive neighbours and print polulation
                for (int row = 0; row < worldY; row++){
                        //cout << "| ";
                        int tmp = 0;
                        for (int col = 0; col < worldX; col++){

                                for (int off_row = row - 5; off_row <= row + 5; off_row++){
                                        for (int off_col = col - 5; off_col <= col + 5; off_col++)
                                                if (!(off_row < 0 || off_row >= worldY || off_col < 0 || off_col >= worldX || (off_row == row &&
off_col == col))) //or substract itself
                                                        tmp += world[off_row*worldX + off_col];//tmp = tmp;
                                }

                                //cout << world[row * worldX + col] << " | ";
                                count[row * worldX + col] = tmp;
                                if (tmp >= 34 && tmp <= 58){
                                        if (tmp <= 45 && state[row * worldX + col] == 0){
                                                state[row * worldX + col] = 1;
                                                alive += 1;
                                        }

                                }
                                else {
                                        if (state[row * worldX + col] == 1){
                                                state[row * worldX + col] = 0;
                                                alive -= 1;
                                        }

                                }
                                tmp = 0;
                        }

                        //cout << "\n";
                }
```

όσο έχουμε ζωντανούς ξεκινάμε με διπλό loop που διατρέχει όλο τον πίνακα ο οποίος ειναι μονοδιάστατος αλλά τον διατρέχουμε σαν 2 διαστάσεων λόγω της χρήσης της θέσης row*worldX + col.

Για κάθε στοιχείο λοιπόν ξανατρέχουμε διπλό loop το οποίο τσεκάρει όλους τους γείτονες σε απόσταση 5 όπως ζητάει η άσκηση. Έπειτα με τις κατάλληλες συνθήκες τσεκάρουμε αν είμαστε εντός ορίων πίνακα ή δε μετράμε το ίδιο στοιχείο.

Στην συνέχεια τρέχουν οι συνθήκες που δείχνουν που δείχνουν τη μοίρα του κελιου και η τιμή alive αλλάζει αναλόγως.

Οι νέες τιμές της νέας γενίας περνάν στον πίνακα state και μετά με την εξής εντολή

```
memcpy(world, state, sizeof(int*) * population);
```

περνάν στον πίνακα world πάλι και το loop συνεχίζει για την επόμενη γενία ώσπου μηδενιστεί το alive.

```
                //      cout << "\n";
                //}
                if (alive < lowest){
                        cout << '\n' << alive;
                        lowest = alive;
                }
        }
```

Εδώ εκτυπώνουμε τα ζωντανά κελιά.

Θα μπορούσαμε με μια αφαίρεση να εκτυπώσουμε και τα νεκρά (population-alive).

ΚΩΔΙΚΑΣ GPU – NO SHARED MEMORY USED

Θα τονίσουμε κάποια διαφορετικά σημεία κώδικα.

```
srand(time(NULL));
for (int i = 0; i < worldX; i++)
{
        for (int j = 0; j < worldY; j++)
        {
                if ((i + j) % 2 == 0) {
                        world[i*worldX + j] = 1;
                }
                else {
                        world[i*worldX + j] = 0;
                }
                //printf("%d ", c[i*arraySizex + j]);
                if (world[i*worldX + j] == 1)
                {
                        alive++;
                }
        }
        //printf("\n");
}
```

Σα πίνακα περνάμε συγκεκριμένο πίνακα που είναι ίδιος πάντα για να ελέγξουμε τα αποτελέσματα με τον κώδικα της shared.

Δημιουργούμε χώρο για τον pointer του pointer του alive στη μνήμη.

```
cudaMalloc((void**)&d_alive, sizeof(int));
cudaMemcpy(d_alive, &alive, sizeof(int), cudaMemcpyHostToDevice);
```

Αντιγράφω μετά το alive στη θέση μνήμης του d_alive στην GPU.

