Simulacija fluida pomoću „*Lattice*“ modela

Ivan Dević

Mentor: doc. dr.sc. Dejan Vinković

ZAVRŠNI RAD

Split, listopad 2011.

Odjel za fiziku

Prirodoslovno-matematički fakultet

Sveučilište u Splitu

Sažetak

U ovom radu prikazana je numerička metoda rješavanja Navier-Stokesove jednadžbe za fluide vremeneski neovisne gustoće pomoću metode koja ne koristi diferencijalne jednadžbe, nego tzv. „*Lattice*“ model. „*Lattice*“ model definira rešetku po kojoj se čestice fluida mogu gibati, dok se interakcije između čestica svode na niz jednostavnih pravila. Takav pristup je vrlo pogodan za numeričko rješavanje.

Sadržaj

Popis slika

Popis tablica

1 Uvod

1.1 Fluidi

Fluidi su vrsta tvari koje se deformiraju i pod najmanjim utjecajem vanjskog djelovanja. S obzirom na razinu deformacija na vanjsko djelovanje, dijelimo ih na plinove i tekućine. Plinovi se pod vanjskim utjecajem izrazito deformiraju, dok se tekućine toliko malo deformiraju, da ih smatramo nestlačivima. Ovakva aproksimacija je izrazito praktična iz razloga što bi inače bilo potrebno uvoditi statističke i termodinamičke proračune pri opisivanju stanja tekućine u mirovanju i kretanju. Aproksimacija nestlačivosti kod plinova je jedino moguća ukoliko je promjena tlaka u sustavu izrazito mala.

1.2 Navier-Stokesova jednadžba

Navier-Stokesova jednadžba, nazvana po Claude-Louis Navieru i George Gabriel Stokesu, skupa s jednadžbom kontinuiteta i drugim Newtonovim zakonom opisuje kretanje čestica unutar fluida.

Jednažba kontinuiteta glasi:

$$\frac{\partial \rho}{\partial t} + \nabla(\rho u) = 0 \tag{1.1}$$

gdje ρ predstavlja gustoću fluida, dok u predstavlja brzinu fluida, koja se za slučaj vremenski neovisne gustoće pretvara u:

$$\nabla u = 0 \tag{1.2}$$

koja govori da količina fluida koja uđe u infinitezimalno mali prostor je jednaka količini fluida koja izađe iz tog prostora.

Za slučaj fluida s vremenski neovisnom gustoćom, Navier-Stokesova jednadžba glasi:

$$\rho\left(\frac{\partial u}{\partial t} + u\nabla u\right) = -\nabla\rho + \eta\nabla^2 u + F \tag{1.3}$$

gdje ρ i u opet predstavljaju gustoću i brzinu fluida, η predstavlja koeficijent viskoznosti fluida, a F predstavlja ukupnu vanjsku silu koja djeluje na fluid.

Rješavanje Navier-Stokesove jednadžbe u većini slučajeva je analitički nemoguće, te se zbog te činjenice razvila potreba za adekvatnim računalnim rješavanjem ovog problema.

1.3 *Cellular automata*

Cellular automata je model rješavanja problema koji su prvi promatrali John von Neumann i Stanislaw Ulam. Karakteristike ovog modela su da prostor i vrijeme nemaju kontinuirani skup vrijednosti, već imaju diskretni skup vrijednosti, te niz precizno utvrđenih pravila između dva vremenska koraka tj. model sam sebe ažurira u određenom trenutku pomoću zadanih pravila i podataka koje se iščitavaju iz prethodnog trenutka.

Prikažimo rad *Cellular automate* na primjeru „Živih susjeda“. Kao što smo već napomenuli, prostor i vrijeme su diskretizirani, te preptostavimo da određena točka u prostoru, u određenom vremensku trenutku može imati jedno od dva stanja: *živo* (1) i *mrtvo* (0). Moramo još definirati pravila ovog modela. Definirajmo da je točka u prostoru *živa* samo ako je u prethodnom trenutku samo jedan od njezinih susjeda bio živ. U jednodimenzionalnom prostoru matematički zapis ovog pravila glasi:

$$a_i(t) = (a_{i-1}(t-1) + a_{i+1}(t-1)) \bmod 2 \tag{1.4}$$

gdje nam *a* predstavlja stanje točke kordinate *i* u trenutku *t*, dok je *mod* matematička operacija koja nam vraća ostatak dijeljenja dva cijela broja, koju smo morali uvesti, jer bi nam inače vrijdnost *a* mogla poprimiti vrijednost 2, što nam se kosi s osnovnim pretpostavkama modela..

Tablica 1 *Prikaz modela "Živih susjeda", gdje su a[1] i a[10] također susjedi*

Korak	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
0	0	0	1	1	0	0	1	0	0	1
1	1	1	1	1	1	1	0	1	1	0
2	1	0	0	0	0	1	0	1	1	1
3	1	1	0	0	1	0	0	1	0	1
4	0	1	1	1	0	1	1	0	0	1
5	0	1	0	1	0	1	1	1	1	0

1.4 Metoda Boltzmannove rešetke

Metoda Boltzmannove rešetke (eng. *Lattice Boltzmann Method, LBM*) je opisana i predstavljena 1998. godine. Ona se za razliku od drugih metoda ne zasniva direktno na Navier-Stokesovoj jednadžbi, već se predočava modelom mikroskopskih čestica koje putuju

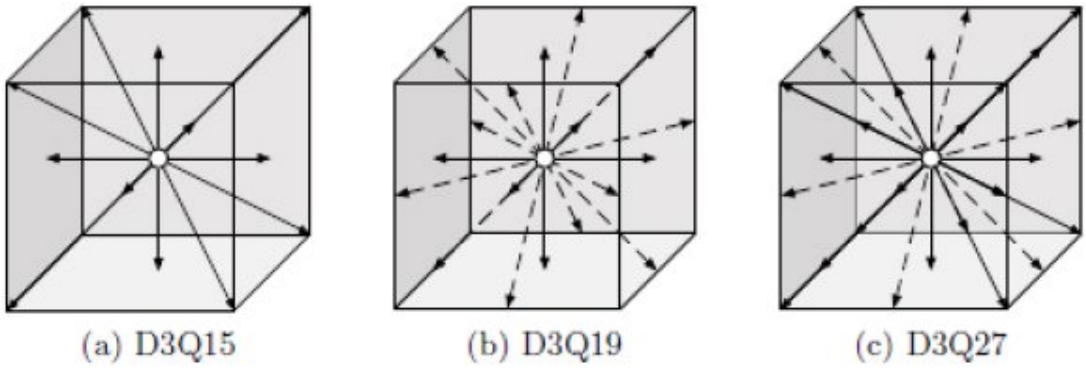
po rešetki (stoga se i jednadžbe kojima su opisane nazivaju jednadžbe Boltzmannove rešetke, eng. *Lattice Boltzmann Equations, LBE*) čija prosječna vrijednost daje makroskopska svojstva koja zadovoljavaju Navier-Stokesove jednadžbe. Izveden je i dokaz kojim se *LBE* svodi na Navier-Stokesovu jednadžbu 1.3. Kako je *LBM* moderna metoda ima mnogo prednosti prethodnih metoda s vrlo malo nedostataka koje su prethodne metode imale. Prednosti su:

- Vrlo jednostavan algoritam za implementaciju i razumijevanje koji je u svojoj prirodi paralelan te ga je moguće prilagoditi za rad na GPU (eng. *Graphics Processing Unit,* grafički procesor), što će biti cilj ove radnje
- Nema problema sa složenim i nepravilnim granicama te miješanjem fluida
- Glavna formula je jednostavna i brzo se izračunava
- Može lako simulirati makroskopska svojstva fluida , te neka mikroskopska svojstva

Glavni nedostatak ovog modela je taj što je rezultat statističke prirode, te bi se na mikroskopskoj razini ovog modela primjetilo dosta nestabilnih rezultata.

LBM se razvio iz *LGCA* (eng. *Lattice-Gas Cellular Automata*), statističkog modela koji je simulirao plin pomoću čestica koje su se nalazile na diskretnim pozicijama u pravilnoj kvadratnoj ili šesterokutnoj mreži. Čestice plina su bile predstavljene Booleovim vrijednostima (na određenom mjestu u mreži čestica ili ima ili nema određeno svojstvo, kao što je npr. masa, ako nema mase, čestica ne postoji na tom mjestu), a te vrijednosti su se izračunavale i rasprostirale po mreži u određenim vremenskim intervalima i fiksnim smjerovima. *LBM* modeli se klasificiraju kao *DdQq*, gdje *d* predstavlja broj dimenzija u kojem se odvija simulacija, dok *q* predstavlja broj smjerova u kojem se čestice mogu kretati. Model D2Q6 se smatra najoptimalnijim modelom za implementirati iz razloga što na velikim dimenzijama rešetke jako precizno rješava Navier-Stokesovu jednadžbu, dok se broj pravila koji treba definirati na modelima D2Q7, D2Q8 povećava eksponencijalno, te ćemo u daljnjem

tijeku rada opisati implementaciju D2Q6 modela, analizirajući sve elemente ovog modela.



