Univerzitet u Beogradu

Matematički fakultet

Mogućnosti primene masivno paralelnog izračunavanja u implementaciji Monte Karlo simulacija

Master rad

Vuk Petković

Beograd,

Univerzitet u Beogradu – Matematički fakultet

Master rad

Autor:	Vuk Petković
Naslov:	Mogućnosti primene masivno paralelnog izračunavanja u implementaciji Monte Karlo simulacija
Članovi komisije:	dr Miodrag Živković, redovni profesor, Univerzitet u Beogradu – Matematički fakultet
	dr Filip Marić, docent, Univerzitet u Beogradu – Matematički fakultet
Za mentora:	dr Saša Malkov, docent, Univerzitet u Beogradu – Matematički fakultet
Datum:	

Sadržaj

1. Uvod

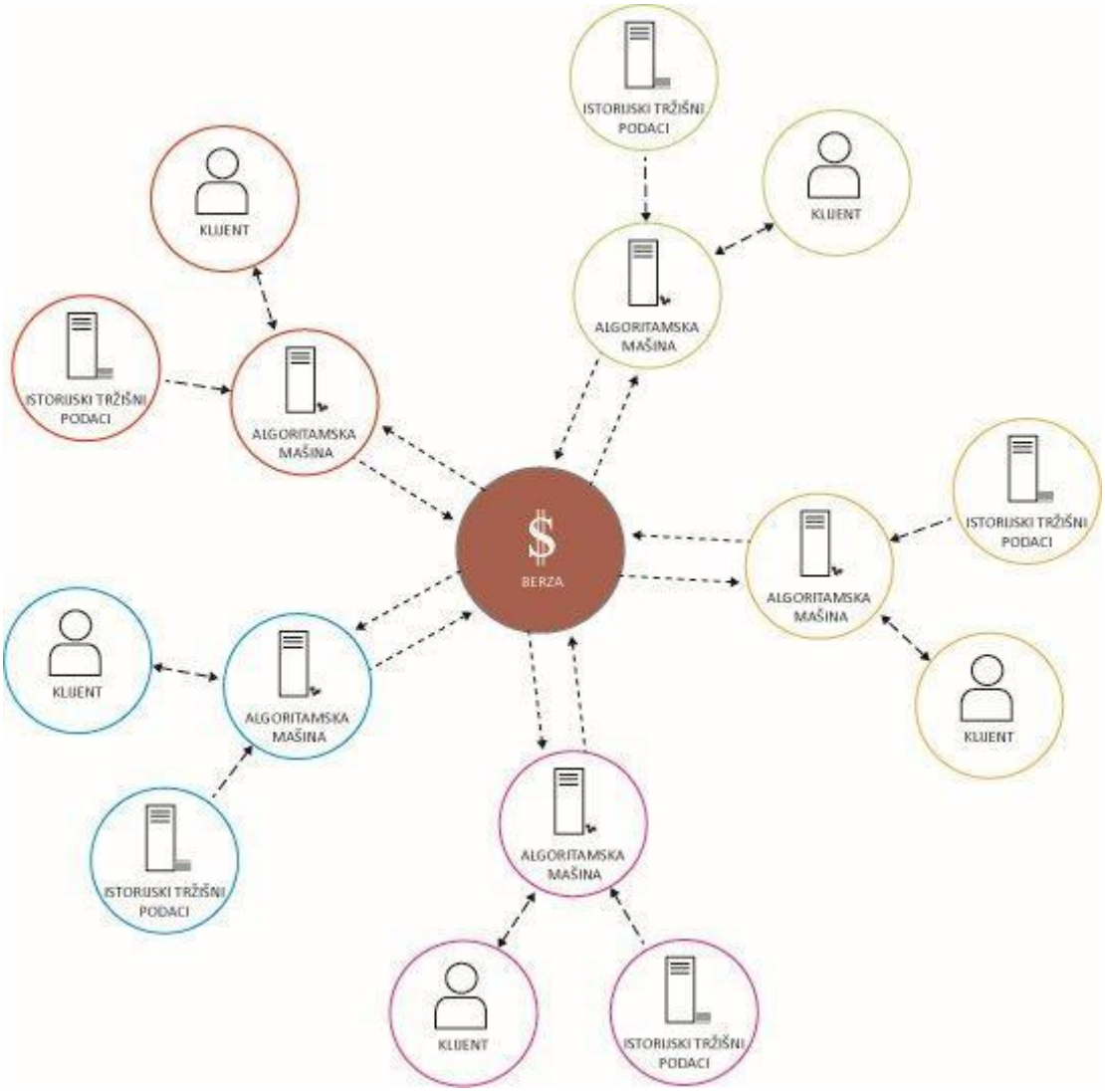
Monte Karlo simulacije (MKS) našle su svoju primenu u raznim oblastima kao što su fizika, primenjena matematika i statistika, kompjuterska grafika, finansije… Razvojem informacionih tehnologija, primena MKS doživljava ekspanziju naročito u oblasti finansija. Uvođenje računarskih sistema i tehnologija u proces trgovine *hartija od vrednosti* dovodi do mogućnosti praćenja cena istih u realnom vremenu. Ovo za posledicu ima nastanak novog oblika trgovanja koje se zove *algoritamsko trgovanje*. Algoritamsko trgovanje podrazumeva upotrebu elektronskih platformi koje same, bez intervencije ljudi, po unapred određenom algoritmu, definisanom određenom strategijom, izvršavaju kupoprodajne naloge za određene hartije od vrednosti. Sama strategija određuje kako optimalno vreme izvršavanja, tako i cenu po kojoj će sama transakcija biti obavljena

U algoritamskom trgovanju MKS se najčešće koriste za određivanje kretanja cena hartija od vrednosti i njihovih *derivata*: *evropskih, američkih i azijskih opcija*, čije će precizne definicije biti navedene kasnije u ovom radu. Iako korisne, MKS su računski veoma skupe pošto pokretanje desetine, ponekad čak i stotine hiljada simulacija zahteva angažovanje ogromne količine računarskih resursa. Sa druge strane, efikasnost MKS u algoritamskom trgovanju se meri brzinom njihovog izvršavanja.

Poslednjih godina, u svim oblastima u kojima su potrebna zahtevna izračunavanja, savremene grafičke kartice doživljavaju pravu ekspanziju zbog svoje jeftinoće i visokih performansi.

U ovom radu biće prikazana implementacija i performanse paralelnog izvršavanja Monte Karlo simulacije na primeru vrednovanja azijske opcije. U skladu sa tim uvodna poglavlja će nas upoznati sa mikrostrukturom tržišta hartija od vrednosti, osnovnim pojmovima iz oblasti finansijskih derivata kao i matematičkim formulama koje se koriste za procenu vrednosti istih. Kasnije se upoznajemo sa CUDA tehnologijom i alatima koje koristimo pri implementaciji MKS za vrednovanje azijske opcije. Na kraju ovog rada upoređujemo performance paralelnog izvršavanja MKS naspram one izvršene sekvencijalnim algoritmom.

1.1 Tržište



Slika 1.1: Primer strukture tržišta

Mnogi veliki investitori na finansijskim tržištima kao što su banke, penzioni i hedž fondovi ulažu velika sredstva, kako u razvoj softverskih tako i u razvoj hardverskih rešenja, koja će im omogućiti što brže izvršavanje algoritama pri trgovanju hartijama od vrednosti na berzama. Uzme li se u obzir i podatak da je u Velikoj Britaniji 77% [8], a u SAD 73% [9] trgovanja svih hartija od vrednosi izvršeno algoritamskim putem, dobija se jasna slika o tome koliko industrija brzih procesora i grafičkih kartica, koje omogućavaju izrčunavanje kompleksnih algoritama u kratkom vremenskom intervalu, dobijaju na važnosti.

Na **slici 1.1** prikazan je primer strukture tipičnog finansijskog tržišta. U centralnom delu slike možemo primetiti *Berzu* , koja simbolizuje centralni server berze na koji su povezani serveri ostalih učesnika na tržistu. Samo tržište funkcioniše tako što svi učesnici na njemu šalju svoje *zahteve (naloge)* za kupovinu ili prodaju određenih akcija, obveznica i drugih finansijskih instrumenata. Zahtev se sastoji od *cene* po kojoj su učesnici na tržištu spremni da kupe ili prodaju određeni finansijski instrument i *količine* koju su spremni da kupe ili prodaju. Na centralnom serveru berze kupoprodajni nalozi se uparuju po određenim pravilima tržišta i na taj način se formiraju cene finansijskih instrumenata i vrši trgovanje između učesnika.

Ono što dodatno komplikuje situaciju jeste naglo razvijanje finansijskih tržišta u poslednjoj deceniji XX i prvoj deceniji XXI veka. Pojava ogromnog broja finansijskih derivata je dala dodatnu dinamiku na finansijskim tržištima širom sveta. Kao jedan od prioriteta svih strategija trgovanja postaje, ne samo određivanje fer vrednosti određenog finansijskog instrumenta već i brzina kojim će se ona odrediti, a samim tim i brzina kojom će se odredjena hartija kupiti ili prodati, sa ciljem pravljena profita. Samo kašnjenje u izračunavanju fer cene, a samim tim i kašnjenje pri izvršavanju kupoprodajnog naloga za posledicu ima gubitak profita. Iz tog razloga brzina je postala najtraženijia a samim tim i najvrednijia "roba" na tržištima širom sveta.

1.1.1 Koncept mikrostrukture tržišta

Prema O'Hari [10], teorija mikrostrukture tržišta se bavi pitanjima kako specifični mehanizmi trgovanja utiču na proces formiranja cena. Hipoteza efikasnog tržišta podrazumeva da su učesnici na tržištu podjednako informisani o objektu trgovanja. Takođe, pretpostavka je da se informacija sadržana u trgovanju odmah reflektuje u ceni. Ona je istovremeno vidljiva svim učesnicima.

Međutim, u praksi ovo nije slučaj. U realnosti, neki učesnici poseduju informacije (koje nisu javne) o objektu trgovanja i to koriste za ostvarivanje profita. Čak i ako je informacija javna, postoji razlika u brzini prenošenja iste različitim učesnicima, što za posledicu ima efekat kašnjenja između objavljivanja vesti i realizacije trgovanja.

Prema tome, sve učesnike na tržištu možemo podeliti na:

1. *Informisane trgovce* - pokazuju sklonost da trguju određenom akcijom o kojoj imaju privatnu informaciju. Oni se mogu prepoznati posmatranjem tržišne aktivnosti, t.j. trguju jedino kada imaju superiornu informaciju sa strategijom da brzo istrguju što veću količinu akcija kako bi ostvarili dobit pre nego što njihova informacija postane javna i dostupna svima.
2. *Neinformisani trgovci* trguju da bi ublažili troškove ili da bi prilagodili rizik svojih portfolija. Oni kupuju akcije u slučaju viška gotovine ili kada su tolerantniji prema riziku, a prodaju akcije kada im je potrebna gotovina ili su manje tolerantni prema riziku.

U istraživanju mikrostrukture tržišta berze u Njujorku (*NYSE*), Hasbrouck je u [5] i [7] dokazao da na tržištu sa asimetrično informisanim trgovcima, trgovanja nose informacije koje dugotrajno utiču na cenu akcija. Nalozi za kupovinu ili prodaju se automatski izvršavaju preko elektronskih sistema kad god se poklope u smislu cene i obima. Elektronski sistemi (**Slika 1.1**) koji omogućavaju trgovanja stvaraju novo okruženje u kojem se razvijaju nove tehnike trgovanja i nova pravila. Trgovanja na ovakvom tržištu postaju mnogo kompleksnija zahtevajući razvijanje dinamičkih strategija trgovanja, t.j. razvoj i implementaciju algoritamskog trgovanja.

1.1.2 Primena koncepta na algoritamsko trgovanje

Algoritamsko trgovanje ponaša se kao trgovanje podstaknuto privatnom informacijom, pošto se svaka informacija nakon realizovanog trgovanja, odslikava u samoj ceni. Ono što se ispostavlja kao krucijalno u ostvarivanju profita je **brzina** kojom se prosleđuju kupoprodajni nalozi, pošto u slučaju tržišta na kome se koristi ovakav način trgovanja mi praktično možemo da napravimo podelu učesnika na dve grupe:

1. *Brži* :

a) Primaju informaciju o ceni nekog finansijskog instrumenta u najkraćem mogućem vremenskom intervalu.

b) Procesiraju datu informaciju (u smislu računanja fer cene) najbržim mogućim algoritmom.

c) Prosleđuju kupoprodajni nalog ka serveru Berze.

2. *Sporiji*:

Koraci u trgovanju ovih učesnika su indetični koracima prve gupe učesnika sa tom razlikom sto se svaki od koraka implementira znatno sporije.

Kao posledicu svega imamo:

1. *Brži učesnici* su ekvivalent informisanih, iz razloga što informaciju mogu brže a procesiraju i pretvore je u profit.
2. Sporiji učesnici su ekvivalent neinformisanih.

Da bi optimizovali brzinu, učesnici na tržištu teže optimizaciji sva tri koraka u obradi informacija: prijem informacije, računanje fer cene i prosleđivanje naloga. Ovaj rad razmatra mogućnosti poboljšanja performansi drugog koraka, odnosno ubrzanje algoritma za procenu vrednosti finansijskog instrumenta. Samo poboljšanje ogleda se u optimizaciji MKS koje se koriste za procenu fer cene hartija od vrednosti. Kao primer koristićemo poseban finansijski instrument – *azijske opcije*. Detaljan opis azijskih opcija nalazi se u poglavlju koje sledi.

1.2 Opcije

1.2.1 Istorijat

Opcije su ugovori koji daju pravo, ali ne i obavezu da se neka roba kupi ili proda po unapred definisanoj ceni, u unapred definisanoj količini i na unapred definisan datum. Precizan istorijski podatak o pojavi prvog ugovora ovog tipa ne postoji. Grci su ovakve ugovore koristili da bi osigurali nisku cenu maslina pre berbe, dok su Rimljani i Feničani koristili slične ugovore prilikom trgovanja. Oko 1600. godine opcije se javljaju u Holandiji, pri čemu im je cilj bio obezbeđivanje odgovarajuće cene lala.

U savremenoj istoriji, opcije se prvi put javljaju u Americi. U XIX veku *kol* i p*ut* opcije su bili ugovori između dve strane i njima se nije moglo trgovati na sekundarnom tržištu. Uslovi su zavisili od ugovora do ugovora i samim tim količina ovakvih ugovora povećavala se jako sporo.