```
cudaMemcpy(d_alive, &alive, sizeof(int), cudaMemcpyHostToDevice);

int gridx = 16;
int gridy = 16;
dim3 grid((worldX / gridx) + 1, (worldY / gridy) + 1);
dim3 blockSize(gridx, gridy);

int* d_world;
int* state;
size_t size = sizeof(int*) * population;
cudaMalloc(&d_world, size);
cudaMemcpy(d_world, world, size, cudaMemcpyHostToDevice);
cudaMalloc(&state, size);
cudaMemcpy(state, world, size, cudaMemcpyHostToDevice);
```

Εδώ δηλώνω μέγεθος μπλοκ που είναι 16χ16.

Δηλώνουμε απο κάτω πόσα μπλοκ θα υπάρχουν με την εντολή dim3 grid...

Δημιουργούμε pointer για το d_world και για το state και δηλώνουμε το μέγεθος του πίνακα για να το περάσουμε παρακάτω όταν μεταφέρουμε από τον επεξεργαστή στην κάρτα γραφικών τα στοιχεία που θέλουμε και για να δημιουργήσουμε χώρο στην d_world και state στην cpu.

Περνάμε τα στοιχεία της world στη θέση μνήμης στην κάρτα γραφικών d_world και state.

Παρακάτω έχουμε τον αλγόριθμο που τρέχει όσο έχουμε ζωντανά κελιά.

```
cudaMemcpy(state, world, size, cudaMemcpyHostToDevice);

while (alive > 0)
{
        t << < grid, blockSize >> >(d_world, state, worldX, worldY, d_alive);
        cudaMemcpy(world, state, size, cudaMemcpyDeviceToHost);
        cudaMemcpy(d_world, state, size, cudaMemcpyDeviceToDevice);
        cudaMemcpy(&alive, d_alive, sizeof(int), cudaMemcpyDeviceToHost);
        cudaDeviceSynchronize();

        //for (int row = 0; row < worldY; row++){
        //      printf("| ");
        //      for (int col = 0; col < worldX; col++){
        //              printf("%d | ", test[row * worldX + col]);

        //      }
        //      printf("\n ");
        //}

        printf("%d\n", alive);
}

cudaFree(d_world);
free(world);
return 0;
```

Καλούμε το kernel με αριθμό μπλοκ grid και μέγεθος μπλοκ blockSize και περνάω τις μεταβλητές με τη σειρά d_world state worldX,worldY, d_alive. Έπειτα περνάμε τις τιμές του state στον world πίσω στον επεξεργαστή.

Ο state πίνακας έχει την επόμενη γενία της world γενικά στον αλγόριθμο.

Έπειτα μεταφέρω μέσα στην μνήμη της gpu τις τιμές τις state στην d_world (DeviceToDevice) κάτι το οποίο αυξάνει ταχύτητα.

Τέλος επαναφέρω στην cpu την τιμή alive από την gpu.

Kernel function:

```
__global__ void t(int* world, int* state, int worldX, int worldY, int* a)
{
        int col = blockIdx.x*blockDim.x + threadIdx.x;
        int row = blockIdx.y*blockDim.y + threadIdx.y;


        if (row < worldY && col < worldX){
                int pos = row * worldX + col;
                int tmp = 0;

                for (int off_row = row - 5; off_row <= row + 5; off_row++){
                        for (int off_col = col - 5; off_col <= col + 5; off_col++)
                                if (!(off_row < 0 || off_row >= worldY || off_col < 0 || off_col >= worldX || (off_row == row && off_col == col)))
or substract itself
                                        tmp += world[off_row*worldX + off_col];//tmp = tmp;
                }

                //cout << world[row * worldX + col] << " | ";

                if (tmp >= 34 && tmp <= 58){
                        if (tmp <= 45 && state[pos] == 0){
                                state[pos] = 1;
                                atomicAdd(a, 1);
                        }
                }
                else {
                        if (state[pos] == 1){
                                state[pos] = 0;
                                atomicSub(a, 1);
                        }
                }