Slika 1 *Primjeri trodimenzionalnih Lattice Boltzmann Method rešetki [5]*

1.5 GPU i CUDA

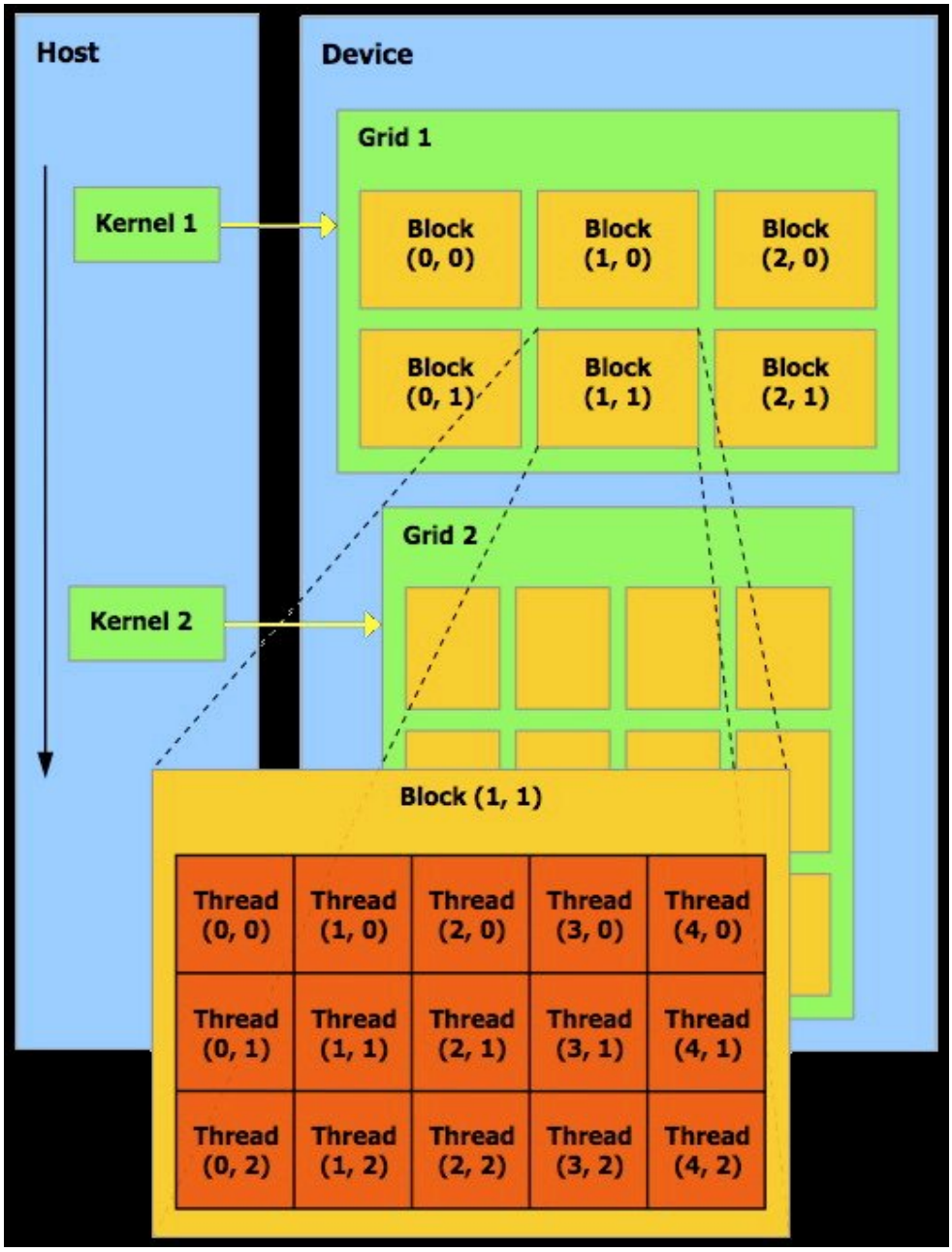
Razvojem znanosti potrebe da se kompliciranji računi provode brzo i točno su bile sve izražajnije. Mnogi se fizičarski problemi simuliraju na računalima, a za simuliranje kompleksnih fizičarskih problema smo trebali razviti nove tehnike rješavanja. Srećom imamo podršku i proizvođača *hardwarea*, koji iz dana u dan izbacuju naprednije modele *hardwarea* za naše potrebe. NVIDIA je nedavno predstavila CUDA-u (eng. *Compute Unified Device Architecture*), programerski model koji nam omogućuje iskorištavanje potencijala koje u sebi imaju GPU-evi zbog sposobnosti paralelnog vršenja programa. Ta sposobnost izlazi iz same arhitekture GPU-a, gdje se većina tranzistora upotrebljava za izvršavanje određenih operacija, dok se na CPU-evima većina tranzistora koristi za skupljanje podataka i kontrolu toka izvršavanja programa stoga ne možemo nikako isključiti CPU iz simulacija, jer se na njemu kontrolira tok programa i prikupljanje podataka koji se pritom šalju na GPU (što su uostalom operacije koje su serijske, a ne paralelene („vremenski ovisne“)).

Za shvaćanje rada GPU-a trebamo prvo definirati pojmove *thread*, *block, grid*, *host*, *device* i *kernel*:

- *Thread* je jedinica za radnju na određeno stanje koju želimo da GPU obavi. Program ćemo implementirati tako da svakom *vortexu* pridružimo jedan *thread* i pri translaciji i pri transformaciji;
- *Block* je skup *threadova* u kojem *threadovi* imaju zajedničku memoriju;

- *Grid* je skup *blockova* koji se treba izvršiti prije nego što program pređe na idući korak određenog programa na GPU-u;
- *Host* nazivamo CPU s kojeg šaljemo naredbe GPU-u;
- *Device* nazivamo GPU na kojem se vrši program;
- *Kernel* nazivamo funkcije koje se izvršavaju na GPU-u i CPU-u. Kod definiranja *kernela* pri programiranju trebamo paziti na to da *kernel* ne može vraćati neku vrijednost (broj ili znak), iz GPU-a natrag na CPU, nego se za to koriste posebne nardbe. *Kernel* se obavlja sve dok ne prođe kroz svaki element *grida*. Kernele dijelimo po mjestu (*device* ili *host*) izvršavanja te funkciju, te ih razlikujemo po prefiksima s pomoću kojih ih definiramo:
  - *_host* - funkcija pokrenuta na s CPU-a, te izvršena na CPU-u ;
  - *_device* – funkcija pokrenuta s GPU-a, te izvršena na GPU-u;
  - *_global* –funkcija pokrenuta na CPU-u, te izvršena na GPU-u

Pri implementaciji GPU modela ćemo koristiti *_global kernele* iz razloga što želimo rasteretiti GPU radnji koje se jednako brzo obavljaju na CPU-u, tako oslobađajući GPU za radnje koje donose vremenske dobitke u odnosu na CPU.



Slika 2 *Grafički prikaz rada GPU-a, gdje redoslije izvršavanja kernela određuje CPU (strelica prema dolje) [4]*

2 „*Lattice*“ model

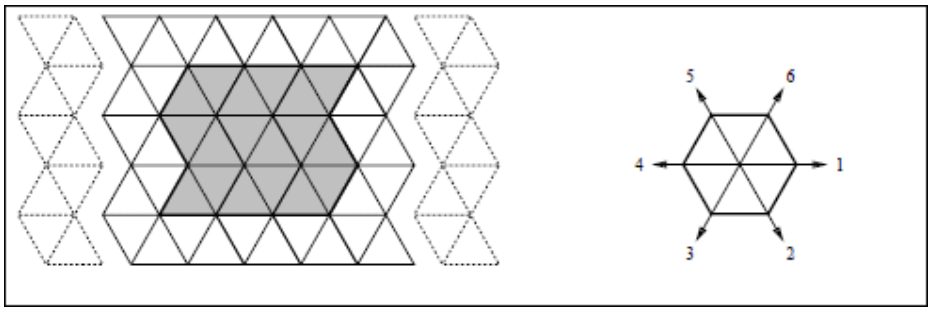
2.1 CPU model

2.1.1 Uvod

Izradu CPU (eng. *Central Processing Unit,* centralni procesor) „*Lattice*“ modela sam podijelio u dva dijela: prvi dio je bio izrada test modela s kojim se testirao dio programa vezan uz ponašanje čestica, koje u ovom programu očitavamo preko vrijednosti *vortexa* (točke na rešetki) tj. testirale su se transformacije *vortexa*, te same kretnje čestica (translacije), dok je drugi dio vezan uz konkretni slučaj kretanja fluida kroz cijev, u kojoj se nalazi prepreka, te gdje pomoću „*Lattice*“ modela možemo provjeravati fenomene koji se događaju pri sudaru fluida i prepreke. S obzirom da se radi o CPU modelu, dimenzije čelije i broj čestica s kojima radimo je manji od dimenzija čelije i broja čestica s kojim bi radili na GPU modelu.

2.1.2 Transformacije i translacije

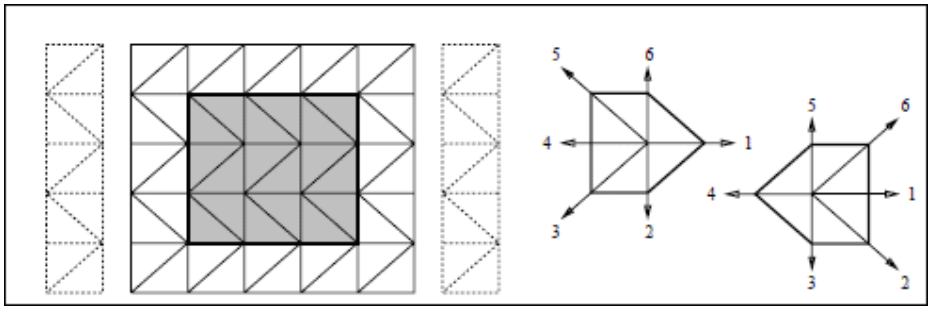
Gibanje fluida ćemo simulirati preko heksagonalne mreže u dvije dimenzije (D2Q6 model), te nas samo zanima stanje na *vorteximа*, te interakcija jednog *vortexa* sa svojih šest susjednih *vortexa*.