Značajna godina za razvoj trgovanja opcijama je 1968. kada *CBOT* (*Chicago Board of Trade*) uvodi red na tržište opcije zbog mnogih neslaganja u ovim ugovorima. Predložene su dve mere. Prvo, da se standardizuju bitni uslovi u ugovorima kao što su *strike cena* (cena izvršenja), datum dospeća i slično. Drugo, da se stvori organizacija koja će biti posrednik i garant dobrog funkcionisanja tržišta opcija. Taj posrednik je danas poznat pod imenom *Option Clearing Corporation*. Godine 1973. osnovan je *Chicago Board Options Exchange (CBOE).*

Godine 1973. su se pojavili radovi *Black i Scholesa* koji su imali ogroman uticaj na povećanje broja ovih ugovora. Ako se uzme u obzir podatak da broj ovakvih ugovora na godišnjem nivou nije prelazio 300 000, a da je do kraja 1974. godine taj broj bio oko 200 000 dnevno, dovoljno govori o rezultatima predloženih mera CBOT-a i uticaju koji su na trgovanje opcijama imali radovi *Black i Scholesa* [14].

1.2.2 Definicija opcije

Finansijski instrument predstavlja ugovor između dve strane na osnovu koga nastaje finansijsko sredstvo za jednu ugovornu stranu i finansijska obaveza za drugu. *Derivati* su finansijski instrumenti koji imaju osobinu da njihova vrednost zavisi od vrednosti drugog finansijskog instrumenta. Finansijski instrumenti od kojih zavisi vrednost derivata nazivaju se *podloga* (eng. *underlying*) *za derivat*. Podloge mogu biti razni finansijski instrumenti kao što su obveznice, akcije, deonice i slično.

Definicija 1.1

Opcije su ugovori koji daju pravo, ali ne i obavezu da se neka podloga kupi ili proda po unapred definisanoj ceni, u unapred definisanoj količini i na (do) unapred definisan datum.

Postoje dva osnovna tipa opcija:

1. *Kol (eng. Call) opcija* - ugovor koji daje pravo kupcu opcije (eng. *buyer*) da kupi određenu podlogu od onoga ko je napisao opciju (eng. *writer, seller*), na određen datum (koji se naziva datum dospeća), po unapred utvrđenoj ceni (koja se zove *strike cena* ili *cena izvršenja*).
2. *Put opcija* - ugovor koji daje pravo kupcu opcije da proda određenu podlogu, na određen datum, po unapred utvrđenoj ceni.

U samom radu ćemo *datum dospeća* obeležavati sa *T*, a cenu izvršenja sa *K*. U zavisnosti od toga kada ih možemo izvršiti opcije delimo na:

1. *Evropske opcije* - mogu biti izvršene *samo* na datum dospeća.
2. *Američke opcije* - mogu biti izvršene *u bilo kom trenutku* od datuma kupovine do datuma dospeća.

Nazivi američke i evropske opcije nemaju veze sa područjem na kojem se trguje. Evropskim opcijama se trguje u Americi i obrnuto. Neke osobine američkih opcija su izvedene iz evropskih opcija. Pored američkih i evropskih opcija postoje i drugi tipovi opcija. Nekim opcijama prihod zavisi od maksimalne vrednosti podloge tokom određenog vremena. Drugima prihod zavisi od prosečne vrednosti podloge tokom vremena. To su takozvane *azijske opcije* i o njima će biti više reči u nastavku ovog rada.

1.2.3 Evropske opcije

Evropske opcije su finansijski instrumenti (ugovori) koji kupcu daju pravo da kupi ili proda određenu podlogu na datum dospeća T za unapred određenu cenu, ali nije u obavezi da to uradi.

Definicija 1.2

Evropska kol opcija daje pravo njenom vlasniku da *kupi* akciju na *datum dospeća* po *ceni izvršenja K*.

Vlasnik ove opcije očekuje da će cena akcije rasti i biti veća od cene izvršenja na datum dospeća. Sa druge strane pisac opcije očekuje da će cena akcije pasti ispod cene izvršenja na datum dospeća.

Cenu akcije u trenutku *t* ćemo označavati sa *S(t)* a na datum dospeća *T* sa *S(T)*. Prihod vlasnika evropske kol opcije je dat sa:

$$Ck(S,T,K) = \begin{cases} S(T) - K, za\ S(T) > K \\ 0, \quad inače \end{cases}$$

Sa druge strane, ako je $S(T) < K$ prihod pisca opcije će biti:

$$Cp(S,T,K) = \begin{cases} K - S(T), za\ S(T) < K \\ 0, \quad inače \end{cases}$$

Definicija 1.3

Evropska put opcija daje pravo njenom vlasniku da *proda* akciju na datum dospeća po ceni izvršenja *K*.

Vlasnik ove opcije očekuje da će cena akcija padati i biti niža od cene izvršenja na datum dospeća. Sa druge strane pisac opcije očekuje da će cena akcije biti iznad cene izvršenja na datum dospeća.

Prihod vlasnika evropske put opcije je dat sa:

$$Pk(S,T,K) = \begin{cases} K - S(T), za\ K > S(T) \\ 0, \quad inače \end{cases}$$

Sa druge strane, ako je $K < S(T)$ prihod pisca opcije je:

$$Pp(S,T,K) = \begin{cases} S(T) - K, za\ K < S(T) \\ 0, \quad inače \end{cases}$$

1.2.4 Američke opcije

Američke opcije su finansijski instrumenti (ugovori) koji kupcu daju pravo da kupi ili proda određenu podlogu u bilo kom trenutku pre datuma dospeća T za unapred određenu cenu, ali nije u obavezi da to uradi.

Definicija 1.4

Američka kol opcija daje pravo, ali ne i obavezu, kupcu da kupi određenu podlogu u bilo kom trenutku pre datuma dospeća *T*, za unapred određenu cenu.

Prihod vlasnika američke kol opcije je dat sa:

$$Ck(S,t,K) = \begin{cases} S(t) - K, ako\ je\ S(t) > K, t \leq T \\ 0, \quad inače \end{cases}$$

Ponovo, američku kol opciju ima smisla izvršiti samo kada je cena akcije veća od strike cene *K*.

Definicija 1.5

Američka put opcija daje pravo, ali ne i obavezu kupcu da proda određenu podlogu u bilo kom trenutku pre datuma dospeća T, za unapred određenu cenu.

Prihod vlasnika američke put opcije je dat sa:

$$Pk(S,t,K) = \begin{cases} K - S(t), ako\ je\ K > S(t), t \leq T \\ 0, \quad inače \end{cases}$$

Postoje američke opcije bez datuma dospeća (eng. *perpetual american options*, *infinite time horizont american options*).

1.2.5 Azijska opcija

Azijska opcija (ili još poznata pod nazivom *Opcija srednje vrednosti*) je posebna vrsta ugovora. Isplatu po ugovoru azijske opcije određuje prosečna cena neke podloge (finansijskog instrumenta na kog se odnosi ova opcija) tokom nekog vremenskog perioda. U odnosu na evropske i američke opcije, prednost azijskih opcija je u smanjenju rizika od manipulacije tržišta sa cenom neke podloge u trenutku dospeća opcije. Zbog svoje osobine da se odnosi na prosečnu vrednost podloge tokom nekog vremenskog intervala, kod azijske opcije dolazi do smanjenja nestabilnosti u kretanju cene, što za posledicu ima i manju cenu u odnosu na evropske i američke opcije.

I u ovom slučaju razlikujemo:

1. Put opcije

$$P(S,T,K) = \begin{cases} K - S_{avg}(T), & ako\ je\ K > S_{avg}(T) \\ 0, & inače \end{cases}$$

2. Kol opcije

$$C(S,T,K) = \begin{cases} S_{avg}(T) - K, & ako\ je\ S(T) > K \\ 0, & inače \end{cases}$$

K je ugovorena cena, a $S_{avg}(T)$ prosečna cena podloge tokom nekog vremenskog perioda [0,T].

S_{avg} može da se računa na nekoliko načina. Ako se cena podloge posmatra na neprekidnom vremenskom intervalu onda se računa kao

$$S_{avg} = \frac{1}{T}\int_0^T S(t)dt$$

a ako se posmatra na diskretnom skupu tačaka $[t_1, t_2 \ldots\ldots t_T]$ onda se računa kao

$$S_{avg} = \frac{1}{T}\sum_{i=1}^{T} S(t_i)$$

U ovom radu kao primer podloge od koje zavisi cena azijske opcije uzećemo akciju neke kompanije. Poglavlje koje sledi daje nam detaljan opis matematičkog modela koji se koristi za opis kretanja cene akcije. Kako ne postoji analitičko rešenje za procenu fer cene azijske opcije, jedan od najčešćih metoda je Monte Karlo simulacija kojom se generišu slučajne vrednosti cene akcije, a samim tim i određuje fer cena azijske opcije.

1.2.6 Stohastički procesi i model kretanja cena akcija

Za bilo koju promenljivu čija se vrednost slučajno menja tokom vremena kažemo da prati *stohastički proces*. Stohastički proces može biti:

1. *Diskretan* – vrednost promenljive se menja u određenim (fiksnim) tačkama vremenskog intervala

2. *Neprekidan* – vrednost promenljive se može promeniti u bilo kom trenutku vremenskog intervala

Da bismo definisali osnovni model kretanja cena akcija koji se primenjuje u finansijama, potrebno je prethodno definisati dva osnovna pojma kao što su *Markovljev* i *Vinerov proces.*

Definicija 1.6

Markovljev proces (eng. Markov process) je poseban tip stohstičkog procesa kod kog je samo trenutna vrednost promenljive relevantna za predviđanje njenih budućih vrednosti.

Definicija 1.7

Vinerov proces (eng. Wiener process) je poseban tip Markovljevog stohastičkog procesa sa očekivanjem 0 i varijansom 1 u nekom vremenskom intervalu, poznato i kao *Braunov proces (eng. Brownian motion).*

U svojoj doktorskoj disertaciji [15] o matematičkom modelu cena akcija na Pariskoj berzi, Luis Bešoli (*Luis Bachelier, 1870-1946)*, francuski matematičar sa početka XX veka, predlaže aritmetičko *Braunovo kretanje (eng. arithmetic Brownian motion*) kao model kretanja cena akcija:

$$dr(t) = \mu dt + \sigma dW(t)$$

$$r(0) = r_0$$

Problem u vezi ovog modela bio je u tome što je predviđao i negativne cene akcija, a nastale kao osobina Vinerovog procesa da kao promenljiva sa normalnom raspodelom, može da ima i proizvoljno veliki broj negativnih vrednosti. Iako neprimećena u matematičkim i berzanskim krugovima, ovu ideju nekih pedeset godina kasnije oživljava Pol Samjuelson (*Paul Samuelson, 1915-2009).* On unosi jednu bitnu ispravku*: Umesto evolucije cene akcija ABM proces opisuje evoluciju* ***prinosa*** *na akcije.* Rešenje ove stohastičke diferencijalne jednačine je:

$$r(t) = r_0 + \mu \int_0^t du + \sigma \int_0^t dW(u) = r_0 + \mu t + \sigma W(t)$$

Saglasno prethodnom, prinos na akciju je normalna promenljiva. Očekivani prinos menja se linearno sa vremenom, a brzina promene prinosa je data veličinom μ. Varijansa prinosa, t.j. rizik investiranja u akciju, takođe linearno raste sa vremenom i data je izrazom $\sigma^2 t$.

Geometrijsko Braunovo kretanje (eng. Geometric Brownian motion) je osnova dinamičkih modela u finansijama. Najčešće se koristi za predivđanje kretanja cene neke akcije i zadovoljava rešenje stohastičke diferencijalne jednačine:

$$\frac{dS}{S} = \mu dt + \sigma dW(t)$$

$$S(0) = S_0$$

gde je *S* cena neke akcije za koju se ne isplaćuje *dividenda* (deo dobiti akcionarskog društva koja se isplaćuje vlasnicima akcija), S_0 – cena akcije u trenutku t_0, μ – brzina promene prinosa na akciju, σ - varijansa prinosa, t.j. rizik investiranja u akciju, a $W(t)$ je Vinerov proces. Detaljno objašnjenje može se pronaći u [2].

Zapisano u diskretnom obliku, geometrijsko Braunovo kretanje ima oblik:

$$S(t + \Delta t) - S(t) = \mu S(t)\Delta t + \sigma S(t)\varepsilon\sqrt{\Delta t} \qquad \text{gde je } \varepsilon{:}N(0,1).$$

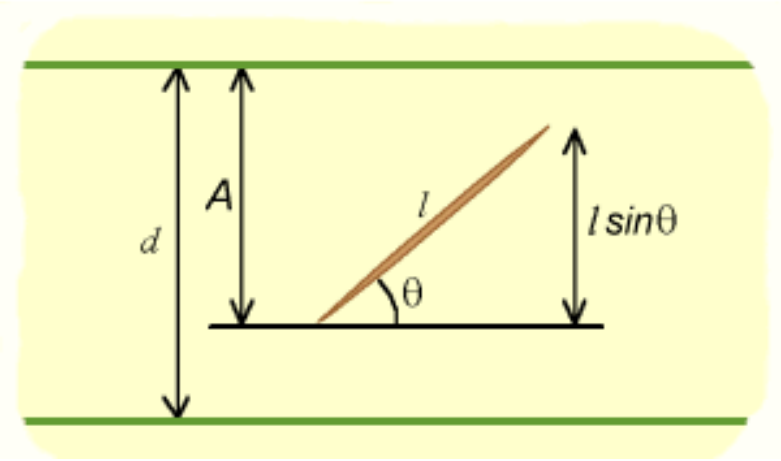
Kako se u praksi za Monte Karlo simulaciju koristi simulacija promenljive *lnS(t)*, a ne *S(t)*, diskretan oblik jednačine $\ln(\frac{S(t)}{S_0}) = \left(\mu - \frac{\sigma^2}{2}\right)t + \sigma W(t)$ izgleda:

$$S(t + \delta t) = S(t)e^{\left(\mu - \frac{\sigma^2}{2}\right)\Delta t + \sigma\varepsilon\sqrt{\Delta t}} \qquad \text{gde je } \varepsilon{:}N(0,1).$$

Ovaj oblik geometrijskog Braunovog kretanja koristimo pri implementaciji Monte Karlo simulacije za vrednovanje azijske opcije.

1.3 Monte Karlo simulacije

Sam princip rada Monte Karlo simulacije nastaje 1777. godine kada se Žorž Luis Lekler poznatiji kao Kompte de Bufon (*Georges Louis Leclerc,Compte de Buffon 1707-1788*) zapitao kolika je verovatnoća da drvce dužine *l* bačeno na rešetku razmaka *d* (*d* >*l*) padne na jednu od linija rešetke (**Slika 1.6**).



Slika 1.6: Primer Bufonovog eksperimenta.