        }
}
```

αρχικά δηλώνουμε τις μεταβλητές row – col για να μας βοηθήσουν να διατρέξουμε τον πίνακα.

Ελέγχουμε αν περνάνε τα όρια του πίνακα world ώστε να μη βγούμε εκτός.

Έπειτα κρατάμε στην μεταβλητή pos τη θέση του στοιχείου , μηδενίζουμε τον μετρητή tmp και βρίσκουμε τους γείτονες του στοιχείου στη θέση POS και ανάλογα αυξάνουμε τον μετρητή.

Στη συνέχεια ελέγχουμε την τιμή του tmp και ανάλογα αποφασίζουμε τη μοίρα του στοιχείου στη θέση pos στον πίνακα state που είναι η επόμενη γενία πάντα.

Παράλληλα με τη μοίρα του state με atomic operation αυξάνουμε την τιμή alive ή τη μειώνουμε.

ΚΩΔΙΚΑΣ GPU – SHARED MEMORY USED

```
int arraySizex = 1024;
int arraySizey = 1024;
int alive = 0;
int size = arraySizex * arraySizey;
int* c = (int*)malloc(sizeof(int*) * size);

for (int i = 0; i < arraySizex; i++)
{
        for (int j = 0; j < arraySizey; j++)
        {
                if ((i + j) % 2 == 0) {
                        c[i*arraySizex + j] = 1;
                }
                else {
                        c[i*arraySizex + j] = 0;
                }
                //printf("%d ", c[i*arraySizex + j]);
                if (c[i*arraySizex + j] == 1)
                {
                        alive++;
                }
        }
        //printf("\n");
}
printf("1 generation alive :%d and dead :%d \n", alive, size - alive);
```

Φτιάχνουμε πίνακα 1024x1024 θέτουμε την τιμή alive = 0 , υπολογίζουμε το μέγεθος του πίνακα και ελευθερώνουμε χώρο στην μνήμη και περνάμε σ αυτόν τον ίδιο πίνακα κάθε φορά.Έπειτα εκτυπώνουμε την 1η γενία με ζωντανούς και νεκρούς.

```
cudaError_t cudaStatus = addWithCuda(c, arraySizex, arraySizey, size, alive);
cudaStatus = cudaDeviceReset();
printf("Press Any Key to Continue\n");
getchar();
free(c);
return 0;
```

Αν δεν υπάρχει error καλείται η addwithcuda με τις μεταβλήτες αυτές .

```
cudaError_t addWithCuda(int *c, int width, int height, int size, int alive)
{
        clock_t tic = clock();
        dim3 threadsPerBlock(32, 32);
        dim3 numBlocks(width / threadsPerBlock.x, height / threadsPerBlock.y);
        cudaError_t cudaStatus;

        int* dev_c;
        int* dev_n;
        int* dev_alive;

        cudaStatus = cudaSetDevice(0);
        cudaStatus = cudaMalloc((void**)&dev_alive, sizeof(int));
        cudaStatus = cudaMemcpy(dev_alive, &alive, sizeof(int), cudaMemcpyHostToDevice);
        cudaStatus = cudaMalloc(&dev_c, size * sizeof(int));
        cudaStatus = cudaMemcpy(dev_c, c, size * sizeof(int), cudaMemcpyHostToDevice);
        cudaStatus = cudaMalloc(&dev_n, size * sizeof(int));
        cudaStatus = cudaMemcpy(dev_n, c, size * sizeof(int), cudaMemcpyHostToDevice);



        int generation = 2;
        int next_alive = 0;
```

Εδώ ουσιαστικά κάνουμε ότι και πριν αλλα για μπλοκ 32χ32 ενώ έχουμε βάλει και χρόνο να μετράει.

Δημιουργούμε τα pointers dev_c, dev_n , dev_alive. Ο πρώτος είναι για τα στοιχεία του αρχικού πίνακα c , ο δεύτερος για τη νέα γενία και το 3ο για τον μετρητή alive. Αυτές οι θέσεις μνήμης θα περαστούν στην κάρτα γραφικών.

Παρακάτω είναι ο αλγόριθμος που υπολογίζει τις ζητούμενες τιμές όσο υπάρχουν ζωντανά κελιά ή δεν έχουμε τον ίδιο αριθμό συνέχεια.