Slika 3 *D2Q6 model [1]*

Napomenuli smo da je D2Q6 najoptimalniji model za simulaciju gibanja fluida, no osnovni problem heksagonalne mreže je memorijski zapis. Ukoliko bi slagali memorijske lokacije po izgledu heksagonalne mreže (Slika 3), tada bi naišli na problem memorijskih lokacija, koje su nam u programu nepotrebne (poloviša horizontalnih stranica). Da bi minimalizirali memorijske zahtjeve heksagonalne mreže, pomaknemo svaki drugi redak

heksagonalne mreže prema lijevoj strani, te heksagonalnu mrežu pretvorimo u kvadratnu mrežu, a sustav kretanja čestica iz *vortexa* u *vortex* podijelimo u dva slučaja: za parne i neparne redove.



Slika 4 *Memorijska vizualizacija D2Q6 modela [1]*

Iznos *vortexa* računamo tako da svaki izlaz predstavimo kao jedan bit tj. *vortex* će bit jednak šesteroznamenkastom broju u binarnom sustavu, gdje će svaka od šest znamenki prikazivati stanje određenog izlaza na tom *vortexu* tj. postoji li čestica na tom izlazu, što opisano matematičkom formulom izgleda:

$$a = \sum_{i=1}^{6} a(i) * 2^{i-1} \tag{2.1}$$

gdje nam *a* predstavlja iznos *vortexa*, dok nam a_i (koji ima vrijednosti 0 ili 1) definira postoji li čestica na *i*-tom izlazu iz *vortexa*.

Transformacije *vortexa* su zadane tablično. Primjećujemo da u narednim tablicama, bitni zapis početne vrijednosti *vortexa* su zapisane kao sedmeroznamenkasti binarni brojevi. Sedmi bit služi za definiciju statičnih čestica u nekom *vortexu* koje postoje u D2Q7 modelu, no ne i u D2Q6 modelu. Pomoću osmog bita radimo definicije prepreka i zidova, tj. kad u nekom *vortexu* definiramo zid ili prepreku, tada će njegov osmi bit imati vrijednost 1. U „*Lattice“* modelu se čestice odbijaju od zidova i prepreka po tzv. *„non-slip“* uvjetu, po kojem se smjer impulsa promijeni za 180°stupnjeva nakon sudara. Ukoliko numeriramo smjerove (Smjer_*i*) kao na Slici 2, tada će transformacije *vortexa* na zidu i prepreci izgledati:

- Smjer_1=Smjer_4
- Smjer_2=Smjer_5
- Smjer_3=Smjer_6
- Smjer_4=Smjer_1 (2.2)

- Smjer_5=Smjer_2
- Smjer_6=Smjer_3

Tablica 2 *Transformacije vortexa početnih iznosa 0-31 [1]*

Početni iznos vortexa		Konačni iznos vortexa	Početni iznos vortexa		Konačni iznos vortexa
0	0000000		16	0010000	
1	0000001		17	0010001	
2	0000010		18	0010010	9, 36
3	0000011		19	0010011	37
4	0000100		20	0010100	
5	0000101		21	0010101	42
6	0000110		22	0010110	13
7	0000111		23	0010111	
8	0001000		24	0011000	
9	0001001	18, 36	25	0011001	52
10	0001010		26	0011010	44
11	0001011	38	27	0011011	45, 54
12	0001100		28	0011100	
13	0001101	22	29	0011101	
14	0001110		30	0011110	
15	0001111		31	0011111	

Tablica 3 *Transformacije vortexa početnih iznosa 32-63 [1]*

Početni iznos vortexa		Konačni iznos vortexa	Početni iznos vortexa		Konačni iznos vortexa
 32	0100000		 48	0110000	
 33	0100001		 49	0110001	
 34	0100010		 50	0110010	41
 35	0100011		 51	0110011	
 36	0100100	9, 18	 52	0110100	25
 37	0100101	19	 53	0110101	
 38	0100110	11	 54	0110110	27, 45
 39	0100111		 55	0110111	
 40	0101000		 56	0111000	
 41	0101001	50	 57	0111001	
 42	0101010	21	 58	0111010	
 43	0101011		 59	0111011	
 44	0101100	26	 60	0111100	
 45	0101101	27, 54	 61	0111101	
 46	0101110		 62	0111110	
 47	0101111		 63	0111111	

Primjećujemo da u većini slučajeva početni iznos *vortexa* je jednak konačnom iznosu *vortexa* . Za slučajeve u tablici gdje ne piše konačni iznos *vortexa*, transformacija ne mijenja iznos *vortexa*. Slučajeve gdje početni iznos *vortexa* nije jednak konačnom iznosu *vortexa* dijelimo na dvije skupine:

- Za određeni početni iznos *vortexa* postoji samo jedan mogući konačni iznos *vortexa*
- Za određeni početni iznos *vortexa* postoje dva moguća konačna iznosa *vortexa* tzv. *„dual-kanali“*

Za rješavanje *„dual kanala“* uvodimo generator nasumičnih brojeva koji svedemo na nula ili na jedan, te raspišemo matematičke formule 2.3. Ukoliko definiramo broj *t,* koji može

poprimati samo binarne vrijednosti (rezultat generatora nasumičnih brojeva), te ukoliko je *n* početni iznos *vortexa*, a *m* konačni iznos *vortexa*, tada matematičke formule po kojima rješavamo *„dual-kanale“ m(n)* glase:

$$\begin{aligned} m(9) &= (t+1)*18 \\ m(18) &= 36^t \\ m(27) &= (t+5)*9 \\ m(36) &= t*9 \\ m(45) &= (t+1)*27 \\ m(54) &= ((t+1)*2+1)*9 \end{aligned} \tag{2.3}$$

U slučaju *vortexa* na zidu ili prepreci postupak je malo drugačiji. Naime, već smo rekli da se čestice na zidu odbijaju tako da se vraćaju po smjeru iz kojeg su došle. Tada je prvo nužno zaključiti individualno stanje (smjer kretanja) svake čestice u *vortexu*, te je poslati u suprotan smjer. Taj proces se odvija pomoću relacija 2.2. Uzmimo za primjer relaciju između smjerova 1 i 4. Ukoliko postoji čestica na smjeru 1, njezin numerički doprinos iznosu *vortexa* je 1, dok je za smjer 4 numerički doprinos iznosu *vortexa* 8 (vidi 2.1). Pomoću razlike numeričkih doprinosa smjerova 1 i 4 (koja iznosi 7; za smjerove 2 i 5 iznosi 14; za smjerove 3 i 6 iznosi 28), definiramo transformacije na preprekama i zidovima:

$$m(n) = \sum_{i=1}^{6} a(i) * (n \pm r(i)) \tag{2.4}$$

gdje *m* predstavlja konačni iznos *vortexa*, *n* predstavlja početni iznos *vortexa*, *a(i)* nam pokazuje postoji li čestica na *i*-tom *vortexa* s iznosom *n*, dok nam *r(i)* predstavlja razliku numeričkih doprinosa smjera *i* i njemu suprotnog smjera:

$$r(i) = 7 * 2^{i \bmod 3 - 1} \tag{2.5}$$

U relaciji 2.4 se uzima operator zbrajanja za *i={1,2,3}*, dok se operator oduzimanja uzima za *i={4,5,6}*.

Kao što smo već spomenuli, zbog minimaliziranja memorijskih zahtjeva, heksagonalnu mrežu smo pretvorili u kvadratnu mrežu, te smo zbog toga nužni predvidjeti dva različita slučaja translacija (Slika 4), pa vidimo da imamo *„lijevu translaciju“* gdje su smjerovi 3 i 5 na dijagonalama, te *„desnu translaciju“* gdje su smjerovi 2 i 6 na dijagonalama.

Translacije na zidovima i preprekama nisu ni po čemu drugačije od translacija u prostoru, zato jer se za translacije koristi binarni operatori AND(&), koji uspoređuje bitove dvaju brojeva logičkom operacijom &, koja vraća vrijednost 1 samo ukoliko oba dva broja na istom bitu imaju vrijednost 1, a s obzirom da za translacije promatramo samo vrijednost prvih šest bitova, osmi bit koji predstavlja zapreku nam ne radi nikakvu razliku. Korištenjem binarnog operatora & lako iz iznosa *vortexa* dobivamo vrijednosti *a(i)* koje nam govore postoji li čestica na *i-tom* izlazu u *vortexu* koji ima iznos *n*:

$$a(i) = n \,\&\, 2^i \tag{2.6}$$

Za potrebe translacije je nužno definirati dvije rešetke (matrice), jedna će nam služiti kao početna (A), druga kao završna (B) rešetka nakon svakog koraka. Matricu B uvijek postavimo tako da su joj vrijednosti svih članova jednake nuli, te ih pomoću pravila transformacija i translacija računamo iz vrijednosti članova matrice A.

2.1.3 Testiranje

Program je napravljen tako da se početno stanje upisuje ručno. Prvo se upiše broj *vortexa* koje želimo postaviti na vrijednost različitu od nule, pa nakon toga unosimo kordinate *vortexa* (gornji-lijevi kut je ishodište kordinatnog sustava, varijabla *i* raste prema desno, dok varijabla *j* raste prema dolje). Testni model sadrži mrežu dimenzija 10x10, s tim da unos kordinata *i* i *j* ide od 1 do 8 (na kordinatama 0 i 9 se nalazi zid).

U testiranju se prvo provjerava valjanost translacija, te se potom namještaju takvi početni uvjeti da se u prvom vremenskom koraku dobiju iznosi *vortexa* koji zahtijevaju transformacije.

PRIMJER TESTIRANJA:

Testirajmo radi li transformacija *vortexa* iznosa 9 i u *vortex* iznosa 18 i u *vortex* iznosa 36. Kao što smo napomenuli, početne uvjete ćemo postaviti takve da u prvom vremenskom koraku koji se dogodi se stvore dva *vortexa* iznosa 9, te ćemo provjeriti, stvaraju li se oba preptostavljena rezultata. U testnom modelu rezultat *„dual-kanala“* nije nasumičan, već postavimo da se određeni rezultat *„dual-kanala“* ponavlja svako drugi put, tj. da se u dva uzastopna pojavljivanja *vortexa* čija transformacija dopušta dvije različite vrijednosti, ne ponovi isti rezultat. *Vortex* iznosa 9 tvorimo tako da sudarimo dvije čestice koje se kreću u

horizontalnoj ravnini (smjerovi 1 i 4). S obzirom da testiramo kretanja samo u jednom pravcu, prikazat ćemo to u redu od 10 članova, na čijim su krajevima zidovi.