Neka je *x* rastojanje centra drvceta od najbliže rešetke. Drvce dodiruje liniju rešetke ako važi

$$x < (l/2)\sin\theta\,, \qquad \text{gde je } \theta \text{ ugao između drvceta i rešetke.}$$

Kako su *x* i θ slučajne promenljive sa raspodelom:

$$x \sim U(0, d/2) \text{ i } \theta \sim U(0, \pi)$$

sledi da mera prostora Ω svih ishoda slučajnog eksperimenta jednaka

$$\mu(\Omega) = \pi d/2$$

Neka je *A* potprostor prostora Ω takav da su elementi skupa svi ishodi za koje važi

$$x < (l/2)\sin\theta.$$

Kako je mera skupa *A* je $\mu(A) = \int_0^{\pi}(l/2)\sin\theta\, d\,\theta = l$ sledi da je verovatnoća ishoda da drvce dužine *l* bačeno na rešetku razmaka *d* (*d* >*l*) padne na jednu od linija rešetke jednaka

$$p = \frac{\mu(A)}{\mu(\Omega)} = \frac{l}{\pi d/2} = \frac{2l}{\pi d}$$

Koristeći ovo rešenje, matematičar Laplas (*Pierre-Simon Laplace,1749-1827)* je došao do jedinstvenog načina određivanja broja π. Pretpostavimo da je Bufonov eksperiment izveden bacanjem drvca *n* puta. Neka *M* označava koliko puta je drvce palo na liniju, onda je verovatnoća p :

$$p \sim M/n$$

Pošto je ovoj i prethodnoj jednačini jednaka leva strana sledi da im je jednaka i desna t.j. :

$$\pi \sim \frac{n}{M}\,\frac{2l}{d}$$

Formalno, Monte Karlo simulaciju razvili su tokom 1940. godine dvojica američkih naučnika Stanislav Ulam (*Stanislaw Ulam, 1909-1984)* i Džon fon Nojman (*John von Neumann, 1903-1957)* u Los Alamos Nacionalnoj laboratoriji dok su sarađivali na projektu *Menhetn* (*eng. Manhattan).* Simulaciju su koristili da bi odredili slučajnu difuziju neutrona i nazvali su je Monte Karlo, prema gradu u Monaku i njegovim mnogobrojnim kazinima. Projekat Menheten je bio naziv za tajni program vlada SAD, Kanade i Ujedinjenog kraljevstva čiji je cilj bio razvoj atomske bombe.

Danas, Monte Karlo metodu poznajemo kao način rešavanja matematičkih problema generisanjem (pseudo) slučajnih brojeva. Statističkim uzorcima, određenim MKS po unapred utvrđenom algoritmu, aproksimiramo rešenje kvantitativnog problema. MKS se koristi u mnogim oblastima u kojima se izvode zahtevni i komplikovani proračuni, kao što su: fizika, finansije, primenjena matematika, računarska grafika i slično. Prednost današnjice u primeni MKS ogleda se u upotrebi moćnih računara koji imaju kapacitet obrade velikog broja podataka u kratkom vremenskom roku.

U finansijama, Monte Karlo metod se pojavljuje 1964. godine kada je Dejvid Hrec (*David B. Hertz, 1919-2011)* objavio članak u "*Harvard Business Review"*-u [12]. U njemu predstavlja upotrebu MKS u korporativnim finansijama. Trinaest godina kasnije Felim Bojl (*Phelim Boyle, 1941-)* je u svom članku [13] primenio metodu na vrednovanje derivata. Monte Karlo metod je još našao svoju primenu u korporativnim finansijama, u proceni

vrednosti opcije na osnovnu aktivu, u proceni vrednosti instrumenata sa fiksnim prihodom kao i u proceni vrednosti portfolija.

1.3.1 Monte Karlo simulacija geometrijskog Braunovog kretanja

Pri simulaciji geometrijskog Braunovog kretanja, koristimo već pomenutu jednačinu

$$S(t+\Delta t) = S(t)e^{\left(\mu-\frac{\sigma^2}{2}\right)\Delta t+\sigma\varepsilon\sqrt{\Delta t}}$$ gde je ε:N(0,1).

Ova jednačina koristi se za opis kretanja cene akcija u trenucima $t = 1.....T$. Kao što je već pomenuto u delu **1.2.5,** isplata po ugovoru azijske opcije određuje se na osnovu prosečnih cena neke podloge (u ovom slučaju akcije) ostvarenih u trenucima $t = 1....T$:

$$C(S,T,K) = \begin{cases} S_{avg}(T) - K, & ako\ je\ S(T) > K \\ 0, & inače \end{cases}$$ za kol opciju

ili

$$P(S,T,K) = \begin{cases} K - S_{avg}(T), & ako\ je\ K > S_{avg}(T) \\ 0, & inače \end{cases}$$ za put opciju

gde je $S_{avg} = \frac{1}{T}\sum_1^T Si(t)$.

Za fer vrednost azijske opcije uzimamo prosečnu vrednost svih ostvarenih isplata u trenutku T, t.j.

$$V_C = \frac{1}{N}\sum_1^N C_i(S,T,K)$$

ili

$$V_P = \frac{1}{N}\sum_1^N P_i(S,T,K)$$ gde je N ukupan broj simulacija.

Jednačina geometrijskog Braunovog kretanja simulira cenu akcija u nekom budućem vremenskom trenutku T, a samim tim V_c i V_p su fer vrednosti azijske opcije u trenutku T. Posmatrano iz ugla racionalnog investitora, da bismo znali da li je trenutna tržišna vrednost

opcije precenjena ili potcenjena, nas zanima vrednost opcije u sadašnjem trenutku t_0. Vrednost azijske opcije u trenutku t_0 dobija se *diskontovanjem* vrednosti opcije dobijene u trenutku T. Ne ulazeći u detalje, napomenućemo samo da diskontovanje u ekonomiji označava postupak izračunavanja sadašnje vrednosti budućih novčanih tokova. Generalno, buduća vrednost sadašnje novčane mase računa po formuli

$$FV = PVe^{rT}$$

gde je FV buduća vrednost *(eng. Future Value)* novca, PV sadašnja vrednost *(eng. Present Value)* novca, r kamatna stopa u nekom vremenskom periodu, a T broj vremenskih perioda.

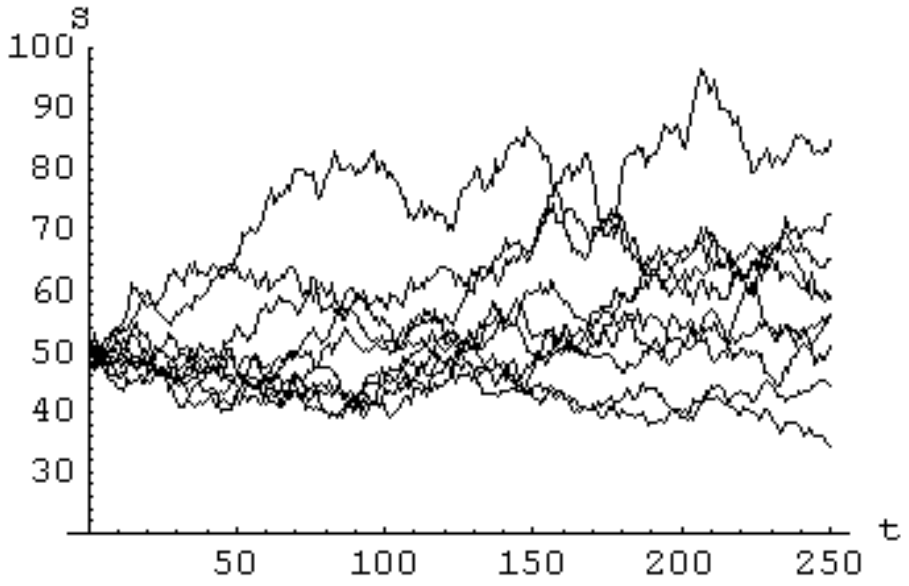
Odavde direktno sledi da se fer vrednost azijske opcije u sadašnjem trenutku t_0, računa kao

$V_{C0} = V_C\, e^{-rT}$, gde je V_{C0} fer vrednost azijske kol opcije u trenutku t_0,

$V_{P0} = V_P\, e^{-rT}$, gde je V_{P0} fer vrednost azijske put opcije u trenutku t_0.

Na osnovu prethodnog, algoritam za određivanje fer vrednosti azijske opcije sastoji od sledećih koraka:

1. Generisanje $N*T$ slučajnih vrednosti koje imaju normalnu raspodelu N(0,1)
2. Generisanje N putanja vrednosti akcije, gde svaka putanja generiše vrednosti akcije u intervalima $t = 1,2 \ldots T$ koristeći jednakost $S(t+\Delta t) = S(t)e^{\left(\mu-\frac{\sigma^2}{2}\right)\Delta t+\sigma\varepsilon\sqrt{\Delta t}}$, preciznije za datu početnu vrednost S_0 u t = 0 pomoću jednačine $S(\Delta t) = S(0)e^{\left(\mu-\frac{\sigma^2}{2}\right)\Delta t+\sigma\varepsilon\sqrt{\Delta t}}$ određuju se vrednosti S(0),S(1),….S(T).
3. Određivanje srednje vrednosti za svaku od putanja
4. Računanje isplata za svaku od putanja
5. Računanje fer cene azijske opcije kao prosečne vrednosti svih ostvarenih isplata
6. Diskontovanje fer cene



Slika 1.7: Primer grafičkog prikaza Monte Karlo sumulacije. Svaka kriva predstavlja grafik jednog kretanja promenljive *S* kroz vreme.

2. Unapređenje implementacije Monte Karlo simulacije

Na početku ovog odeljka upoznajemo se sa tehnologijom *CUDA* koju koristimo za unapređenje implementacije Monte Karlo simulacije. Upoznavanje sa samom tehnologijom će nam dati precizniju sliku o mogućnostima paralelnog izvršavanja simulacija. Takođe razmatramo mogućnosti paralelizacije algoritma za računanje fer vrednosti azijske opcije, a nakon što se odlučimo za adekvatan način, prikazujemo i implementaciju samog algoritma.

2.1 Tehnologija CUDA

Compute Unified Device Architecture (*CUDA*) je arhitektura za paralelna izračunavanja razvijena u kompaniji *Nvidia*-e. Ova ahitektura koristi grafički procesor (GPU) za obradu velikog broja podataka. Zbog narastajuće industrije video igara, grafički procesori su prvobitno bili namenjeni za obradu grafike, a vremenom su ovi procesori evoluirali u paralelne i visoko programabilne procesore. Danas, GPU imaju veliku primenu u ogromnom broju industrija kao što su hemijska, biološka, finansijska, itd. Koriste za kompleksna matematička izračunavanja (koja podrazumevaju pre svega množenje matrica ogromnih dimenzija), obradu signala , baze podataka itd. CUDA predstavlja i niz standarda, a sa programerskog stanovištva svakako je najbitniji programski model koji nudi ekstenzije za rad sa C i C++ programskim jezicima. Postoje i neoficijelni (eng. *third party*) paketi za podršku sledećim jezicima: Python, Perl, Fortran, Java, Ruby, Lua, Matlab, IDL i neke ekstenzije za programski paket Mathematica. CUDA ima i konkurente u vidu OpenCL standarda, koji razvija *Khronos Group*, kao i *DirectCompute* sistema koji razvija kompanija *Microsoft*. Napori *Nvidia*-e u poslednjih nekoliko godina bili su pre svega usmereni ka podizanju nivoa apstrakcije u pogledu programiranja. Neki od ranijih standarda kao što je npr. *OpenGL* nisu imali za cilj da iskoriste GPU za namene generalnog programiranja, već su se ticali uske oblasti 2D i 3D grafike, razvoja igrica i slično. Određeni krugovi IT stručnjaka prepoznali su prednosti koje leže u srži grafičkih procesora za generalna izračunavanja. Nasuprot klasičnih procesorskih jedinica, koje imaju velike skupove instrukcija i centralizovani pristup, grafički procesori imaju mali skup primitivnih operacija, ali distributivno orijentisan model izračunavanja. *Nvidia* je razvojem svog CUDA standarda uspela da olakša razvoj aplikacija i ponudi ga široj zajednici programera. CUDA nudi pristup virtuelnom skupu instrukcija i specifičnom memorijskom modelu koji je sastavni deo CUDA GPU elemenata za paralelna izračunavanja

2.1.1 Stuktura i prevođenje CUDA programa

U ovom odeljku prikazaćemo osnovnu strukturu CUDA programa kao i način njegovog prevođenja i izvršavanja. U **tabeli 2.1** data su pojašnjenja CUDA terminologije koja se koriste u ovom odeljku, do je njihova detaljna specifikacija navedena kasnije.

Termin	Značenje
Domaćin (eng. Host)	Centralna Procesorska Jedinica (eng. CPU)
Uređaj (eng. Device)	Grafička kartica (eng. GPU)
Jezgro (eng. Kernel)	Funkcije koje se izvršavaju od strane više niti
Globalne promenljive	Deo su globalne memorije

Tabela 2.1 Deklaracija CUDA termina

Struktura CUDA programa se može prikazati sledećim koracima:

1. Definisanje globalnih promenljivih i alokacija memorije na domaćinu
2. Definisanje promenljivih koje će se koristiti pri poketanju jezgara i alokcija memorije na uređaju
3. Prenos potrebnih podataka sa domaćina na uređaj
4. Izvršavanje jednog ili više jezgara
5. Prenos rezultata rada jezgara sa uređaja na domaćina
6. Prikazivanje rezultata
7. Oslobađanje alocirane memorije

CUDA program sastoji se od jedne ili više faza koje se izvršavaju bilo na domaćinu bilo na uređaju (eng. *Device* ili *GPU* u CUDA terminologiji), t.j. grafičkoj kartici. Faze koje ne sadrže paralelizam se izrvršavaju na domaćinu, a faze sa obiljem paralelizma se izvršavaju na uređaju. CUDA program je jedinstveni kod koji obuhvata i CPU i GPU kod. Prevodilac nvcc razdvaja ova dva koda u toku procesa prevođenja.

CPU kod je tipičan ANSI C kod, koji je kompajliran standardnim C kompajlerom i izvršava se na CPU. GPU kod je pisan koristeći proširenu verziju ANSI C. Ova verzija je obogaćena ključnim rečima koje služe za obeležavanje funkcija koje se izvršavaju paralelno i nazivaju se *jezgra* (eng. *kernel)*, kao i njima srodnim strukturama podataka. GPU kod se prevodi pomoću nvcc-a i izvršava se na uređaju.

2.1.2 CUDA funkcije

Pravila deklaracija CUDA funkcija se sastoje u sledećem:

1. Jezgra moraju imati kvalifikator __global__
2. Kvalifikator __host__ označava funkcije koje se izvršavaju samo na strani domaćina
3. Kvalifikator __device__ označava funkcije koje se izvršavaju samo na strani uređaja
4. Kvalifikator __host__ i __device__ se mogu koristiti zajedno

	Poziva	Izvršava
__device__ float devFunc()	uređaj	uređaj
__global__ void kerFunc()	domaćin	uređaj
__host__ float hostFunc()	domaćin	domaćina

Tabela 2.2 Deklaracija CUDA funkcija (pozivi i izvršenja)

Izvršna biblioteka obezbeđuje određene ugrađene tipove podataka kao što su*: [u] char[1..4], [u] short [1..4], [u] int [1..4],[u] long[1..4],float [1..4]*. To su structure koje imaju x,y,z,w polja.