```
    while (alive > 0 && (next_alive != alive))
    {
            next_alive = alive;
            addKernel << < numBlocks, threadsPerBlock >> > (dev_c, dev_n, width, height, dev_alive);
            cudaStatus = cudaMemcpy(&alive, dev_alive, sizeof(int), cudaMemcpyDeviceToHost);
            cudaStatus = cudaMemcpy(c, dev_n, size * sizeof(int), cudaMemcpyDeviceToHost);
            cudaStatus = cudaMemcpy(dev_c, c, size * sizeof(int), cudaMemcpyHostToDevice);
            cudaStatus = cudaDeviceSynchronize();


            //printf("%d generation : \n",generation);
            //for (int row = 0; row < width; row++){
            //for (int col = 0; col < height; col++){
            //printf("%d ", c[row * width + col]);

            //}
            //printf("\n");
            //}

            printf("%d generation alive :%d and dead:%d \n", generation, alive, size - alive);
            generation++;

    }
    clock_t toc = clock();

    printf("%d generations time elapsed: %f seconds\n",generation-1, (double)(toc - tic) / CLOCKS_PER_SEC);
    cudaFree(dev_c);
    cudaFree(dev_n);
    cudaFree(dev_alive);
    return cudaStatus;
```

Στο τέλος εκτυπώνουμε γενιά , ζωντανά , νεκρά όταν ολοκληρωθεί εκτωπώνουμε τα ίδια + χρόνο.

Kernel function:

```
__global__ void addKernel(int* c, int* n, int width, int height, int* alive)
{
        int col = blockIdx.x*blockDim.x + threadIdx.x;
        int row = blockIdx.y*blockDim.y + threadIdx.y;
```

c είναι η τωρινή γενία , το n αναφέρεται στην επόμενη .

```
//Shared
const int block = 42;
__shared__ int temp[block][block];

int x = threadIdx.y;
int y = threadIdx.x;
```

Δηλώνω μνήμη 42χ42 δηλαδή +10 σε κάθε διάσταση από του μπλοκ.

Οι μεταβλητές χ,y θα μας βοηθήσουν στη διάτρεξη του πίνακα.

Αρχικά γεμίζουμε την shared και ελέγχουμε το χ να ειναι απο 0-31 που σημαίνει με το offset μετά θα είναι από 5-36. Μέσα σε αυτό το διάστημα θα περνάει το κάθε thread την τιμή του.

```
if (x < 32 && y < 32) {
```

```
temp[x + 5][y + 5] = c[row*width + col];
```

Έπειτα έχουμε:

```
//getsma shared
if (x < 5)
{
        if (row >= 5)
        {
                temp[x][y + 5] = c[(row - 5) * width + col];
        }
        else
        {
                temp[x][y + 5] = 0;
        }
        if (y < 5  )
        {
                if ((row >= 5 && col >= 5))
                {
                        temp[x][y] = c[(row - 5) * width + (col - 5)];
                }
                else
                {
                        temp[x][y] =0;
                }

        }
        else if (y >= 27)
        {
                if (row >= 5 && (col + 5) < width)
                {
                        temp[x][y + 10] = c[(row - 5)* width + (col + 5)];
                }
                else {
                        temp[x][y + 10] = 0;
                }

        }

}
else if (y >= 27 )
```

για χ<5 και γραμμές μεγαλύτερες του 5 στον κανονικό πίνακα το κάθε στοιχείο θα φέρει το στοιχείο που είναι 5 γραμμές πάνω από αυτό, αλλιώς αν οι γραμμές είναι μικρότερες του 5 θέσε 0 στην shared memory.

Αν το y <5 τότε αφού και χ<5 φέρε τα διαγώνια πάνω αριστερά. Το κάθε στοιχείο φέρνει το 5ο πάνω αριστερά διαγώνιο του αλλιώς αν δεν ισχύει η y<5 μηδένισε την shared.

Για y>=27 και χ<5 θέλουμε τα πάνω δεξία διαγώνια. Το κάθε στοιχείο φέρνει το 5ο πάνω δεξιά διαγώνιο του.Αν δεν υπάρχει η θέση της κοινής μνήμης πάλι εκεί μηδενίζεται. Υπενθυμίζουμε το y παίρνει τιμές τώρα απο 27-31 άρα γιαυτό βάζουμε το + 10 για να θέτω στην shared στις στήλες 37-41 δηλαδή τις 5 τελευταίες.