Tablica 4 *Testiranje vortexa iznosa 9, koji nastaje sudarom čestica suprotnih smjerova (1 i 4), gdje u 2. koraku očekujemo dobiti vortexe iznosa 18 i 36*

1.korak	128	000	001	000	008	000	001	000	008	128
2.korak	128	000	000	018	000	000	000	036	000	128

S ovim smo primjerom pokazali da translacija u smjerovima 1 i 4, te transformacije *vortexa* iznosa 9 ispravno rade. Postupak za provjeravanje drugih elemenata je identičan.

2.1.4 Konačni program

U CPU modelu smo htjeli simulirati tok fluida kroz cijev u kojoj se nalazi prepreka. Sada kad smo testirali svaku komponentu test modela, implementacija konkretnog problema je zahtjevala samo par preinaki:

- Početne uvjete smo u test modelu unosili ručno, dok se sada početno stanje postavi tako da na lijevom i desnom rubu ne postoji zid (gornji i donji zidovi su i dalje u programu), stavimo prepreku na sredinu cijevi (definiramo tri *vortexa* kao zid), te definiramo da svaki *vortex* ima vrijednost 1, voda u cijevi ide prema desno
- Mreža je povećana s dimenzija 10x10 na 15x15
- Jedini unos koji sada vršimo u program jest koliko vremenskih koraka želimo da nam program obavi
- Pri ispisu ne ispisujemo prvi i zadnji stupac. Prvi stupac postavimo kao „izvor“, te ga na kraju svakog ažuriranja postavimo na 1 (konstantan tok) izuzev *vortexa* koji pripadaju zidovima. Zadnji stupac postavimo kao „odvod“, te ga na kraju svakog ažuriranja postavimo na 0 bez iznimke jer ne želimo da nam „odvod“ ima ikakav utjecaj na cjelokupnu mrežu, a kad se *vortexi* koji pripadaju zidovima u zadnjem stupcu ne bi postavili na 0 zbog *non-slip* uvjeta refleksije bi bio moguć utjecaj na ostatak rešetke

Tablica 5 *Početno stanje CPU modela*

128	128	128	128	128	128	128	128	128	128	128	128	128
001	001	001	001	001	001	001	001	001	001	001	001	001
001	001	001	001	001	001	001	001	001	001	001	001	001
001	001	001	001	001	001	001	001	001	001	001	001	001
001	001	001	001	001	001	001	001	001	001	001	001	001
001	001	001	001	001	001	001	001	001	001	001	001	001
001	001	001	001	001	001	001	128	001	001	001	001	001
001	001	001	001	001	001	001	128	001	001	001	001	001
001	001	001	001	001	001	001	128	001	001	001	001	001
001	001	001	001	001	001	001	001	001	001	001	001	001
001	001	001	001	001	001	001	001	001	001	001	001	001
001	001	001	001	001	001	001	001	001	001	001	001	001
001	001	001	001	001	001	001	001	001	001	001	001	001
001	001	001	001	001	001	001	001	001	001	001	001	001
128	128	128	128	128	128	128	128	128	128	128	128	128

Tablica 6 *Stanje CPU modela nakon 500 vremenskih koraka*

128	130	128	128	130	128	134	128	128	132	128	128	132
001	003	001	035	033	001	019	019	001	001	001	019	001
001	001	001	039	003	003	001	033	001	003	001	033	033
017	042	049	001	017	033	001	039	003	001	000	032	003
017	001	001	001	017	038	037	035	003	001	003	000	001
001	001	001	005	035	049	001	001	003	003	035	001	001
001	042	001	017	000	033	008	128	000	000	000	000	000
001	017	001	001	001	036	017	128	032	002	000	002	032
017	017	033	000	000	003	032	136	032	034	002	000	000
050	003	001	017	005	003	004	001	001	000	035	001	001
001	000	033	007	001	002	017	034	001	000	001	001	001
001	003	005	017	020	035	034	033	001	033	001	003	001
001	033	033	017	041	037	018	003	002	001	001	000	001
001	001	023	043	003	016	023	001	003	033	001	001	001
128	128	128	144	144	128	128	128	128	128	128	128	128

2.2 GPU model

2.2.1 Uvod

Dimenzije rešetke koje smo koristili pri izradi CPU modela nisu dovoljne za dobiti makroskopske veličine koje će zadovoljavati Navier-Stokesovu jednadžbu. No sada dolazimo do osnovnog problema CPU-a za rješavanje ovog problema. Zbog arhitekture CPU-a, koji sve operacije provodi serijski (jednu po jednu), okrećemo se GPU-u, koji operacije provodi paralelno (više njih istovremeno). Kao što smo napomenuli priroda „*Lattice*“ modela je paralelna: transformacije određenog *vortexa* ovise samo o početnom iznosu tog istog *vortexa*, dok translacije vršimo tako da se završna matrica B popunjava iz podataka početne matrice A, te nije bitno kojim redom analizirmo podatke iz matrice A. Način rada transformacija i translacija je zbog toga itekako pogodan za paralelno izvšravanje.

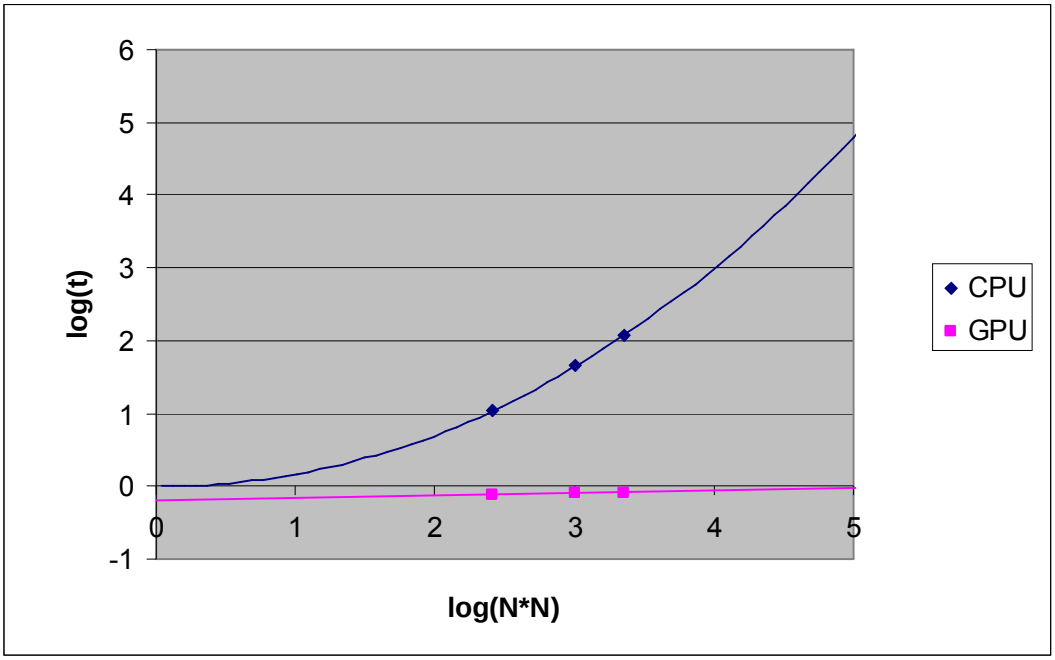
2.2.2 Razlike između GPU i CPU modela

U poglavlju 2.1 *CPU model* smo opisali sve elemente CPU „*Lattice“* modela, te logiku i način na kojeg smo ga implementirali. GPU model zahtijeva određene preinake da bi bio uspješno implementiran. Za početak smo transformacije definirali preko *tablice transformacija*, gdje smo ispisali sve transformacije iz Tablica 1 i 2, tj. pridružili smo konačne iznose *vortexa* početnim iznosima, dok smo na CPU modelu to implementirali preko logičkog operatora *ako* (eng. *if*). Translacije su zahtjevale da ih izmjenimo zbog načina na koji se kodira u CUDA-i. Dok smo na CPU modelu rešetku pamtili kao matricu, na GPU se samo zapamti memorijska lokacija prvog člana rešetke, te pomicanjem po memorijskim lokacijama prelazimo preko rešetke, tj. matricu smo pretvorili u niz (dvodimenzionalno u jednodimenzionalno polje), tako da se konačna matrica B ne popunjava pomoću mjesta nekog *vortexa* u početnoj matrici A, već po memorijskoj lokaciji tog *vortexa* u početnoj matrici A, koja se u CUDA-i definiraju preko *threadova* i *blockova*.

Sad ćemo razmatrati vrijeme potrebno da se simulacija izvrši na CPU-u, te na GPU-u, da potvrdimo pretpostavljenu potrebu da sa CPU-a simulaciju prebacimo na GPU. Pri ispitavanju potrebnog vremena za obavljanje simulacije ćemo postaviti vremenski korak od 50000 koraka, dok ćemo mijenjati dimenzije rešetke: 16x16; 32x32; 48x48.