Ugrađeni tip *dim3* zasnovan je na uint3 i koristi se za zadavanje dimenzija:

- **dim3 gridDim** - dimenzije rešetke u broju blokova (1D,2D ili 3D), maksimalne dimenzije 65535x65535x65535
- **dim3 blockDim** – dimenzije bloka u broju niti (1D,2D ili 3D), maksimalne dimenzije 512x512x64 (*Tesla* arhitektura 512 niti), 1024x1024x64 (*Fermi* arhitektura 1024 niti)
- **dim3 blockIdx** – indeks bloka unutar rešetke
- **dim3 threadIdx** – indeks niti unutar bloka

Maksimalne vrednosti pojedinih parametara su hardverski zavisne, što implicira njihovu promenu u budućnosti sa razvojem novih serija grafičkih kartica.

Što se tiče matematičkih funkcija, izvršna biblioteka obezbeđuje određeni skup matematičkih funkcija:

- **pow,sqrt,cbrt,hzpot**
- **exp,exp2,expm1**
- **log,log2,logl0,loglp**
- **sin,cos,tan,asin,acos,atan,atan2**
- **sinh,cosh,tanh,asinh,acosh,atanh**
- **ceil,floor,trunc,round**

U slučaju kada se funkcije izvršavaju na strani domaćina, koriste se implementacije iz standardne C biblioteke. One varijante koje imaju slovo **f** u nastavku imena, poput **sinf** rade sa podacima jednostruke preciznosti (float). Takođe, postoje i brže ali manje precizne varijante ovih funkcija koje se mogu izvršavati samo na strani uređaja:

- **__sin**
- **__cos**
- **__tan**

2.1.3 Jezgra

Jezgra su funkcije, koje bivaju izvršene od strane više niti. Bitno je napomenuti da se podela „posla“ ne vrši na nivou obavljanja različitih vrsta aktivnosti, već na nivou različitog skupa ulaznih podataka. CUDA implementira tzv. *Single Instruction Multiple Thread* (SIMT) (srp. Jedna Instrukcija Više Niti) model, što je u suštini *Single Instruction Multiple Data* (SIMD) (srp.Jedna Instrukcija Više Podataka). Niti su u ovom modelu izuzetno jednostavne i ne treba ih poistovećivati sa CPU nitima. Vreme potrebno za njihovo je kreiranje je kratko, a efikasan način kontrole niti omogućava brzo prebacivanje sa jedne na drugu.

Pokretanje jezgra inicira aktiviranje *niti* (eng. *threads*). Svaku nit određuje njen jedinstveni ID. Niti između sebe mogu da sarađuju samo indirektno, korišćenjem neke od dozvoljenih memorija na uređaju:

- Globalna memorija (kvalifikator __device__) je najveća i ima najsporije vreme pristupa. Može joj se pristupati sa uređaja i domaćina u bilo kom momentu tokom izvršavanja aplikacije.

- Deljena (nalazi se na čipu, kvalifikator __shared__) je izuzetno brza memorija. Oko 150 puta je brža od globalne, ali ograničene je količine, svega 16KB po jednoj multiprocesorskoj jedinici (8400 GS ima jednu multi-procesorsku jedinicu, dok GT 230M ima šest multi-procesorskih jedinica).Vidljiva je u okviru određene grupe niti, koja se zove blok.

- Registarska memorija (nalazi se na čipu) je barem toliko brza, koliko i deljena. Ima je manje, a vidljiva je samo niti u kojoj je kreirana. Nema poseban kvalifikator, jednostavno je kreirana u okviru tela jezgarne funkcije.

- Konstantna memorija je sporija od deljene i registarske, ali ne mnogo. Nalazi se u globalnoj memoriji, ali se kešira na čipu, tako da je dosta brža od globalne (kvalifikator __constant__).

Niti se grupišu u blokove niti. Na GeForce 230M grafičkoj kartici koja je ovom prilikom korišćena za implementaciju MKS, maksimalna veličina bloka je 512 niti. Prednost korišćenja niti u istom bloku je u tome što sve mogu da vide deljenu memoriju. Stoga je ovaj parameter grafičke kartice od izuzetnog značaja, posebno ukoliko je potrebno praviti visoko

kooperativnu aplikaciju. Način grupisanja može biti b jednodimenzionalan, dvodimenzionalan ili trodimenzionalan (po x, y i z osi). Ono na šta se mora obratiti pažnja je da ukupan broj iskorišćenih jedinica ne premašuje maksimalan broj. Blokovi se dalje grupišu u takozvane *rešetke* (eng. *grids*), čiji redosled izvršavanja nije deterministički sa aspekta programiranja. Ovakvu organizaciju je povoljno koristiti za relativno nezavisne elemente aplikacije

Kerneli moraju biti pozvani pomoću odgovarajuće izvršne konfiguracije koja je ekstenzija jezika C, a ima oblik:

```
// deklaracija kernelske funkcije

__global__void KernelFunction(.....);

.....

dim3 DimGrid(64,128);                //64*128 = 8192 blokova niti

dim3 DimBlock(32,8);          //32*8 = 256 niti po bloku

KernelFunction<<<DimGrid,DimBlock>>>(arg1,arg2,.....);
```

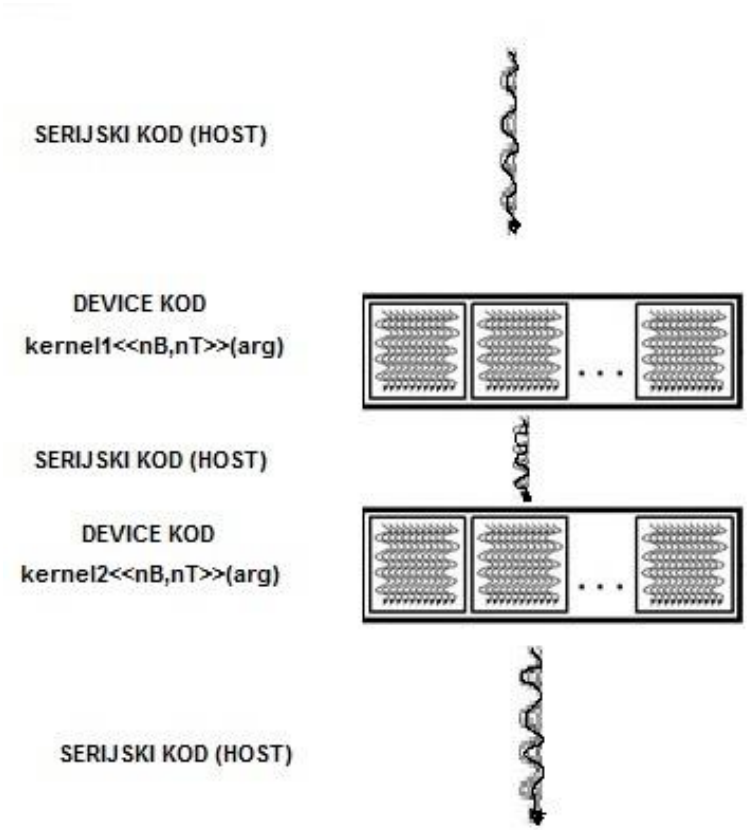
Parametri *DimGrid i DimBlock* definišu organizaciju blokova niti na nivou rešetke i niti na nivou bloka, t.j. broj blokova u rešetki i broj niti u bloku. Postoje još dva opciona parametra: za eksplicitno rezervisanje deljene memorije na nivou bloka i za upravljanje *tokovima* (eng. *streams*). Bitno je napomenuti da je svaki poziv jezgru asinhron, a kontrola se odmah vraća centralnom procesoru. U ovom primeru je definisana dvodimenzionalna rešetka 64*128 i dvodimenzionalni blok 32*8. Tip *dim3* je ugrađeni CUDA tip. Unutar kernela svaka nit određuje podatke nad kojima će raditi pomoću ugrađenih promenljivih *threadIdx i blockIdx.*

Grafički procesor je specijalizovan za računski intenzivna, paralelna izračunavanja. Pogodan je za računanje *data-parallel* tipa. Isti skup instrukcija se izvršava nad velikim brojem podataka istovremeno.Veliki broj izračunavanja se odigrava u odnosu na jedan pristup memoriji. Sva kašnjenja prilikom pristupa memoriji se mogu sakriti intezivnim izračunavanjem umesto velikim keševima podataka. Upravo u tome se i zasniva osnovna ideja korišćenja i upotrebe grafičkih kartica.

Centralni i grafički procesor se najbolje koriste u režimu koprocesiranja. CPU treba koristiti za sekvencijalni deo aplikacije, t.j. u onim slučajevima gde je bitno kašnjenje, pošto je CPU najmanje red veličine brži od GPU-a prilikom izvršavanja sekvencijalnog koda. GPU treba koristiti za delove koda koji troše najviše vremena, t.j. za ubrzanje operacija koje obrađuju veliku količinu podataka.

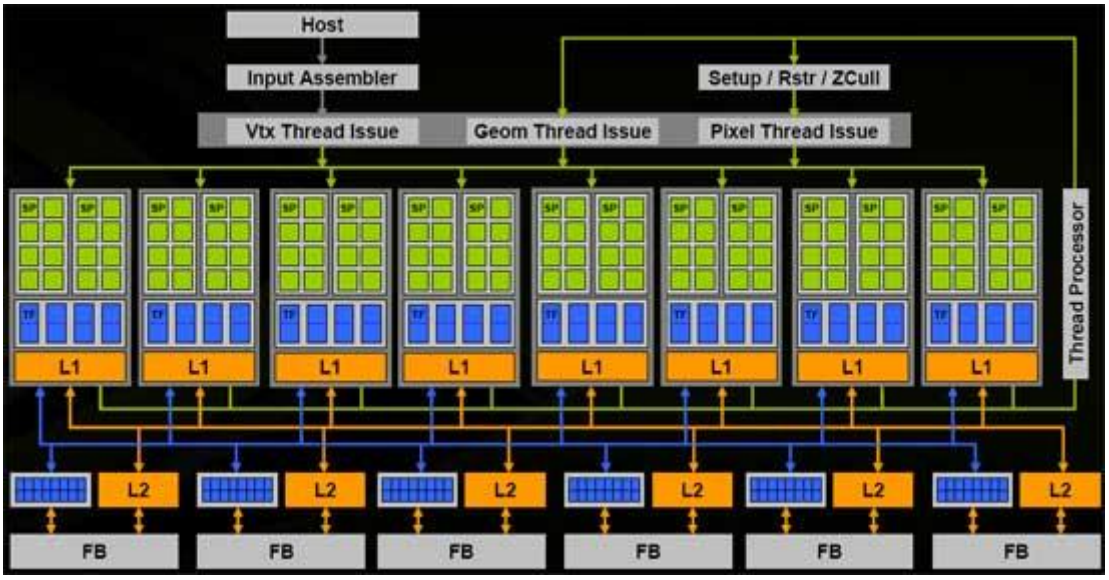
Na **Slici 2.1** prikazan je primer programa sa dve jezgarne funkcije. Ovde se može uočiti osnovna ideja CUDA programa. Prvi deo programa je sekvencijalni deo koji se izvršava na domaćinu. Kada program dođe do dela u kome se obrađuje veliki broj podataka (npr. Množenje matrica velikih dimenzija), onda domaćin prebacuje podatke na uređaj, nakon čega

se aktiviraju niti da izvrše *kernel1* sa brojem blokova nB i brojem niti nT u svakom od bloka. Nakon izvršenog računanja, uređaj vraća rezultat domaćinu na dalju obradu. Nakon obrade podatka domaćin ponovo vraća podatke na uređaj, nakon čega se ponovo aktiviraju niti kako bi se izvršio *kernel2* sa brojem blokova nB i brojem niti nT u svakom od bloka. Ovde je slučajno uzet isti broj blokova u rešetki i niti u bloku pri pozivu obe jezgarne funkcije. To generalno ne mora da bude slučaj. Nakon toga, po drugi put uređaj prosleđuje rezultat, ovog puta rezultat rada jezgra *kernela2*, na domaćina koji vrši krajnju obradu podataka (skladišti, štampa, prikazuje na ekranu…).



Slika 2.1: Primer programa sa dve jezgarne funkcije

2.1.4 Hardverska implementacija



Slika 2.2: Primer arhitekture serije 8 grafički kartica

Hardversku implementaciju GPU-a objasnićemo na primeru arhitekture grafičke kartice G80. Grafički procesor se može posmatrati kao skup multiprocesora. Svaki multiprocesesor je skup 32-bitnih skalarnih procesora SIMD arhitekture. U svakom taktu svaki procesor unutar multiprocesora izvršava istu instrukciju. Ovakav način izvršavanja je skalabilan, t.j. sa lakoćom se barata sa rastućom količinom posla i dodavanjem novih multiprocesora popravljaju se performanse postojeće konfiguracije. Tipična arhitektura (grafički procesori serije 10) sadrže 240 *skalarnih procesora* (SP) koji izvršavaju niti jezgra i organizovani su u 30 *streaming multiprocesor-a* (SM) koji sadrže svaki po 8 SP-a. SM sadrže deljenju memoriju za saradnju na nivou bloka, jednu jedinicu za rad u pokretnom zarezu dvostruke preciznosti i dve jedinice specijalne namene (SFU).

Niti se izvršavaju na skalarnim procesorima, dok se blokovi niti izvršavaju na pojedinačnim multiprocesorima i ne mogu da migriraju. Moguće je takođe da se nekoliko blokova niti izvršava na jednom multiprocesoru (u zavisnosti od potreba za resursima, registrima, deljenom memorijom...). Hardver raspoređuje blokove niti procesorima slobodno, bez ikakvih ograničenja. Jezgra su skalabilana nezavisno od broja procesora na kojima se izvršavaju. Svaki blok se može izvršiti u bilo kom relativnom poretku u odnosu na druge blokove. Niti se multiprocesorima dodeljuju na nivou bloka. Na G80 arhitekturi najviše 768 niti se može izvršavati na jednom multiprocesoru.

Multiprocesor je zadužen za upravljanje i raspoređivanje niti za izvršavanje i održava *blockID* i *threadID* za svaku nit. Niti iz bloka se na SM-u izvršavaju u jedinicama koje se nazivaju *warp*-ovi, gde je 1 *warp* = 32 niti. *Warp* je jedinica za raspoređivanje na SM-u. Ako se, npr. na SM-u izvršava 3 bloka od po 256 niti: Svaki blok se deli na 256/32 = 8 *warp*-ova.

Na SM-u je onda 8*3 = 24 *warp* –ova za izvršavanje.