```
else if (x >= 27 )
{
        if ((row + 5) < width)
        {
                temp[x + 10][y + 5] = c[(row + 5) * width + col];
        }
        else {
                temp[x + 10][y + 5] = 0;
        }
        if (y < 5)
        {
                if (col >= 5 && row + 5 < width)
                {
                        temp[x + 10][y] = c[(row + 5) * width + (col - 5)];
                }
                else {
                        temp[x + 10][y] = 0;
                }

        }
        else if (y >= 27)
        {
                if (row + 5 < width && (col + 5) < width)
                {
                        temp[x + 10][y + 10] = c[(row + 5) * width + (col + 5)];
                }
                else {
                        temp[x + 10][y + 10] = 0;
                }

        }


}
```

Τώρα αν το χ είναι μεγαλύτερο του 27 δηλαδή ανοίκει στις τελευταίες 5 γραμμές.

Τσεκάρω αν το row+5 <width ισχύει για να μη βγούμε εκτός ορίων και το κάθε στοιχείο απαυτές τις γραμμές φέρνει ΑΝ έχει τα 5 κάτω του αλλιώς αν δεν έχει βάζουμε 0 στην κοινή μνήμη.

Έπειτα φέρνω τα κάτω αριστερά διαγώνια και μετά τα κάτω δεξιά διαγώνια όπως πριν.

Τέλος φέρνω τα αριστερά και τα δεξία όπως φαίνεται παρακάτω για την κοινή μνήμη.

```
        }

        if (y < 5 )
        {
                if (col >= 5) {
                        temp[x + 5][y] = c[row*width + (col - 5)];
                }
                else {
                        temp[x + 5][y] = 0;
                }
        }
        else if (y >= 27 )
        {
                if (col + 5 < width)
                {
                        temp[x + 5][y + 10] = c[row*width + (col + 5)];
                }
                else {
                        temp[x + 5][y + 10] = 0;
                }
        }
```

Περιμένω να τελειώσουν όλα τα threads.

```
__syncthreads();
```

```
                __syncthreads();
                int alive_n = 0;
                int posx = x + 5;
                int posy = y + 5;
                for (int i = posx - 5; i <= posx + 5; i++)
                {
                        for (int j = posy - 5; j <= posy + 5; j++)
                        {
                                if (i >= 0 && i < block && j >= 0 && j < block && !(i == posx && j == posy))
                                {
                                        alive_n += temp[i][j];
                                }
                        }
                }
                if (alive_n >= 34 && alive_n <= 58)
                {
                        if (alive_n <= 45 && temp[posx][posy] == 0)
                        {
                                n[row*width + col] = 1;
                                atomicAdd(alive, 1);
                        }
                }
                else
                {
                        if (temp[posx][posy] == 1)
                        {
                                n[row*width + col] = 0;
                                atomicSub(alive, 1);
                        }
                }
        }
}
```

Και τώρα μέσω της κοινής μνήμης βρίσκω τους γείτονες και υπολογίζω τις τιμές που χρειαζόμαστε όπως και στους άλλους κώδικες.

ΑΠΟΤΕΛΕΣΜΑΤΑ:

Όλοι οι κώδικες δίνουν τα σωστά αποτελέσματα. Η cpu είναι πιο αργή από τη gpu για μεγάλο μέγεθος πίνακα.

Όμως δεν καταφέραμε την gpu με shared memory πιο γρήγορη από την gpu χωρίς shared.