Tablica 7 *Vrijeme obavaljanje simulacije ovisno o veličini rešetke (N={16,32,48}) CPU i GPU modela. U prvom redu se nalazi broj vortexa, u drugom redu vrijeme obavljanja simulacije na CPU-u, dok u zadnjem redu vrijeme obavljanja simulacije na GPU-u. Oba vremena su izražena u sekundama*

N*N	256	1024	2304
CPU [s]	10,719	45,172	118,89
GPU [s]	0,76	0,79	0,82



Slika 5 *Grafički prikaz logaritma potrebnog vremena za obavljanje simulacije GPU i CPU modela, ovisno o logaritmu (N*N), gdje je N={16,32,48}, a predstavlja broj elemenata u jednom retku ili stupcu kvadratne matrice, pomoću koje smo definirali rešetku*

Primjećujemo da razlika ova dva vremena raste eksponencijalno s dimenzijama rešetke, te zaključujemo da je nužno vršiti ovu simulaciju na GPU-u. Razloge ovolike vremenske razlike obavljanja simulacije GPU i CPU modela smo već naveli: algoritam „*Lattice*“ model je po svojoj prirodi paralelan, a paralelno izvršavanje simulacije je moguće na GPU-u zbog njegove arhitekture.

Također je potrebno definirati suradnju GPU-a i CPU-a (*device* i *host*), jer je CPU i dalje procesor koji kontrolira tijek simulacije, te memorijske transakcije između *hosta* i *devicea*.

2.2.3 Tijek simulacije

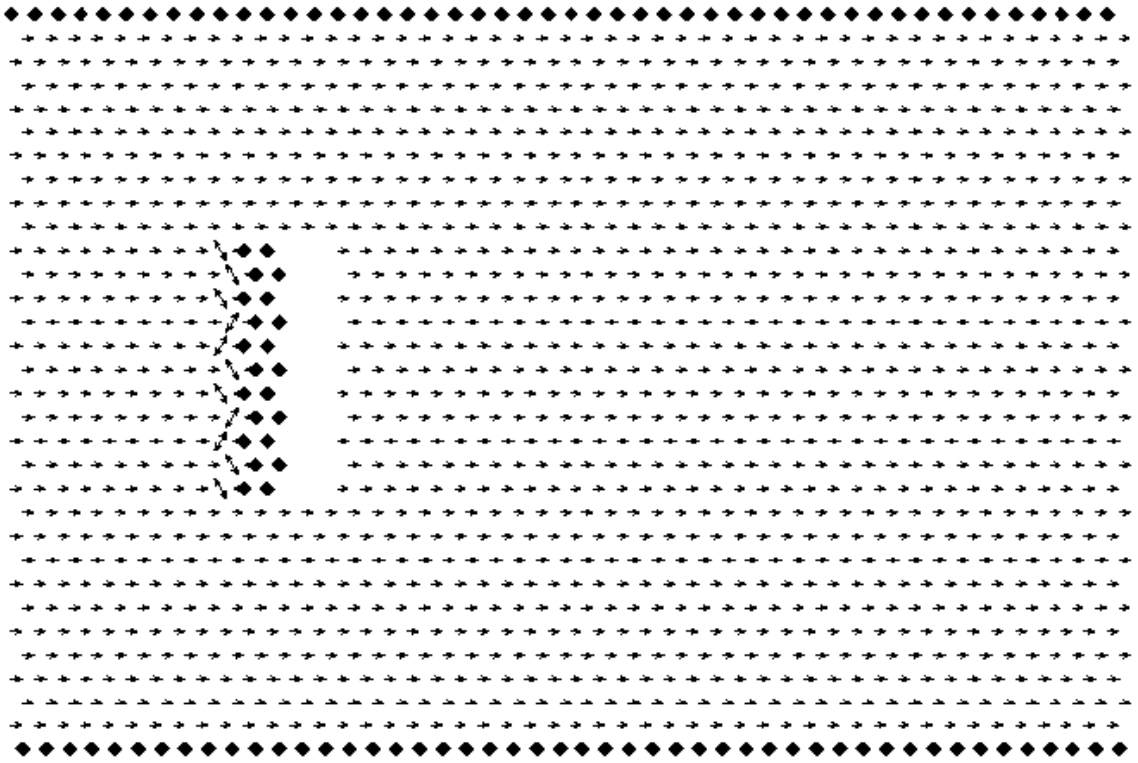
Nakon što smo se upoznali s načinom rada GPU-a, te kakve će to preinake u odnosu na CPU model zahtijevati od nas, napokon možemo opisati tok simulacije na GPU modelu. Kao i kod CPU modela, na GPU modelu također simuliramo tok fluida kroz cijev u kojoj se nalazi prepreka. Tok simulacije je sljedeći:

- Definiranje početnih uvjeta: veličina rešetke, zidovi, prepreke, početni tok koji podrazumjeva i definiranje stupca „izvora“ i broj koraka;
- Nakon što smo definirali veličinu rešetke, rezerviramo memorijske lokacije na *hostu* i *devicu* za rešetke (na *hostu* definiramo samo jednu matricu, dok na *deviceu* dvije);
- Definiramo *kernele* transforamcija i translacija, te ih pozivamo u parovima, tj. u jednom koraku dvaputa pozovemo *kernel* transformacije, te dvaput pozovemo *kernel* translacije, jer s ovakvim tokom elimiramo potrebu za kontrolnom varijablom. Nek imamo dvije matrice A i B, te smo u A definirali početne podatke. Tada radimo sljedeće: A translatiramo u B, transformiramo B, B translatiramo u A, te transformiramo A. Na ovaj način će nam matrica A na početku svakog koraka biti početna, te što je najbitnije u prethodnom koraku posljednja ažurirana (kontinuirani tok simulacije).
- Nakon što se obavi zadani broj koraka CPU preuzme podatke matrice B iz posljednjeg koraka, te ispiše.
- S obzirom na brzinu obavljanja simulacije na GPU-u, moguće je napraviti slijed slika, gdje će svaka slika reprezentirati svako drugi korak u simulaciji (zbog dvostruke uporabe *kernela* transformacija i translacija). Za potrebe ovakve simulacije nećemo ispisivati iznose *vortexa* na ekran, već ćemo ih grafički prikazati kao što su prikazani u Tablicama 1 i 2, dok ćemo zidove i prepreke prikazati crnim rombovima.

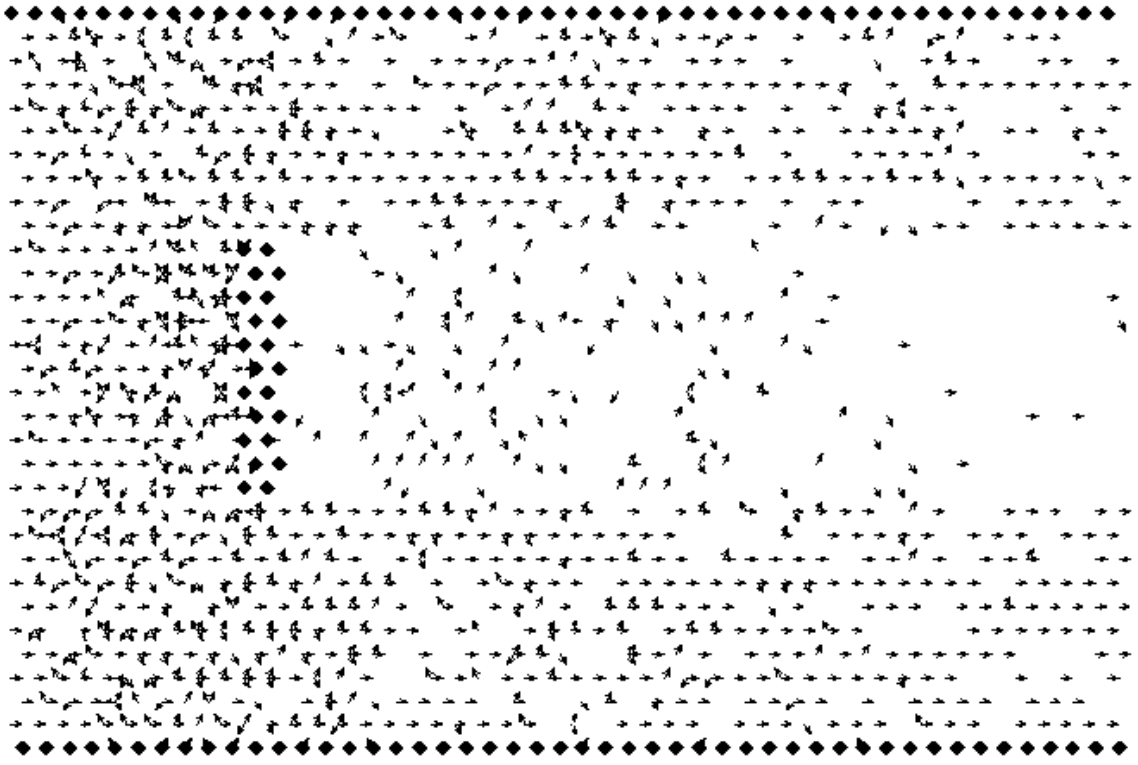
Ukoliko je „*Lattice*“ modelom uistinu moguće simulirati fluid, tada očekujemo da na grafičkom prikazu simulacije dobijemo sličnosti u ponašanju fluida u sudaru s nekom preprekom:

- Fluid će imati manju gustoću u dijelu prostora koji se nalazi nakon prepreke u odnosu na tok fluida, te da će imati veću gustoću u dijelu prostora prije prepreke u odnosu na tok fluida
- Vrtloženje toka

Slika 6 *Primjer fluida u sudaru s preprekom [6]*



Slika 7 *Grafički prikaz rešetke nakon 2 koraka*



Slika 8 *Grafički prikaz rešetke nakon 100 koraka*

3 Zaključak

Cilj ove radnje je prikazati kako se fenomene koje opažamo u prirodi, a opisujemo kompliciranim matematičkim jednadžbama, mogu na izrazito lake načine računalno riješiti. Na grafičkom prikazu (Slike 7 i 8) uistinu opažamo neke stvari koje se intuitivno zaključuju, te koje se mogu svakodnevno primjetiti kada promatramo neki fluid u sudaru s nekom preprekom: fluid je gušći u području prije prepreke u odnosu na vlastiti tok, nego što je to slučaj u području koje slijedi nakon prepreke u odnosu na tok fluida. Područje iza prepreke se također postupno upotpunjava fluidom, te fluid stvara vrtloge pri popunjavanju tog prostora.