Blokovi niti se u *warp*-ove dele na sledeći način:

- Indeksi niti unutar bloka su rastući i uzastopni
- *Warp*-ovi se formiraju počevši od niti sa indeksom 0
- Podela se uvek radi na isti način i ta činjenica se može iskoristiti kod kontrole toka

Redosled izvršavanja *warp*-ova ne mora biti jednoznačan. Ukoliko postoje zavisnosti među nitima unutar jednog bloka, mora se vršiti sinhronizacija pomoću funkcije *syncthreads()* koja sinhronizuje niti na nivou bloka. Na SM-ovima je implementirana *zero-overhead* politika raspoređivanja warp-ova. U datom trenutku samo jedan warp se izvršava na SM-u. Raspoloživi za izvršavanje su samo warp-ovi kod kojih su svi operandi dostupni za izvršavanje u narednoj instrukciji. *Warp*-ovi se biraju za izvršavanje na osnovu prioriteta (*round robin + aging*). Sve niti unutar *warp*-a izvršavaju istu instrukciju kada su izabrane. Prosečno vreme pristupa globalnoj memoriji je oko 200 ciklusa procesora. Na G80, 4 ciklusa su potrebna da bi se instukcija poslala svim nitima na izvršenje u warp-u. Ako u proseku, svaka četvrta instrukcija zahteva pristup memoriji, onda je potrebno najmanje 13 warp-ova na jednom SM-u da bi se u potpunosti tolerisalo kašnjenje.

2.2 Alati i tehnologije

Algoritam Monte Carlo simulacije za vrednost azijske opcije implementiran je na platformi *Microsoft Visual Studio 2010*. *Microsoft Visual Studio 2010* je integrisano razvojno okruženje proizvedeno od strane kompanije *Microsoft*. Koristi se za razvoj konzolnih i grafičkih aplikacija, veb sajtova, veb aplikacija i veb servisa kako u izvornom kodu tako i u kodu svih platformi podržanih od strane *Microsoft Windows*, *Windows Mobile*, *Windows CE*, *.NET Framework*, *NET Compact Framework* i *Microsoft Silverlight*. Osnovni alat sa kojim se svaki korisnik susreće je svakako deo za pisanje koda (eng. *Code Editor*) i dizajn veb stranica (eng. *Designer*), kao i integrisani program za otklanjanje grešaka (eng. debugger).

Što se hardverskih karakteristika tiče, na raspolaganju nam je *Intelov* procesor *P7450* sa dva jezgra na 2,13 GHz i 4GB rama, kao i *Nvidia* – ina grafička kartica GeForce GT230M sa 6 MP (*Multi Procesor*) jedinica, 1GB globalne memorije, maksimalnim brojem od 512 niti po bloku i organizaciji od 32 warp-a.

2.3 Razmatranje mogućnosti paralelizacije

Adekvatan izbor načina implementacije MKS-e je od krucijalne važnosti za postizanje željenih performansi pri izvršavanju samog algoritma. Dva su ograničavajuća faktora na koja obraćamo pažnju pri izboru implementacije. Prvi je svakako brzina izvršavanja, dok se drugi ogleda u hardverskom ograničenju, t.j. količini GPU memorije koju eksploatišemo pri izvršavanju programa.

Svaka simulacija, kao što je već napomenuto, pokreće funkciju

MonteCarloSimulation(vreme,slučajna_promenljiva,….);

Pored ostalih ulaznih parametara, opisanih u jednačini geometrijskog Braunovog kretanja, koje su konstantne vrednosti, funkcija sadrži i promenljive *vreme* – uzima vrednosti od 0 do T-1 , gde je T dužina vremenskog intervala u kom želimo da simuliramo kretanje cene akcije i *slučajna_promenljiva* koja uzima vrednosti iz normalne raspodele *N(0,1).* Promenom ove dve vrednosti generišemo putanje kretanja akcije.

U zavisnosti od toga kako ćemo imlementirati korake algoritma opisanog u **poglavlju 1.3**, implementaciju možemo podeliti na:

1. **Jedna nit - više putanja**

Pri ovom načinu implementacije prvo pokrećemo jezgarnu funkciju sa ukupno N*T niti za generisanje N*T slučajnih promenljivih sa raspodelom N(0,1) i smeštamo ih u memoriju uređaja, a zatim u *for petlji* T puta pozivamo jezgarnu funkciju

MonteCarloSimulation<<<N,512>>>(….);

sa N niti, gde jedna nit generiše N vrednosti u nekom trenutku *t.* Drugim rečima, pri svakom pokretanju *for petlje* određujemo vrednosti *N* simulacija u jednom vremenskom trenutku. Ove vrednosti zatim smeštamo na globalnu memoriju. Isti algoritam ponavljamo *T* puta, a zatim za svaku putanju računamo srednju vrednost, a samim tim i isplatu za svaku od njih. Na kraju računamo fer vrednost opcije kao prosek svih isplata.

Kako pri svakoj iteraciji dolazi do razmene podataka između uređaja i domaćina, t.j. u svaki put se prenosi niz veličine *N*sizeof(float),* zbog ograničene memorije uređaja, ovaj način impementacije je pre svega veoma spor. Takodje, ovaj način je i veoma skup u smislu eksploatisanja memorije uređaja. Pri generisanju *N*T* slučajnih vrednosti potrebno je alocitrati memoriju za *N*T*sizeof(curandState)* (detaljnije će biti objašnjeno u poglavlju koje sledi), kao i memoriju za *N*sizeof(float*) na uređaju u koju će biti smeštani rezulati simulacija. Sa druge strane, potebno je alocirati i *N*T*sizeof(float)* na globalnoj memoriji u koju će biti skladišteni podaci sa uređaja, a koji će biti iskorišćeni za statističku obradu pri računanju fer vrednosti. Statistička obrada može da se izvrši, na strani domaćina, sekvencijalnim ili na

strani uređaja, paralelnim izvršavanjem. Ovaj drugi slučaj bi zahtevao dodatno vreme za prebacivanje podataka na uređaj, što bi dodatno usporilo izvršavanje programa.

2. Više niti - jedna putanja

Ovaj način je sličan prethodnom . Razlika se ogleda pre svega u tome što ovog puta vršimo pokretanje jezgarne funkcije

MonteCarloSimulation<<<T+1,512>>>(....);

sa *T* niti, gde jedna nit generiše vrednosti akcije u trenutku *T*. Pokretanjem ove jezgarne funkcije *N* puta u okviru *for petlje* generišemo vrednosti *N* putanja. Da bismo smanjili troškove eksploatisanja memorije, pre svakog poziva jezgarne funkcije u okviru *for petlje MonteCarloSimulation()* vršimo i poziv jezgarne funkcije za generisanje *T* slučajnih vrednosti. Na ovaj način smo smanjili "trošak" memorije *N* puta. Bitno je napomenuti da smo isti princip mogli primeniti i u prethodnom slučaju, ali je u cilju diskusije mogućnosti paralelizacije namerno izostavljen.

Iako smo optimizovali troškove memorije u odnosu na prvi slučaj, statistička obrada dobijenih podataka se vrši isto kao i u prethodnom slučaju, što za posledicu ima sporost algoritma.

3. Jedna nit - Jedna putanja

Da bismo optimizovali i vreme izvršavanja simulacije potrebno je smanjiti komunikaciju , t.j. prenos podataka između domaćina i uređaja. Kako vrednost azijske opcije zavisi od ostvarene srednje vrednosti akcije tokom nekog vremenskog intervala, moguće je kako generisati putanju,tako i računati vrednost opcije u okviru *for petlje* jedne niti. Drugim rečima, nije potrebno pamtiti vrednosti akcije u svakom vremenskom trenutku, već je dovoljno u jednoj lokalnoj promeljivoj čuvati trenutnu sumu ostvarenih vrednosti akcije u trenutcima *0,1,2,...T-1* i jednostavno na kraju izračunati prosečnu vrednost. Šta više, u okviru iste niti vršimo upoređivanje cene izvršenja sa ostvarenom prosečnom cenom akcije i na taj način računamo vrednost azijske opcije za ostvarenu simulaciju putanje kretanja cene akcije.

Ova vrednost je izlaz iz jezgarne funkcije *MonteCarloSimulation()*, a njen poziv paralelno izvršava *N* niti koje predstavljaju *N* simulacija putanja, t.j.

MonteCarloSimulation<<<N,512>>>(....);

Kako bismo simulirali vrednosti akcije tokom nekog vremenskog perioda, prethodno je potrebno generisati *N*T* slučajnih vrednosti sa raspodelom N(0,1) na isti način kao i u prvom slučaju, t.j. pozivom jezgarne funkcije sa *N*T* niti i smeštanjem istih u memoriju uređaja.

Nakon poziva *MonteCarloSimulation()* u memoriji uređaja nalazi se *N* vrednosti isplata azijske opcije. U cilju statističke obrade paralelnim izvršavanjem vršimo sumiranje svih isplata. Ovu vrednost vraćamo domaćinu, gde računamo fer vrednost azijske opcije kao diskontovanu prosečnu vrednost svih isplata. Iz svega navedenog možemo zaključiti da algoritam za računanje fer vrednosti azijske opcije ima sledeću strukturu:

```
main()
  {

//generisanje svih promenljivih i alokacije memorije na domaćinu i uređaju
        ....
//prebacivanje potrebnih podataka sa domaćina na uređaj
        .....
//poziv jezgarne funkcije za generisanje niza  curanState-a

  setup_state<<<N*T,512>>>();

  //pozivanje jezgarne funkcije za simulaciju putanja i isplata opcije
  MonteCarloSimulation<<<N,512>>>();

  //računanje sume svih isplata
  partialSum<<<grid_size,block_size>>>();

        ....
//vraćanje rezultata domaćinu i računanje prosečne vrednosti sume


  }
```

2.4 Implementacija

U ovom poglavlju dato je detaljno objašnjenje algoritma za računanje fer vrednosti azijske opcije čiju smo strukturu objasnili u prethodnom poglavlju. Sam kod implementacije je dat u **Dodatku A**.

Aplikacija za računanje fer vrednosti azijske opcije razvijena je na programskom jeziku C, t.j. preciznije na proširenoj verziji C-a koja je prilagođena za razvoj CUDA aplikacija. Bilo koji izvorni kod koji sadrži CUDA ekstenzije mora se prevesti pomoću *nvcc* prevodioca, koji radi tako što poziva sve neophodne alate i prevodioce (cudacc,cl…). Izlaz *nvcc* prevodioca su *C kod* koji se izvrvšava na strani *domaćina* (eng. *Host* ili *CPU* u CUDA terminologiji), a koji se mora dalje prevesti odgovarajućim prevodiocem i *PTX (Parallel Thread eXecution) kod* koji predstavlja neku vrstu međukoda za grafički procesor. Takođe, kod koji sadrži CUDA pozive, zahteva dve dinamičke biblioteke CUDART i CUDA. Da bismo uopšte mogli da pokrenemo prevođenje potrebno je predhodno instalirati Nvidia-inu

platformu sa nazivom „*Parallel Nsight*“, koja je zapravo dodatak (eng. *plugin*) za *Visual Studio 2010* (platforma koja je korišćena za implementaciju). On poseduje bogato okruženje, a najbitniji je bio alat za otkrivanje grešaka.

Pozivanjem *main()* funkcije prvo se vrši deklaracija i alokacija memorije za ulazne promenljive koje služe kao parametri pri pozivanju jezgarnih funkcija, a pre svega funkcije *MonteCarloSimulation():*

```
//ulazni parametri
float S0;          //početna vrednost akcije
int T;             //broj vremenskih intervala
int N;             //broj simulacija (putanja)
float K;           //cena izvršenja
float mean;        //srednja vrednost prinosa akcije u proteklom periodu
float stdev;       //standardna devijacija prinosa akcije
float r;           //kamatna stopa
```

Zatim sledi deklaracija i alokacija memorije promeljivih i struktura pomoću kojih se vrši komunikacija i prenos podataka između domaćina i uređaja. Nakon toga, deklarišu se promenljive

```
int block_size =512;
int grid_size = N / block_size;
int grid_size_2 = N*T / block_size;
```

koje služe za određivanje broja niti po bloku (*int block_size*), kao i ukupnog broja blokova (*int grid_size i int grid_size_2*) pri pokretanju jezgarnih funkcija. Bitno je napomenuti da je promenljiva *block_size* postavljena na 512 zbog fizičkog ograničenja uređaja na kome je testirana ova simulacija, a koji dozvoljava maksimalno pokretanje od 512 niti po bloku.

Jezgarna funkcija

```
setup_state<<<grid_size_2,block_size>>>(State);
```

se pokreće za ukupno *N*T* niti koje generišu niz *State* od *N*T* članova *curandState*-a. Promenljive *curandState* koriste se u jezgarnoj funkciji *MonteCarloSimulation()* za generisanje brojeva sa raspodelom *N(0,1).*

Nakon pokretanja funkcije *setup_state()* i generisanja niza *State* pokreće se jezgarna funkcija

MonteCarloSimulation<<<grid_size,block_size>>>(d_S0,d_K,d_S,State, T, N, mean, stdev);

sa ukupno *N* niti. Rezultat pokretanja ove funkcije je generisan niz *d_S* koji u sebi sadrži isplate azijske opcije za svaku od putanja generisanih u okviru jedne niti.

Kako bi bila izračunata fer vrednost azijske opcije potrebno je prvo izvršiti sumiranje svih elemenata niza *d_S* , a zatim izračunati prosečnu vrednost svih isplata i istu diskontovati sa kamatnom stopom *r*. U cilju maksimalnog ubrzanja algoritma ovaj posao je podeljen u dve faze. Sumiranje niza *d_S* vrši se na uređaju pozivanjem jezgarne funkcije

partialSum<<<grid_size,block_size>>>(d_S);

koja pokreće *N* niti i vrši paralelno sumiranje svih elemenata niza *d_S*, a rezultat smešta u prvi član datog niza, odnosno u *d_S[0]*. Dobijenu sumu zatim prebacujemo sa uređaja na domaćina pozivom

CUDA_CALL(cudaMemcpy(h_S, d_S, sizeof(float), cudaMemcpyDeviceToHost));

i smeštamo u promenljivu *h_S*. Na kraju štampamo prosečnu diskontovanu vrednost isplata

*printf("%f\n",(h_S[0]/N)*expf(-r*T));*

i osloađamo alociranu memoriju

```
cudaFree(d_S);
cudaFree(d_S0);
cudaFree(d_mean);
cudaFree(d_stdev);
cudaFree(d_K);
cudaFree(State);
```

U nastavku je detaljnije prikazan princip rada sve tri jezgarne funkcije koje smo koristili u implementaciji paralelnog izvršavannja Monte Karlo simulacije za računanje fer vrednosti azijske opcije.

2.4.1 Kernelska funkcija *setup_state*

Pokretanjem jezgarne funkcije

setup_state<<<grid_size_2,block_size>>>(State);

dolazi do generisanja N*T niti. Svaka od niti pokreće funkciju

*curand_init(unsigned long long Seed,unsigned long long Sequence,unsigned long long Offset,curandState_t *state)*

koja generiše *curandState.* Funkcija *curand_init* (), kao i promenljiva *curandState* su deklarisane u biblioteci *CURAND,* a njihova detaljnija specifikacija se može naći u [11]. *CURAND* biblioteka obezbeđuje objekte za jednostavno i efikasno generisanje pseudoslučajnih i kvazislučajnih brojeva i sastoji se iz dva dela: biblioteka za CPU i GPU.

Da bi se koristio API za uređaj potrebno je uključiti zaglavlje *curand_kernel.h.*. Funkcija *curand_init()* generiše stanje *curandState* u zavisnosti od parametara:

1. *Seed* – je šezdesetčetvorobitni ceo broj koji određuje početno stanje generatora pseudoslučajnih brojeva..
2. *Sequence* – predstavlja dužinu sekvence
3. *Offset* – se koristi kada želimo da preskočimo nekoliko mesta u nizu. Ako je *Offse* t = 100, prva generisana slučajna vrednost biće na stotom mestu u nizu.

Različiti *Seed* proizvodi različita početna stanja i različite sekvence, dok isti *Seed* generiše uvek isto početno stanje i samim tim isu sekvencu. Funkcija *curand(curandstate_t *state)* kao parametar za generisanje slučajnih brojeva koristi stanje, t.j. promenljivu tipa *curandstate_t.* Ako generišemo stanje sa nekim parametrom *Seed,* biće potrebno $2^{67} * Sequence + Offset$ poziva funkcije *curand()* da bi se ponovo dobio isti broj.

Pri pozivu funkcije *curand()* koja redom uzima elemente iz nizova stanja *curandState,* a koji su generisani sa različitim parametrom *Seed,* dobijamo slučajne brojeve koji međusobno nisu korelisani. Isto važi i kada se kenerišu stanja sa istim parametrom *Seed* , ali različitim parametrom *Sequence.*

Prema [11, strana 13], za najkvalitetnije paralelno generisanje pseudoslučajnih vrednosti, svakom pokretanju bi trebalo dodeliti jedinstveno početno stanje t.j. *Seed.* Međutim, ukoliko generisanje podrazemeva pokretanje više niti istovremeno (što je naš slučaj), preporučljivo je da niti imaju isto početno stanje, a da parametar *Sequence* bude monotono rastući.

Iz svega navedenog sledi da za "najkvalitetnije" generisanje slučajnih vrednost jezgarna funkcija *setup_state* koja u sebi poziva *curand_init()* bi trebalo da ima oblik:

```
__global__ void setup_state(curandState *state)

{

int id = blockIdx.x * blockDim.x + threadIdx.x ;

curand_init(1234, id, 0, &state[id]);

}
```

Svakoj nit, pri generisanju, dodeljuje jedinsvten identifikacioni broj u okviru rešetke. Ako se pokreće ukupno 512 niti, one će redom imati vrednosti 0,1,2….511. Jedinstven identifikacioni broj dobijamo pozivom

*int id = blockIdx.x * blockDim.x + threadIdx.x;*

Ovo svojstvo koristimo da bi smo parametar *Seqence* postavili upravo na vrednost *id*. Na ovaj način pri svakom generisanju niti i pokretanju jezgarne funkcije *curand_init()* obezbeđuje se da sva stanja budu generisana sa jedinstvenim *Seed*-om i različitim parametrima *Sequence* koji su monotono rastući. Vrednost parametra *Seed* preuzeta je iz primera u [11]. Niz *curandState-a* dužine *N*T* zatim koristimo u kernelskoj funkciji *MonteCarloSimulation()* gde pozivamo funkciju

*curand_normal(curandState *state);*

koja koristi prethodno generisani niz kako bi proizvela pseudoslučajne brojeve sa raspodelom *N(0,1)*. Ova funkcija takođe je sastavni deo biblioteke CURAND.

2.4.2 Kernelska funkcija *MonteCarloSimulation*

Pozivanjem jezgarne funkcije

MonteCarloSimulation<<<grid_size,block_size>>>(d_S0,d_K,d_S,State, T, N, mean, stdev);

generiše se *N* niti, svaka sa jedinstevnim identifikacionim brojem u okviru rešetke:

*int id = blockIdx.x * blockDim.x + threadIdx.x*;

Rezultat pokretanja rešetke je niz *d_S* koji u sebi sadrži isplate azijske opcije za svaku od generisanih putanja u okviru jedne niti. Funkcija *MonteCarloSimulation()* ima osam ulaznih parametara:

1. *d_S0* – početna vrednost akcije
2. *d_K* – cena izvršenja opcije
3. *d_S* – niz u koji se smešaju rezultati simuliranih putanja
4. *State* – niz *curandState-a* pretdodno generisanih paralelnim izvršavanjem jezgarne funkcije *setup_state()*
5. *T* – broj vremenskih intervala u kojima simuliramo kretanje cene akcije
6. *N* – broj simulacija
7. *mean* – srednja vrednost prinosa akcije
8. *stdev* – standardna devijacija prinosa akcije

Nakon deklarisanja lokalnih promenljivih, simulacija putanje odvija se u petlji

```
for(i=0; i<T; i++)
{

curandState local_state = global_state[id*T + i];
s*= expf(MEAN_DT*i + SQRT_DT_STDEV*sqrt((float)i)*curand_normal(&local_state));
global_state[id*T + i] = local_state;
average_value+=s;

}
```

Pri svakoj iteraciji u okviru *for petlje* računa se vrednost akcije u datom trenutku pomoću izraza

```
s*= expf(MEAN_DT*i + SQRT_DT_STDEV*sqrt((float)i)*curand_normal(&local_state));
```

koji predstavlja implementaciju jednačine

$$S(\delta t) = S(0)e^{\left(\mu-\frac{\sigma^2}{2}\right)\Delta t+\sigma\varepsilon\sqrt{\Delta t}}$$

definisane na kraju **poglavlja 1.2.6.** Preciznije, na osnovu date početne vrednosti *S0=S(0),* algoritam računa vrednosti akcije u trenucima 1,2,3...*T*-1, t.j. određuje vrednosti *S(1),S(2),....S(T-1).* Suma svih ostvarenih vrednosti tokom *T* intervala se čuva u lokalnoj promenljivoj *average_value* koja je inicijalno setovana na nulu, a zatim se računa prosečna vrednost svih ostvarenih pomoću izraza

```
average_value = average_value/T;
```

U izrazu za računanje vrednosti akcije u nekom vremenskom trenutku vrši se poziv ranije pomenute funkcije

```
curand_normal(curanState *state);
```

koja kao parametar koristi elemente niza *State,* prethodno generisanog pokretanjem jezgarne funkcije *setup_state().* Generisanje slučajne vrednosti se odvija u tri koraka:

1. Lokalna promenljiva *local_state* tipa *curandState* uzima vrednost iz prethodno generisanog niza *State*:

```
curandState local_state = global_state[id*T + i];
```

2. Poziva se funkcija *curand_normal(local_state)* koja generiše slučajnu promenljivu sa raspodelom *N(0,1).* Pri pokretanju svake niti funkcija *curand_normal()* uzima vrednosti iz niza *State* počevši od vrednosti *id*T,* a zatim sledećih T-1 (za svaki vremenski trenutak po jednu). Tako, na primer, za *id* = 0 *curand_normal()* uzima stanja *State[0], State[1],....., State[T-1].* Zatim za *id = 1,* "preskače" T članova niza i uzima članove *State[T], State[T+1],....., State[2*T-1]* i tako dalje. Ovim se obezbeđuje generisanje različitih slučajnih vrednosti za svaku od putanja.

3. Promenljiva *local_state* se vraća na tekući član niza:

```
global_state[id*T + i] = local_state;
```

Isplata azijske opcije se računa tako što se *average_value* (ostvarena prosečna vrednost akcije tokom jedne simulirane putanje) poredi sa *strike_price* (cenom izvršenja)*:*

```
if((average_value - strike_price) > 0)

d_S [id]= average_value - strike_price;

else d_S [id] = 0;                }
```

U ovom slučaju radi se o isplati za azijsku kol opciju. Na sličan način računa se i isplata za azijsku put opciju, sa tom razlikom što bi se u tom slučaju gledala razlika između cene izvršenja i prosečne ostvarene cene:

```
if((strike_price - average_value) > 0)

d_S [id]= strike_price - average_value;

else d_S [id] = 0;                }
```

Rezultat rada svake od niti smešta se u niz *d_S*, a kako je promenljiva *id* jedinstvena za svaku nit i uzima vrednosti od 0 do N-1, vrednost prvog člana niza d_S[0] biće generisana od strane prve (t.j. nulte) niti i tako redom.

2.4.3 Kernelska funkcija *partialSum*

Ovo je poslednja u nizu kernelskih funkcija koja se poziva u okviru *main()*-a. Funkcija *partialSum()* vrši sumiranje članova niza koji mu je prosleđen kao parametar. Sumiranje članova nekon niza se može izvršiti na više načina, a da bismo našli najoptimalniji, razmotrimo prvi slučaj implementacije:

```
__shared__ float partial Sum[];

__global__void partialSum_B()

{

        unsigned int  t = threadIdx.x;

        for(unsigned int stride = blockDim.x>>1;stride > 0;stride>>1)

{       __syncthreads();
```

```
        If( t < stride)
        partialSum[t] += partialSum[t + stride];
}
}
```

U ovom primeru prikazana je paralelna redukcija sume, t.j. paralelno sumiranje elemenata niza koji se nalazi u globalnoj memoriji. Svaki blok niti smanmjuje sekciju početnog niza tako što učitava elemente sekcije u deljenu memoriju i izvodi paralelno sumiranje. Sumiranje se sprovodi *in place* što znači da će elementi deljenje memorije biti zamenjeni vrednostima parcijalne sume pri svakoj od iteracija. Funkcija *__syncthreads()__* u *for petlji* brine se da se sve parcijalne sume predhodne iteracije generišu i time se obezbedi da su sve niti spremne da uđu u sledeću iteraciju pre nego li bilo koja od njih dobije dozvolu da vrši sledeću iteraciju. Na ovaj način niti će u sledećoj iteraciji koristiti vrednosti generisane u predhodnoj. Nakon finalne iteracije, totalna suma niza biće smeštena u nultom članu početnog niza. Ovaj algoritam će sabirati vrednosti dva susedna člana niza i smestiti ga u predhodni. U slučaju da imamo 512 elemenata niza, biće potrebno 9 iteracija da se odredi suma celog niza. Korišćenjem *blockDim.x* u *for petlji*, pretpostavlja se da je kernel pokrenut sa istim brojem niti koliko ima elemenata u sekciji niza.

Tokom prve iteracije petlje, samo niti sa neparnim vrednostima *threadIdx.x* će izvršiti operaciju sabiranja, t.j. dodavanja, dok će dodatni prolaz biti potreban da se izvrše oni koji nisu predhodno izvršili dodavanje. Ovo jasno dovodi do divergencije niti. Da bi se ovaj problem otklonio i poboljšao potrebna je mala izmena algoritma koji će umesto narednog elementa dodati element koji je udaljen pola sekcije od njega:

```
__shared__ float partial Sum[]
__global__void partialSum_B()
{
        unsigned int  t = threadIdx.x;
        for(unsigned int stride = blockDim.x>>1;stride > 0;stride>>1)
{       __syncthreads();
        If( t < stride)
        partialSum[t] += partialSum[t + stride];
}
}
```

Ovo je omogućeno postavljanjem promenljive *stride* na vrednost jedne polovine dužine sekcije. Posle prve iteracije, sve vrednosti elemenata su smeštene u prve polovine sekcija. Pre ulaska u sledeću iteraciju, vrednost promenljive *stride* se deli sa dva izrazom *stride >>1* koji predstavlja pomeranje vrednosti *stride-a* udesno za jedno bit mesto, a što je brži način nego da smo ga prosto podelili sa dva. Iako i dalje postoji *if()* izjava i broj niti ostaje isti kao i u predhodnom primeru, zbog pozicije niti koje izvršavaju operaciju dodele dolazi do krucijalne razlike između predhodna dva algoritma. Naime, kako će u prvoj iteraciji od 512 elemenata, samo niti sa 0 <= *threadIdx.x <256* izvršiti sabiranje i samim tim parcijalne sume će biti dodeljene elementima niza od 0 do 255, a kako se *warp* sastoji od 32 niti sa uzastopnim vrednostima threadIdx.x, sve nitu u *warp*-u od 0 do 7 će izvršiti dodavanje, dok će oni sa vrednostima od 8 do15 to preskočiti. Pošto sve niti u svakom *warp*-u prate istu putanju, dolazimo do zaključka da ne postoji divergencija niti, što znači da je drugi algoritam brži a samim tim njega uzimamo za implementaciju naše parcijalne sume.

3. Diskusija

U ovom poglavlju dat je tabelarni prikaz rezultata dobijenih pokretanjem Monte Karlo simulacija implementiranih na CPU i GPU, kao i analiza istih. Nakon zaključka o performansama obe verzije, razmatramo kako o mogućnosti primene GPU verzije MKS – e, tako i njeno potencijalno unapređenje. Zbog fer analize najpre upoznajmo dva rivala koja se "nadmeću u ovom okršaju " Monte Karlo simulacije. Sa jedne strane imamo Intelov procesor *P7450* sa dva jezgra na 2,13 GHz i 4GB rama. Sa druge strane imamo *Nvidia*-inu GPU karticu GeForce *GT230M* sa 48 jezgara (eng. *Streaming Processors (SP)*) na 1,1 GHz, 1GB globalne memorije, maksimalnim brojem od 512 niti po bloku i organizaciji od 32 warp-a.