No rješenja koja smo prikazali grafički nisu dovoljno točna kako bi makroskopske veličine koje se mogu računati pomoću ovih rješenja zadovoljavale Navier-Stokesovu jednadžbu. Razlog tome je što je osnovna preptostavka *Cellular automate* da vrijeme i prostor podijelimo u diskretni spektar, s tim da razmak između dvije diskretne vrijednosti prostora i vremena bude toliko sitan, da takav spektar možemo smatrati kontinuiranim spektrom. Dimenzije gore prikazanog GPU modela ipak nisu tolike. Za potpuni model je potrebno napraviti rešetku s puno većim dimenzijama, te grafički prikaz bi se vršio tako da prostor podijelimo na regije, te grafički prikažemo vektor prosječnog impulsa svih čestica u određenoj regiji. Pri simulacijama rešetke velikih dimenzija simulacija se treba izvršavati sve dok se ne pojave rješenja koja su vremenski neovisna iz razloga što je jedna od osnovnih pretpostavki jednadžbe (1.3) da je gustoća vremenski nepromjenjiva, te samo ovisi o prostornim varijablama.

„Lattice“ model je samo jedan od mnogih modela koji su nastali od *cellular automate.* Mnogi problemi u fizici i ostalim znanostima se modeliraju pomoću *cellular automate*, jer je simulacija bitno brža ukoliko izbjegnemo numeričko rješavanje diferencijalnih jednadžbi. Istraživanje modela poput „*Lattice“* modela je u znanosti relativno novo (uzlet tek 1990-ih godina), a rezultati koji se postižu modelima nastalih od *cellular automate* su precizni koliko i rezulati dobiveni numeričkim rješavanjem diferencijalnih jednadžbi, te su svi pogodni za paralelno izvršavanje, pa možemo iskoristiti cijeli potencijal koji u sebi imaju GPU-evi.

Upotreba GPU-eva je bitna trenutno znanosti. Dok su CPU-evi došli do granice svojih performansi (cca. 3GHz), te se tehnologija okrenila višejezgrenim procesorima, koji iako imaju mogućnost izvršavanja paralelnih programa, GPU-evo izvršavanje paralelnog programa

u odnosu na višejezgrene CPU-eve je i dalje efikasnije, a matematičko rješavanje raznih fenomena iz prirode je iz dana u dan sve kompliciranije, te su CPU-evi postali neupotrebljivi za ovakve proračune naprema snazi GPU-eva.

Budućnost računalnog rješavanja fizikalnih problema će se definitivno oslanjati na osnove koje su dane *cellular automatom* zbog jednostavnog načina implementacije jednom kad se utvrde pravila i paralelnosti takvih modela. Jednostavnost implementacije je što se programski kod ne treba promatrati fizičarskom intepretacijom, već striktno čistom logikom izvršavanja zadanih pravila. To uvelike pojednostavljuje programersku fazu, naročito pri numeričkom rješavanju diferencijalnih jednadžbi, gdje je svodimo na elementarnu logiku, a ne na matematičku logiku i inerpretaciju, te rezultati koji se dobivaju su dovoljno precizni.

4 Literatura

[1] Brian J N Wylie „Application of Two-Dimensional Cellular Automaton Lattice-Gas Models to the Simulation of Hydrodynamics“, University of Edinburgh, 1990

[2] CUDA, Rocky Mountain Mathematics Consortium http://www.caam.rice.edu/~timwar/RMMC/CUDA.html

[3] NVIDIA CUDA Compute Unified Device Architecture,Programming guide, Version 1.0, 2007

[4] http://www.evl.uic.edu/aej/525/pics/cuda_threads.jpg

[5] Mario Volarević „Animacija modela vodenih površina“, Sveučilište u Zagrebu, 2011

[6] Mohamed Fayed, Rocco Portaro, Amy-Lee Gunter, Hamid Ait Abderrahmane, and Hoi Dick Ng „Visualization of flow patterns past various objects in two-dimensional flow using soap film“, Department of Mechanical and Industrial Engineering, Concordia University, Montréal, Québec H3G 1M8, Canada

Dodatak 1 – CPU model

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <assert.h>
#include <cuda.h>
#define DIRECTION_1     1
#define DIRECTION_2     2
#define DIRECTION_3     4
#define DIRECTION_4     8
#define DIRECTION_5     16
#define DIRECTION_6     32
#define STATIC_PARTICLE 64
#define WALL            128
#include <sys/time.h>
#include <iostream>
#include <iomanip>

int br=0;
int transform (int n)
{
   int m;
   if (n<WALL)
   {
   switch (n)
   {
        case 9: m=(br++%2+1)*18;
        break;
        case 11: m=38;
        break;
```

```
case 13: m=22;
break;
case 18: m=9*int(pow(4,br++%2));
break;
case 19: m=37;
break;
case 21: m=42;
break;
case 22: m=13;
break;
case 25: m=52;
break;
case 26: m=44;
break;
case 27: m=(br++%2+5)*9;
break;
case 36: m=(br++%2+1)*9;
break;
case 37: m=19;
break;
case 38: m=11;
break;
case 41: m=50;
break;
case 42: m=21;
break;
case 44: m=26;
break;
case 45: m=(br++%2+1)*27;;
break;
case 50: m=41;
break;
case 52: m=25;
```

```
        break;
        case 54: m=((br++%2+1)*2+1)*9;
        break;
        default: m=n;
        break;
    }
    }
    else
    {
      n-=WALL;
      m=n;
      if ((n&DIRECTION_1)>0)
        m+=7;
      if ((n&DIRECTION_2)>0)
        m+=14;
      if ((n&DIRECTION_3)>0)
        m+=28;
      if ((n&DIRECTION_4)>0)
        m-=7;
      if ((n&DIRECTION_5)>0)
        m-=14;
      if ((n&DIRECTION_6)>0)
        m-=28;
      m+=128;
    }

    return m;
}



int main (void)
{
```

```
int ***a, i, j, A, B, n, k;
FILE *p;
p=fopen("lattice.txt","w");
a=(int ***)malloc(sizeof(int **)*2);
a[0]=(int **)malloc(sizeof(int *)*15);
a[1]=(int **)malloc(sizeof(int *)*15);
for(i=0;i<15;i++)
{
        a[0][i]=(int*)malloc(sizeof(int)*15);
        a[1][i]=(int*)malloc(sizeof(int)*15);
}
printf("\nBroj koraka? ");
scanf("%d", &n);
clock_t start = clock();
for(j=0;j<15;j++)
{
        for(i=0;i<15;i++)
        {
                if (j==0 || j==(15-1))
                        a[0][i][j]=128;
                else if (i==8 && j>5 && j<9)
                        a[0][i][j]=128;
                else
                        a[0][i][j]=1;
        }
}

for(j=0;j<15;j++)
{
        fprintf(p, "\n");
        for(i=1;i<(15-1);i++)
                fprintf(p, " %.3d ", a[0][i][j]);
```

```
}
for (k=1;k<=n;k++)
{
   fprintf(p, "\n%d\n", k);
   if (k%2>0)
     A=0;
   else
     A=1;
   B=1-A;
   for(i=0;i<15;i++)
            for(j=0;j<15;j++)
            {
              if (j==0 || j==(15-1))
                        a[B][i][j]=128;
              else if (i==8 && j>5 && j<9)
                        a[B][i][j]=128;
              else
                        a[B][i][j]=0;
              if (i==0 && j!=0 && j!=(15-1))
                        a[A][i][j]=1;
              else if (i==(15-1))
                        a[B][i][j]=0;
              }
   for(j=0;j<15;j++)
            for(i=0;i<(15-1);i++)
            {
                     if ((a[A][i][j]&DIRECTION_1)>0)
                              a[B][i+1][j]+=DIRECTION_1;
                     if ((a[A][i][j]&DIRECTION_2)>0)
                       {
                              if (j%2==0)
                                a[B][i][j+1]+=DIRECTION_2;
                              else
```

```
                    a[B][i+1][j+1]+=DIRECTION_2;
                }
                if ((a[A][i][j]&DIRECTION_3)>0)
                {
                    if (j%2==0)
                        a[B][i-1][j+1]+=DIRECTION_3;
                    else
                        a[B][i][j+1]+=DIRECTION_3;
                }
                if ((a[A][i][j]&DIRECTION_4)>0)
                    a[B][i-1][j]+=DIRECTION_4;
                if ((a[A][i][j]&DIRECTION_5)>0)
                {
                    if (j%2==0)
                        a[B][i-1][j-1]+=DIRECTION_5;
                    else
                        a[B][i][j-1]+=DIRECTION_5;
                }
                if ((a[A][i][j]&DIRECTION_6)>0)
                {
                    if (j%2==0)
                        a[B][i][j-1]+=DIRECTION_6;
                    else
                        a[B][i+1][j-1]+=DIRECTION_6;
                }

            }

    for(j=0;j<15;j++)
    {
        fprintf(p, "\n");
        for(i=0;i<15;i++)
        {
```

```
                        if (i==0 && j!=0 && j!=(15-1))
                            a[B][i][j]=1;
                        if (i==(15-1))
                            a[B][i][j]=0;
                        a[B][i][j]=transform(a[B][i][j]);
                        if(i!=0 && i!=14)
                        fprintf(p, " %.3d ", a[B][i][j]);
                }
        }


        fprintf(p, "\n\n");

    }
    fprintf(p, "\nKraj");
    printf("\nKraj");
    printf ("\n%f", ( (double)clock() - start ) / CLOCKS_PER_SEC );
    getchar();
    getchar();

    return 0;
}
```

Dodatak 2 – GPU model

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>
#include <math.h>
#include <time.h>

#define BLOCK_SIZE 16  // squared (minimum size = 16x16)
#define GRID_SIZE_X 3
#define GRID_SIZE_Y 2
#define NvolumeX (GRID_SIZE_X*BLOCK_SIZE)
#define NvolumeY (GRID_SIZE_Y*BLOCK_SIZE)
#define UINT32_MAX                      4294967295
#define PI                                      3.14159265358979323846f
clock_t currenttime1,currenttime2;

#define DIRECTION_1     1  // 00000001
#define DIRECTION_2     2  // 00000010
#define DIRECTION_3     4  // 00000100
#define DIRECTION_4     8  // 00001000
#define DIRECTION_5     16 // 00010000
#define DIRECTION_6     32 // 00100000
#define STATIC_PARTICLE 64 // 01000000
#define WALL            128 // 10000000
#define DENSITY         0.9 // density of created particles (fraction of non-empty cells)

////////////////////////////////////////////////////////////////////////////////
////////// DEFINE LOOKUP TABLE ////////////////////////////////////////////////////

#define CODE_FOR_DUAL 192  // output channel shift for finding dual output channels
```

```
#define N_DUAL_CHANNELS 12 // total number of output channels in case of dual channels


unsigned  int Collision_Channels[256] = {
  //              5   6
  //               \ /
  //              4--O--1
  //               / \
  //              3   2
  //
  // bits: ABCDEFGH: A=wall, B=static particle, CDEFGH=directions 654321 (see sketch above)
  //
  // collision output                    // collision input
   0, 1,  2,  3,  4,  5,  6,  7,  8, 192,  //  0  1  2  3  4  5  6  7  8  9
  10, 38, 12, 22, 14, 15, 16, 17, 193, 37, // 10 11 12 13 14 15 16 17 18 19
  20, 42, 13, 23, 24, 52, 44, 194, 28, 29, // 20 21 22 23 24 25 26 27 28 29
  30, 31, 32, 33, 34, 35, 195, 19, 11, 39, // 30 31 32 33 34 35 36 37 38 39
  40, 50, 21, 43, 26, 196, 46, 47, 48, 49, // 40 41 42 43 44 45 46 47 48 49
  41, 51, 25, 53, 197, 55, 56, 57, 58, 59, // 50 51 52 53 54 55 56 57 58 59
  60, 61, 62, 63,                     // 60 61 62 63
  // unused (configurations with static particles)
                0,  0,  0,  0,  0,  0, //            64 65 66 67 68 69
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0, // 70 71 72 73 74 75 76 77 78 79
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0, // 80 81 82 83 84 85 86 87 88 89
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0, // 90 91 92 93 94 95 96 97 98 99
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0, // 100 101 102 103 104 105 106 107 108 109
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0, // 110 111 112 113 114 115 116 117 118 119
   0,  0,  0,  0,  0,  0,  0,  0,         // 120 121 122 123 124 125 126 127
  // configurations with a wall (wall = 128 = 10XXXXXX )
                                  128, 136, //                         128 129
  144, 152, 160, 168, 176, 184, 129, 137, 145, 153, // 130 131 132 133 134 135 136 137 138 139
  161, 169, 177, 185, 130, 138, 146, 154, 162, 170, // 140 141 142 143 144 145 146 147 148 149
  178, 186, 131, 139, 147, 155, 163, 171, 179, 187, // 150 151 152 153 154 155 156 157 158 159
  132, 140, 148, 156, 164, 172, 180, 188, 133, 141, // 160 161 162 163 164 165 166 167 168 169
```

```
  149, 157, 165, 173, 181, 189, 134, 142, 150, 158, // 170 171 172 173 174 175 176 177 178 179
  166, 174, 182, 190, 135, 143, 151, 159, 167, 175, // 180 181 182 183 184 185 186 187 188 189
  183, 191,                                 // 190 191
  // unused
          0,  0,  0,  0,  0,  0,  0,  0, //         192 193 194 195 196 197 198 199
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0, // 200 201 202 203 204 205 206 207 208 209
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0, // 210 211 212 213 214 215 216 217 218 219
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0, // 220 221 222 223 224 225 226 227 228 229
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0, // 230 231 232 233 234 235 236 237 238 239
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0, // 240 241 242 243 244 245 246 247 248 249
   0,  0,  0,  0,  0,  0                  // 250 251 252 253 254 255
 };


unsigned  int Collision_Dual_Channels[N_DUAL_CHANNELS] = {
    18, 36, // input configuration 9
     9, 36, // input configuration 18
    45, 54, // input configuration 27
     9, 18, // input configuration 36
    27, 54, // input configuration 45
    27, 45  // input configuration 54
 };


////////////////////////////////////////////////////////////////////////////////
////////// DEFINE GPU FUNCTIONS ////////////////////////////////////////////////////////////

__global__ void Update_Cells (unsigned  int *Volume_GPU, unsigned  int *Collision_Channels_GPU,
                    unsigned  int *Collision_Dual_Channels_GPU, unsigned int z, unsigned int w);
__global__ void Move_Particles (unsigned  int *Volume1_GPU, unsigned  int *Volume2_GPU, unsigned int z,
unsigned int w);


////////////////////////////////////////////////////////////////////////////////
////////// Main                ////////////////////////////////////////////////////////////
```

```
int main(void){

  dim3 blocks (GRID_SIZE_X,GRID_SIZE_Y);
  dim3 threads(BLOCK_SIZE,BLOCK_SIZE);

  unsigned int w = 500450271;
  unsigned int z = 367900313;

  FILE *p;
  int i,j,n,k;
  p=fopen("lattice.txt","w");

  // lookup table on GPU
  unsigned  int *Collision_Channels_GPU;
  unsigned  int *Collision_Dual_Channels_GPU;

  // computational volumes
  unsigned  int *Volume;
  unsigned  int *VolumeA_GPU;
  unsigned  int *VolumeB_GPU;

  // set lookup tables on GPU
  cudaMalloc((void **) &Collision_Channels_GPU, sizeof(unsigned  int)*(256));
  cudaMalloc((void **) &Collision_Dual_Channels_GPU, sizeof(unsigned  int)*(N_DUAL_CHANNELS));
  cudaMemcpy(Collision_Channels_GPU, Collision_Channels, sizeof(int)*(256), cudaMemcpyHostToDevice);
  cudaMemcpy(Collision_Dual_Channels_GPU, Collision_Dual_Channels,
sizeof(int)*(N_DUAL_CHANNELS), cudaMemcpyHostToDevice);

  // set the volume configuration
  Volume = (unsigned  int *)malloc(sizeof(unsigned  int)*(NvolumeX*NvolumeY));
  cudaMalloc((void **) &VolumeA_GPU, sizeof(unsigned  int)*(NvolumeX*NvolumeY));
  cudaMalloc((void **) &VolumeB_GPU, sizeof(unsigned  int)*(NvolumeX*NvolumeY));
```

```
  for (n=0; n<NvolumeX*NvolumeY; n++) Volume[n]=0; // first set everything to empty

  cudaMemcpy(VolumeB_GPU, Volume, sizeof(unsigned  int)*(NvolumeX*NvolumeY),
cudaMemcpyHostToDevice);


  clock_t start = clock();


  for (j=0; j<NvolumeY; j++) for (i=0; i<NvolumeX; i++) { // finally, set walls

     n=j*NvolumeX+i;

     if (j==0 || j==NvolumeY-1) Volume[n]=128; // lower and upper wall

     if (i==10 || i==11){ if (10<j && j<22) Volume[n]=128;} // obstacle (wall)

    }

  for (j=0; j<NvolumeY; j++) for (i=0; i<NvolumeX; i++){

     n=j*NvolumeX+i;

     if(Volume[n]!=128)

     Volume[n]=1;

     }

  cudaMemcpy(VolumeA_GPU, Volume, sizeof(unsigned  int)*(NvolumeX*NvolumeY),
cudaMemcpyHostToDevice);


  // loop over particle dynamics

  printf("\nUnesi n");

  scanf("%d", &n);

  for (k=0; k<n; k++){

    z = 36969 * (z & 65535) + (z >> 16);

    w = 18000 * (w & 65535) + (w >> 16);

    Move_Particles <<<blocks,threads>>>(VolumeA_GPU,VolumeB_GPU,z,w);

    z = 36969 * (z & 65535) + (z >> 16);

    w = 18000 * (w & 65535) + (w >> 16);

    Update_Cells
<<<blocks,threads>>>(VolumeB_GPU,Collision_Channels_GPU,Collision_Dual_Channels_GPU,z,w);

    z = 36969 * (z & 65535) + (z >> 16);

    w = 18000 * (w & 65535) + (w >> 16);

    Move_Particles <<<blocks,threads>>>(VolumeB_GPU,VolumeA_GPU,z,w);

    z = 36969 * (z & 65535) + (z >> 16);
```

```
   w = 18000 * (w & 65535) + (w >> 16);

   Update_Cells
<<<blocks,threads>>>(VolumeA_GPU,Collision_Channels_GPU,Collision_Dual_Channels_GPU,z,w);

  }


  // transfer data back to CPU