3.1 Analiza rezultata

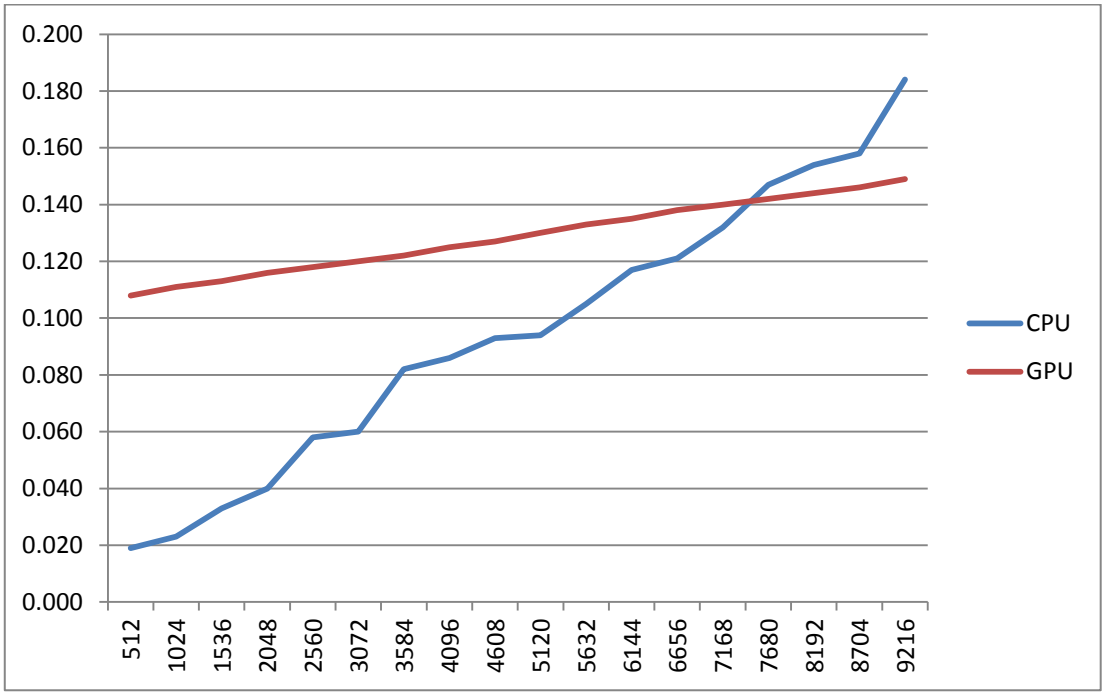
Jedan od važnijih segmenata rada sa optimizacionim algoritmima je svakako i verifikacija njegove korektnosti. Da bi se fer i pravilno uporedile CPU i GPU verzije Monte Karlo simulacije potrebno je testirati performanse obe, kako za mali i veliki broj petlji u glavnoj funkciji, tako i za mali i veliki broj simulacija. Izvršićemo simulacije za T=100, T=1000 i T=10000 i za svako *T* povećavati broj simulacija N sa korakom 512, a zatim uporediti rezultate testiranja. U tabelama koje slede prikazani su rezultati simulacija. Kolona N sadrži broj simulacija, a kolone CPU i GPU vreme izvršavanja programa na domaćinu, odnosno uređaju, izraženo u sekundama.

1. T = 100

Iako je u početku (N = 512 simulacija) CPU neznatno brži kada se radi o malom broju petlji (T = 100), sa povećanjem broja simulacija dolazi do usporavanja procesora u odnosu na GPU, što se može uočiti na grafičkom prikazu **tabele 3.1**, gde horizontalna osa predstavlja broj simulacija sa korakom od 512, dok vertikalna predstavlja vreme izvršavanja simulacija izraženo u sekundama. Možemo primetiti da obe funkcije imaju linearan trend, s tim što grafik CPU verzije ima veći nagib, t.j. "brže usporava" sa porastom broja simulacija. Kod N = 512 simulacija CPU je brži oko 5,68 puta, brzine su približno iste za N = 8192 simulacija, da bi sa brojem od N = 9216 simulacija, GPU nadmašio CPU za oko 1,23 puta.

N	CPU	GPU
512	0,019	0,108
1024	0,023	0,111
1536	0,033	0,113
2048	0,040	0,116
2560	0,058	0,118
3072	0,060	0,120
3584	0,082	0,122
4096	0,086	0,125
4608	0,093	0,127
5120	0,094	0,130
5632	0,105	0,133
6144	0,117	0,135
6656	0,121	0,138
7168	0,132	0,140
7680	0,147	0,142
8192	0,154	0,144
8704	0,158	0,146
9216	0,184	0,149

Tabela 3.1 Vreme izvršavanja simulacija za $T = 100$, izraženo u sekundama



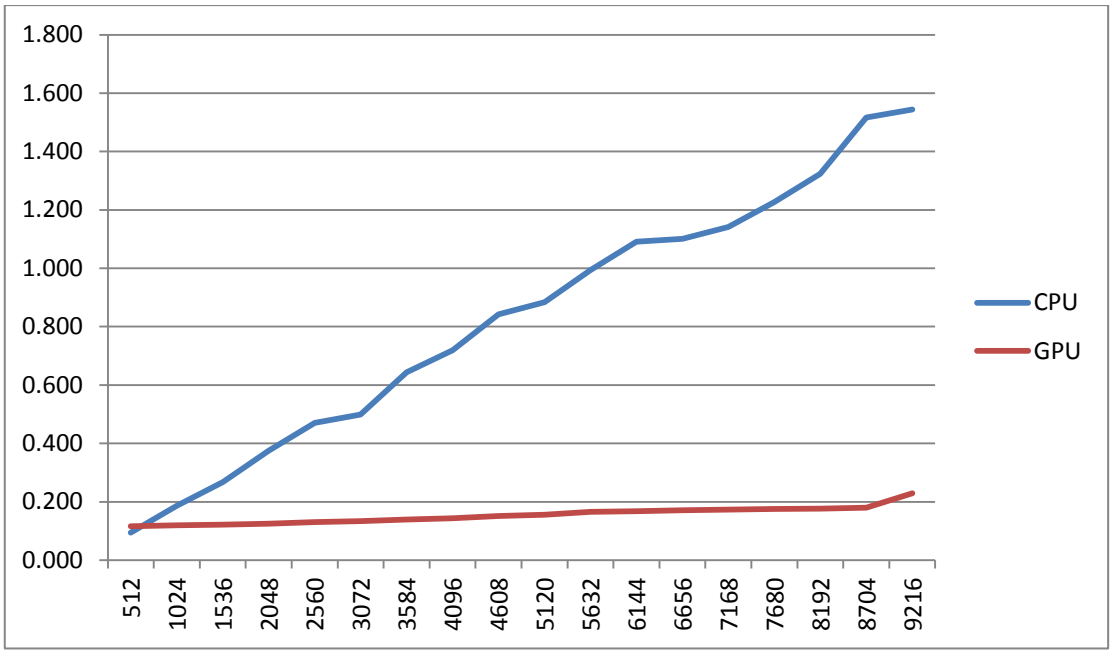
Grafikon 3.1 Grafički prikaz tabele 3.1 Na horizontalnoj osi prikazan je broj simulacija, a na vertikalnoj vreme izvršavanja programa izraženo u sekunda

2. T = 1000

N	CPU	GPU
512	0,094	0,116
1024	0,186	0,119
1536	0,268	0,122
2048	0,375	0,125
2560	0,470	0,130
3072	0,499	0,134
3584	0,644	0,139
4096	0,719	0,144
4608	0,842	0,151
5120	0,884	0,156
5632	0,993	0,165
6144	1,091	0,168
6656	1,101	0,171
7168	1,141	0,173
7680	1,227	0,175
8192	1,324	0,177
8704	1,517	0,180
9216	1,544	0,229

Tabela 3.2 Vreme izvršavanja simulacija za $T = 1000$, izraženo u sekundama

Još jednom je CPU, u početku (N=512 simulacija), neznatno brži i kada se radi o 10 puta većem broju petlji (T=1000) u odnosu na predhodnu analizu, ali takođe sa povećanjem broja simulacija dolazi do usporavanja procesora u odnosu na GPU, što se može uočiti na grafičkom prikazu **tabele 3.2**. Možemo primetiti da obe funkcije imaju ponovo linearan trend, s tim što grafik CPU verzije imaju veći (i strmiji u odnosu na **grafikon 3.1**) nagib, što znači da još "brže usporava" kako sa porastom broja simulacija, tako i broja petlji u samoj funkciji. Kod N=512 simulacija CPU je brži oko 1,23 puta, već u sledećem koraku (N=1024) GPU je brži oko 1,56 puta, da bi sa brojem od N=9216 simulacija GPU nadmašio CPU za oko 6,79 puta. Već ovde bi nam se isplatilo da pri algoritamskom trgovanju koristimo paralelno izvršavanje Monte Karlo simulacije za računanje vrednosti azijske opcije.



Grafikon 3.2 Grafički prikaz tabele 3.2 Na horizontalnoj osi prikazan je broj simulacija, a na vertikalnoj vreme izvršavanja programa izraženo u sekundam

3. T = 10000

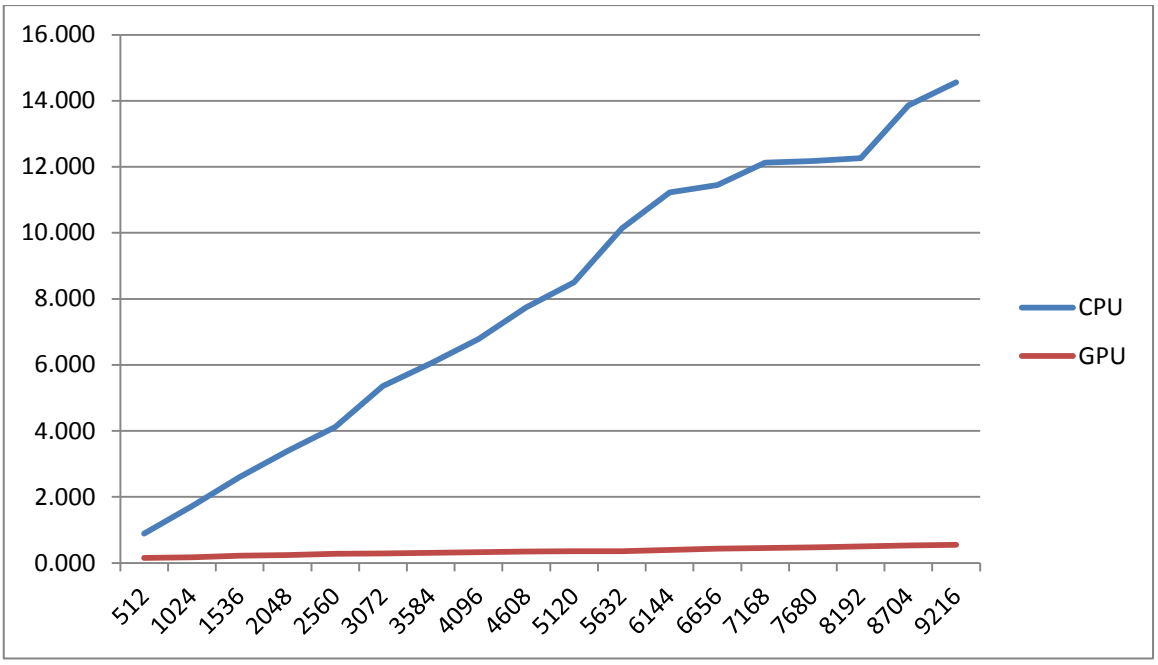
N	CPU	GPU
512	0,892	0,156
1024	1,725	0,172
1536	2,609	0,216
2048	3,396	0,240
2560	4,116	0,275
3072	5,360	0,293
3584	6,048	0,310
4096	6,788	0,331
4608	7,740	0,349
5120	8,497	0,358
5632	10,141	0,361
6144	11,221	0,392
6656	11,446	0,430
7168	12,128	0,449
7680	12,172	0,476
8192	12,259	0,501
8704	13,863	0,532
9216	14,556	0,553

Tabela 3.1 Vreme izvršavanja simulacija za $T = 10000$, izraženo u sekundama

Ovaj slučaj nas najviše zanima zbog same ideje za nastanak ovog master rada, t.j. algoritamsko trgovanje zahteva veliki broj petlji i simulacija. Pri minimalnom broju simulacija N = 512, možemo uočiti superiornost GPU verzije i ubrzanje od oko 5,72 puta u odnosu na CPU verziju, da bi pri maksimalnom broju simulacija to ubrzanje iznosilo 26,32 puta.

Ukoliko testiramo Monte Karlo simulaciju (sekcencijalno izvršavanje algoritma) na najnovijem Intelovom procesoru *Intel i7- 4790K* (4 jezgra, maks. frekvencije 4,4 GHz) možemo očekivati potencijalno ubrzanje od oko 4,4GHz/2,13GHz = 2,0657 puta, t.j. u slučaju T = 10000 i N = 9216, očekivano vreme izvršavanja bi bilo oko 14,556/2,0657 = 7,0465

Sa druge strane, ukoliko uporedimo performanse najnovije *Nvidia*-ine grafičke kartice *GeForce GTX Titan Z* (5760 SP-a, 876 MHz, 12GB memorije) sa *GT 230M* (48 SP-a, 1100MHz, 1GB memorije) možemo primetiti neznatno sporiju frekvenciju novije serije. Međutim za razliku od *GT 230M* koja pokreće maksimalno 512 niti po bloku koja se izvršavaju na svakom od SP-a, *GTX Titan Z* pokreće 1024 niti po bloku. Ukoliko zanemarimo minimalnu razliku u frekvenciji, u slučaju izvršavanja Monte Karlo simulacije (paralelnim algoritmom) sa parametrima T=10000 i N = 9216 možemo očekivati približno isto vreme izvršavanja na obe grafičke kartice. Međutim, u slučaju *GT 230M* ovo su istovremeno maksimalne vrednosti parametara, pošto pri pokušaju pokretanja većeg broja simulacija, sistem javlja grešku u smislu preopterećenosti resursa. Kako je pokrenuto maksimalno 9216 simulacija, a svaki SP pokreće 512 niti, sledi da je ukupno eksploatisano 9216/512 = 18 SP-a od ukupno 48 (6MP-a puta 8 SP-a po MP-u). Sledeći istu logiku možemo pretpostaviti da je pri izvršavanju MKS-e na *GTX Titan Z* moguće "angažovati" oko 5760/2,6 = 2215 SP-a (48/18 = 2,6), što daje ukupno oko 2215*1024= 2268160 niti, t.j. oko 2 miliona simulacija. Upravo u ovome leži ogromna prednost novijih verzija GPU-a, kao što je *GTX Titan Z,* na kojoj je moguće pokrenuti preko milion simulacija i time značajno povećati preciznost samog algoritma u proceni vrednosti opcije. Pokretanje milion simulacija značajno bi usporilo izvršavanje seknecijalnog algoritma čak i na najnovijim serijama procesora. Ukoliko se u obzir uzme i memorija od 12 GB (12 puta veća od memorije koju poseduje *GT 230M*) kojom raspolaže *GTX Titan Z,* dobija se jasna slika o ogromnim mogućnostima masivno paralelnih izračunavanja koje je moguće implementirati i izvršiti na najnovijim serijama grafičkih kartica.



Grafikon 3.3 Grafički prikaz tabele 3.3 Na horizontalnoj osi prikazan je broj simulacija, a na vertikalnoj vreme izvršavanja programa izraženo u sekundama

3.2 Mogućnosti primene

Generalno gledano, mogućnosti primene paralelnog izvršavanja Monte Karlo simulacije su ogromne. Algoritam paralelnog izvršavanja je pre svega upotrebljiv u oblastima u kojima se zahteva generisanje, analiza i obrada velikog broja podataka, kao i u oblastima u kojima je brzina manipulacije podacima značajna u postizanju optimalnih rezultata.