  cudaMemcpy(Volume, VolumeA_GPU, sizeof(unsigned  int)*(NvolumeX*NvolumeY),
cudaMemcpyDeviceToHost);


  for (j=NvolumeY-1; j>=0; j--) {

    for (i=0; i<NvolumeX; i++) { // finally, set wallsj=0; j<NvolumeY; j++

      n=j*NvolumeX+i;

      fprintf(p,"%.3d ",Volume[n]);

    }

    fprintf(p,"\n");

  }

  printf ("\n%f", ( (double)clock() - start ) / CLOCKS_PER_SEC );

  fclose(p);

  free(Volume);

  cudaFree(Collision_Channels_GPU);

  cudaFree(Collision_Dual_Channels_GPU);

  cudaFree(VolumeA_GPU);

  cudaFree(VolumeB_GPU);


}


////////////////////////////////////////////////////////////////////////////////

////////// GPU - UPDATE CELLS  ////////////////////////////////////////////////////////

__global__ void Update_Cells (unsigned  int *Volume_GPU, unsigned  int *Collision_Channels_GPU,

                    unsigned  int *Collision_Dual_Channels_GPU, unsigned int z, unsigned int w){


  // localy shared lookup tables for this block

  __shared__ unsigned  int Collision_Channels_shared[256];
```

```
__shared__ unsigned  int Collision_Dual_Channels_shared[N_DUAL_CHANNELS];

// random number
float random;

// collision configurations
unsigned  int input_configuration, output_configuration;

// index of this thread in the volume grid
int it = threadIdx.x;
int jt = threadIdx.y;
int i = BLOCK_SIZE * blockIdx.x + it;
int j = BLOCK_SIZE * blockIdx.y + jt;
int cell_ID = BLOCK_SIZE*jt + it;

// read lookup table into the shared memory
if (cell_ID <256) Collision_Channels_shared[cell_ID]=Collision_Channels_GPU[cell_ID];
if (cell_ID <N_DUAL_CHANNELS)
Collision_Dual_Channels_shared[cell_ID]=Collision_Dual_Channels_GPU[cell_ID];
// synchronize threads to make sure that all data is loaded
__syncthreads();

// update the cell collision
cell_ID = NvolumeX*j + i;
input_configuration = Volume_GPU[cell_ID];

// output configuration
output_configuration = Collision_Channels_shared[input_configuration];
if (output_configuration>191) {
  output_configuration = 2*(output_configuration-192);
  // find random number
  z += cell_ID+1;
  w += cell_ID+1;
```

```
    z = 36969 * (z & 65535) + (z >> 16);
    w = 18000 * (w & 65535) + (w >> 16);
    w = ((z << 16) + w);  /* 32-bit result */
    z = 36969 * (z & 65535) + (z >> 16);
    w = 18000 * (w & 65535) + (w >> 16);
    z = ((z << 16) + w);  /* 32-bit result */
    z = 36969 * (z & 65535) + (z >> 16);
    w = 18000 * (w & 65535) + (w >> 16);
    random=(float)(((z << 16) + w)/(float)UINT32_MAX); // between 0 and 1
    // pick between two options
    if (random < 0.5) output_configuration = Collision_Dual_Channels_shared[output_configuration];
    else output_configuration = Collision_Dual_Channels_shared[output_configuration+1];
  }


  // store the output
  Volume_GPU[cell_ID] = output_configuration;


}


////////////////////////////////////////////////////////////////////////////////

////////// GPU - MOVE PARTICLES ////////////////////////////////////////////////////////

__global__ void Move_Particles (unsigned  int *Volume1_GPU, unsigned  int *Volume2_GPU, unsigned int z,
unsigned int w){


  __shared__ unsigned  int Local_Volume[BLOCK_SIZE+2][BLOCK_SIZE+2];


  // index of this thread in the volume grid
  int it = threadIdx.x;
  int jt = threadIdx.y;
  int i = BLOCK_SIZE * blockIdx.x + it;
  int j = BLOCK_SIZE * blockIdx.y + jt;
  int cell_ID = NvolumeX*j + i;
```

```
 // collision configuration
 unsigned  int configuration;



 // set Local_Volume to zero
 Local_Volume[it+1][jt+1] = 0;
 if (it==0) {
  Local_Volume[it][jt+1] = 0;
  if (jt==0) Local_Volume[it][jt] = 0;
  else if (jt==BLOCK_SIZE-1) Local_Volume[it][jt+2] = 0;
 }
 if (it==BLOCK_SIZE-1) {
  Local_Volume[it+2][jt+1] = 0;
  if (jt==0) Local_Volume[it+2][jt] = 0;
  else if (jt==BLOCK_SIZE-1) Local_Volume[it+2][jt+2] = 0;
 }
 if (jt==0) Local_Volume[it+1][jt] = 0;
 if (jt==BLOCK_SIZE-1) Local_Volume[it+1][jt+2] = 0;
 // synchronize threads to make sure that all data is loaded
 __syncthreads();


 // read configuration
 configuration = Volume1_GPU[cell_ID];


 // move particles
 //if ((configuration & STATIC_PARTICLE) > 0) Local_Volume[it+1][jt+1] |= STATIC_PARTICLE; // static
particle
 if ((configuration & WALL) > 0) Local_Volume[it+1][jt+1] |= WALL; // wall


 if (j%2 == 0) {
  //     5 6
  //      \|
  //     4-O-1
```

```
  //     /|
  //    3 2
  if ((configuration & DIRECTION_1) > 0) Local_Volume[it+2][jt+1] |= DIRECTION_1;
  if ((configuration & DIRECTION_2) > 0) Local_Volume[it+1][jt]   |= DIRECTION_2;
  if ((configuration & DIRECTION_3) > 0) Local_Volume[it][jt]     |= DIRECTION_3;
  if ((configuration & DIRECTION_4) > 0) Local_Volume[it][jt+1]   |= DIRECTION_4;
  if ((configuration & DIRECTION_5) > 0) Local_Volume[it][jt+2]   |= DIRECTION_5;
  if ((configuration & DIRECTION_6) > 0) Local_Volume[it+1][jt+2] |= DIRECTION_6;
 }
 else {
  //      5 6
  //      |/
  //    4-O-1
  //      |\
  //      3 2
  if ((configuration & DIRECTION_1) > 0) Local_Volume[it+2][jt+1] |= DIRECTION_1;
  if ((configuration & DIRECTION_2) > 0) Local_Volume[it+2][jt]   |= DIRECTION_2;
  if ((configuration & DIRECTION_3) > 0) Local_Volume[it+1][jt]   |= DIRECTION_3;
  if ((configuration & DIRECTION_4) > 0) Local_Volume[it][jt+1]   |= DIRECTION_4;
  if ((configuration & DIRECTION_5) > 0) Local_Volume[it+1][jt+2] |= DIRECTION_5;
  if ((configuration & DIRECTION_6) > 0) Local_Volume[it+2][jt+2] |= DIRECTION_6;
 }

 // add new particles into the system
 if (i==0) if ((configuration & WALL) == 0) {
  Local_Volume[it+1][jt+1] = DIRECTION_1;
 }

 // synchronize threads to make sure that all particle in the block moved
 __syncthreads();

 // store the result
 if (it>0 && it<BLOCK_SIZE-1 && jt>0 && jt<BLOCK_SIZE-1)
```

```
  Volume2_GPU[cell_ID] = Local_Volume[it+1][jt+1];
 else {// here other blocks might have produced some results, so we have to use OR
  //configuration = Volume2_GPU[cell_ID];
  //Volume2_GPU[cell_ID] = configuration | Local_Volume[it+1][jt+1];
  atomicOr(&(Volume2_GPU[cell_ID]),Local_Volume[it+1][jt+1]);
  if (it==0 && i>0) {
   //configuration = Volume2_GPU[cell_ID-1];
   //Volume2_GPU[cell_ID-1] = configuration | Local_Volume[it][jt+1];
   atomicOr(&(Volume2_GPU[cell_ID-1]),Local_Volume[it][jt+1]);
   if (jt==0 && j>0) {
    //configuration = Volume2_GPU[cell_ID-1-NvolumeX];
    //Volume2_GPU[cell_ID-1-NvolumeX] = configuration | Local_Volume[it][jt];
    atomicOr(&(Volume2_GPU[cell_ID-1-NvolumeX]),Local_Volume[it][jt]);
   }
   else if (jt==BLOCK_SIZE-1 && j<NvolumeY-1) {
    //configuration = Volume2_GPU[cell_ID-1+NvolumeX];
    //Volume2_GPU[cell_ID-1+NvolumeX] = configuration | Local_Volume[it][jt+2];
    atomicOr(&(Volume2_GPU[cell_ID-1+NvolumeX]),Local_Volume[it][jt+2]);
   }
  }
  else if (it==BLOCK_SIZE-1 && i<NvolumeX-1) {
   //configuration = Volume2_GPU[cell_ID+1];
   //Volume2_GPU[cell_ID+1] = configuration | Local_Volume[it+2][jt+1];
   atomicOr(&(Volume2_GPU[cell_ID+1]),Local_Volume[it+2][jt+1]);
   if (jt==0 && j>0) {
    //configuration = Volume2_GPU[cell_ID+1-NvolumeX];
    //Volume2_GPU[cell_ID+1-NvolumeX] = configuration | Local_Volume[it+2][jt];
    atomicOr(&(Volume2_GPU[cell_ID+1-NvolumeX]),Local_Volume[it+2][jt]);
   }
   else if (jt==BLOCK_SIZE-1 && j<NvolumeY-1) {
    //configuration = Volume2_GPU[cell_ID+1+NvolumeX];
    //Volume2_GPU[cell_ID+1+NvolumeX] = configuration | Local_Volume[it+2][jt+2];
    atomicOr(&(Volume2_GPU[cell_ID+1+NvolumeX]),Local_Volume[it+2][jt+2]);
```

```
    }
   }
   if (jt==0 && j>0) {
    //configuration = Volume2_GPU[cell_ID-NvolumeX];
    //Volume2_GPU[cell_ID-NvolumeX] = configuration | Local_Volume[it+1][jt];
    atomicOr(&(Volume2_GPU[cell_ID-NvolumeX]),Local_Volume[it+1][jt]);
   }
   if (jt==BLOCK_SIZE-1 && j<NvolumeY-1) {
    //configuration = Volume2_GPU[cell_ID+NvolumeX];
    //Volume2_GPU[cell_ID+NvolumeX] = configuration | Local_Volume[it+1][jt+2];
    atomicOr(&(Volume2_GPU[cell_ID+NvolumeX]),Local_Volume[it+1][jt+2]);
   }
  }

  __syncthreads();

  // set previous values to "empty"
  Volume1_GPU[cell_ID] = 0;

}
```