U ovom poglavlju fokusiramo se na primenu paralelnog izvršavanja MKS-e u domenu finansija, pa u skladu sa tim, bez ulaženja u dublju analizu, navodimo nekoliko segmenata iz oblasti finansija u kojima bi paralelno izvršavanje simulacije našlo svoju primenu. Kako se u finansijama Monte Karlo metod pretežno koristi za simulaciju kretanja raznih izvora nesigurnosti koji kao parametri sudeluju i određuju vrednosti raznih finansijskih instrumenata, ukupnog portfolija ili planiranih investicija, paralelno izvršavanje simulacije svoju primenu može naći u:

1. *Korporativnim finansijama*

Jedan od osnovnih zadataka korporativnih finansija jeste investiciono odlučivanje koje podrazumeva selekciju profitabilnih ulaganja. Veliki broj parametara utiče na ostvarivanje profita koji predstavlja razliku između inicijalne investicije i očekivanih novčanih prihoda. Metod Monte Karlo simulacije pomaže finansijskim analitičarima da procene očekivane prihode i rashode, a samim tim i isplativost planiranih investicija. Ukoliko se uzme u obzir veliki broj parametara koje treba uključiti u modele procene kao i procena velikog broja

paniranih investicija u okviru neke kompaije, uviđa se važnost i potreba primene paralelnog izvršavanja Monte Karlo simulacije.

2. *Za procenu vrednosti instrumenata sa fiksnim prihodom*

Osnovni izvor nesigurnosti u ovom slučaju koji se simulira Monte Karlo metodom je kamatna stopa na period od godinu ili više dana. U slučaju obveznica, promena vrednosti kamatne stope, menjaće i vrednost same obveznice. Zato će se primenom metode i simuliranjem dobiti približna slika vrednosti kamatne stope na željeni period, a samim tim i vrednost obveznice u istom periodu. Slično se mogu proceniti i vrednosti raznih derivata kojima je osnovna aktiva upravo kamatna stopa. I u ovom slučaju postoji veliki broj (makroekonomskih) parametara koji utiču na kretanje kamatnih stopa, a ako se zahteva i istovremena procena većeg broja kamatnih stopa, dimenzija problema se uvećava, a samim tim se i princip paralelizacije MKS-a nalazi svoju primenu.

3. *Za procenu vrednosti portfolija*

U ovom slučaju procena vrednosti je slična predhodnim primerima, t.j. Monte Karlo metodom se simuliraju vrednosti svakog finansijskog instrumenta koji se nalazi u portfoliju plus se u razmatranje uzima i njihova eventualna korelacija. Simulirana vrednost svakog finansijskog instrumenta u određenom vremenskom period daće nam procenjenu vrednost čitavog portfolija u datom period. Zbog korelacije velikog broja instrumenata, model zahteva visok stepen saradnje između procesa procene svakog elementa portfolija. Upotreba paralelnog pokretanja niti pri čemu bi svaka vršila procenu jednog finansijskog instrumenta nameće se kao logično rešenje. U prilog ovoj ideji ide i mogućnost koordinacije rada niti.

4. *U upravljanju kreditnim i tržišnim rizicima u bankama i finansijskim institucijama*

Slično pretdodnom primeru, portfolio banke u sebi sadrži kredite više klijenata kao i razne hartije od vrednosti. Svaki klijent može imati po nekoliko kredita, a sami klijenti mogu biti segmentirani u nekoliko grupa (po veličini, tipu proizvoda (kredita) i slično). Analiza performansi klijenata i proizvoda, kao i u pretdodnom primeru, zahteva visok stepen koordinacije pri analizi kako klijenata tako i proizvoda. Monitoring raznih hartija od vrednosti koji se nalaze u portfoliju sličan je primerima 1,2 i 3.

3.3 Mogućnosti daljeg unapređivanja

Najveći uticaj na poboljšanje performansi samog algoritma svakako imaju hardverske osobine uređaja. Sa porastom mogućnosti pokretanja maksimalnog broja niti u okviru rešetke, rastu i performanse paralelnog u odnosu na sekvencijalno izvršavanje. Takođe, kao što je već i spomenuto, memorija uređaja predstavlja bitan faktor pri implementaciji simulacija. Veća memorija samnjuje potrebe komunikacije između uređaja i domaćina. Najnovije serije

grafičkih kartica (*NVidia GeForce GTX TITAN Z*) sadrže u sebi 5760 jezgara i čak 12GB memorije. Međutim, čak i pored velikog broja jezgara i značajne memorije kojom raspolažu najnovije serije *NVidia*-inih grafičkih kartica, arhitektura u kojoj su memorije procesora i grafičke kartice razdvojene značajno utiče na brzinu izvršavanja programa čiji se delovi izvršavaju na grfičkoj kartici. Kašnjenje (eng. *latency*) u početnim koracima izvršavanja programa prouzrokovano je potrebnim vremenom prebacivanja podataka sa CPU na GPU memoriju i nazad. Da bi rešili i ovaj problem i tako poboljšali performanse izvršavanja programa, inžinjeri *AMD*-a su kreirali arhitekturu koja ima hetero memorijsku organizaciju i poznata je kao *HSA* (eng. *Heterogeneous System Architecture*). U HSA-i CPU i GPU dele istu memoriju, tako da je kašnjenje pri prenosu podataka svedeno na minimum. Ovakav tip arhotekture uz povećanje broja jezgara na grafičkoj kartici značajno doprinosi kako brzini izvršavanja MKS-a, tako i kompleksnosti samih algoritama koji se mogu implementirati.

Sa druge strane, pod određenim uslovima postoji optimizacija samog algoritma. Objasnimo ovo na primeru portfolija koji u sebi sadrži nekoliko akcija i čiju je procenu vrednosti potrebno uraditi za 512 (maskimalan broj niti po bloku) vremenskih intervala (npr. dana). Umesto iterativnog pokretanja jezgarne funkcije za svaku akciju posebno moguće je da simuliramo jednu akciju po bloku, pa ćemo pri samo jednom pokretanju moći da generišemo vrednosti za *maksimalan_broj_niti_u_rešetki*/512 akcija.

4. Zaključak

U ovom radu prikazana je primena masivno-paralelnog izračunavanja u simulacijama metodom Monte Karlo. Prikazan paralelni algoritam za izračunavanje fer vrednosti azijske opcije primenom Monte Karlo simulacije implementiran primenom CUDA tehnologije i izvršavan na Nvidia grafičkoj kartici. U radu je takođe opisana i primena ovako generisanog algoritma u algoritamskom trgovanju, što je i bila sama inspiracija za nastanak istog. Pored kratkog osvrta na opcije, samu Monte Karlo simulaciju, Braunovo kretanje i CUDA tehnologiju, prikazano je poboljšanje koje se postiže primenom paralelnog izvršavanja u odnosu na sekvencijalno. Postignuto ubrzanje od oko 26 puta ukazuje nam na to da primena masivno-paralelnog izračunavanja svakako nalazi svoju primenu u algoritamskom trgovanju, gde delić sekunde pravi razliku između profita i gubitka.

Zanimljivo bi bilo uporediti performanske najnovije serije *Intelovih* procesora sa *Tesla* ili *GTX TITAN Z* serijom *Nvidia* grafičkih kartica, kao i najnovijom serijom *AMD* grafičkih kartica sa *HAS* arhitekturom. Jedina, uslovno rečeno, mana samog algoritma ogleda se u tome što rezultat simulirane cene u mnogome zavisi od same performanse generatora slučajnih brojeva. Potencijalna poboljšanja algoritma odnose se na paralelno simuliranje *mean* i *stdev* vrednosti u jednačini kako bi se dobile eventualno preciznije vrednosti cene, što ovom prilikom nije urađeno zbog stavljanja akcenta na samu brzinu izvršavanja algoritma, a čemu autor ovog teksta planira da se posveti u bližoj budućnosti.

Dodatak A – KOD: Paralelno izvršavanje Monte Karlo simulacije

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdlib.h>
#include <cuda.h>
#include <curand_kernel.h>
#include <math.h>
#include <stdio.h>
#include <time.h>
#include <limits.h>
#include <time.h>


#define CUDA_CALL(x) do {   if((x) != cudaSuccess) { \
       printf("Error at %s:%d\n",__FILE__,__LINE__); \
       system("pause"); return EXIT_FAILURE;}} while (0)


//jezgarna funkcija koja generise niz curandState-a
__global__ void setup_state(curandState *state)
{
       int id = blockIdx.x * blockDim.x + threadIdx.x ;

       curand_init(1234, id, 0, &state[id]);

}



//jezgarna funkcija koja generiše putanju akcije u T vremenskih intervala
__global__   void   MonteCarloSimulation(float   *S0,float   *K,float   *d_S,curandState
*global_state, int T, int N, float mean, float stdev)
       {
              float dt = 1.0/T;
              float MEAN_DT = (mean - 0.5*stdev*stdev)*dt;
              float SQRT_DT_STDEV = stdev*sqrt(dt);
              int id = blockIdx.x * blockDim.x + threadIdx.x ;
              int i;
              float average_value = 0;
              float strike_price = *K;
              float s;
              s = *S0;

              //generisanje slucajnih vrednosti
              for(i=0; i<T; i++){

                     curandState local_state = global_state[id*T + i];
                     s*=                          expf(MEAN_DT*i                          +
SQRT_DT_STDEV*sqrt((float)i)*curand_normal(&local_state));
                     global_state[id*T + i] = local_state;
                     average_value+=s;
              }
              average_value = average_value/T;

              if((average_value - strike_price) > 0)

              d_S[id]= average_value - strike_price;

              else
```

```
		d_S[id] = 0;

	}

// Jezgarna funkcija za paralelno sumiranje elemenata niza
__global__ void partialSum(float *S)
	{
		unsigned int t = threadIdx.x;
		for(unsigned int stride = blockDim.x >>1;stride > 0;stride >>=1)
		{
		__syncthreads();
		if( t < stride)
			S[t] += S[t + stride];

		}
	}


int main(int argc, char *argv[])
{
	clock_t start = clock();

	//ulazni parametri
	float S0;		//početna vrednost akcije
	int T;			//broj vremenskih intervala
	int N;			//broj simulacija(putanja)
	float K;		//cena izvršenja
	float mean;		//srednja vrednost prinosa akcije u proteklom periodu
	float stdev;		//standardna devijacija prinosa akcije
	float r;		//kamatna stopa



	if(argc!=6){
			printf("Nije dobar poziv, potreban je sledeci format\n .exe <S0-
pocetna vrednost akcije> <broj vremenskih intervala> <broj simulacija> <kamatna stopa>
<standardna devijacija><cena izvrsenja><kamatna stopa>\n");
		}else{
			S0 = atof(argv[1]);
			T = atoi(argv[2]);
			N = atoi(argv[3]);
			mean = atof(argv[4]);
			stdev = atof(argv[5]);
			K =    atof(argv[6]);
			r =    atof(argv[7]);
		}




	//opšte promenljive
	int num_bytes = N*sizeof(float);
	curandState *State;


	//host promenljive i alokacija memorije
	float * h_S;
	float *h_S0=&S0;
```

```
	float *h_mean=&mean;
	float *h_stdev=&stdev;
	float *h_K =&K;


	//device prmenljive
	float *d_S;
	float *d_S0;
	float *d_mean;
	float *d_stdev;
	float *d_K;


	//alokacija memorije na GPU
	CUDA_CALL(cudaMalloc((void**)&d_S, num_bytes));
	CUDA_CALL(cudaMalloc((void**)&State, N *T* sizeof(curandState)));
	CUDA_CALL(cudaMalloc((void**)&d_S0, sizeof(float)));
	CUDA_CALL(cudaMalloc((void**)&d_mean, sizeof(float)));
	CUDA_CALL(cudaMalloc((void**)&d_stdev, sizeof(float)));
	CUDA_CALL(cudaMalloc((void**)&d_K, sizeof(float)));

	//prenošenje podataka sa host-a na device
	CUDA_CALL(cudaMemcpy(d_S0, h_S0, sizeof(float), cudaMemcpyHostToDevice));
	CUDA_CALL(cudaMemcpy(d_mean, h_mean, sizeof(float), cudaMemcpyHostToDevice));
	CUDA_CALL(cudaMemcpy(d_stdev, h_stdev, sizeof(float), cudaMemcpyHostToDevice));
	CUDA_CALL(cudaMemcpy(d_K, h_K, sizeof(float), cudaMemcpyHostToDevice));


	//postavljanje parametara za poziv kernelskih funkcija
	//512 je hardversko ogranicenje broja niti po blokuk
	int block_size =512;
	int grid_size = N / block_size;
	int grid_size_2 = N*T / block_size;



	//generisanje niza N*T curandState-a
	setup_state<<<grid_size_2,block_size>>>(State);

	//pozivanje kernelske funkcije MonteCarloSimulation
MonteCarloSimulation<<<grid_size,block_size>>>(d_S0,d_K,d_S,State, T, N, mean, stdev);

	//sumiranje svih isplata azijske opcije na device-u
	partialSum<<<grid_size,block_size>>>(d_S);

	//prebacivanje podataka sa device-a na host
	CUDA_CALL(cudaMemcpy(h_S, d_S, sizeof(float), cudaMemcpyDeviceToHost));



	//štampanje rezultata prosečne diskontovane vrednosti
	printf("%f\n",(h_S/N)*expf(-r*T));


	//oslobadjanje memorije

		cudaFree(d_S);
		cudaFree(d_S0);
		cudaFree(d_mean);
		cudaFree(d_stdev);
		cudaFree(d_K);
		cudaFree(State);
```

```
    clock_t ends = clock();
    printf("Vreme programa je:\n");
    printf("%f\n", (double) (ends - start) / CLOCKS_PER_SEC);

    system("pause");



}
```

Literatura

[1] David B. Kirk, Wen-mei Hwu - „Programming Massively Parallel Processors”, Elsevier Inc., 2010

[2] John C. Hull - „OPTIONS, FUTURES, & OTHER DERIVATIVES”, Prentice Hall, 2003

[3] Victor Podlozhnyk, Mark Harris – „Monte Carlo Option Pricing”, Nvidia, 2008

[4] Steven Roman, Introduction to the mathematics of finance, Springer, 2004

[5] Hasbrouck J. - “The Summary Informativeness Content of Stock Trades: An Econometric Analysis”, The Review of Financial Studies, 1991

[6] Hasbrouck J. - “Empirical Market Microstructure”, Oxford, 2007

[7] Hasbrouck J. - “Measuring the Information Content of Stock Trades”, Journal of Finance, 1991

[8] Bloomberg, http://www.bloomberg.com/news/2011-01-24/high-frequency-trading-is-77-of-u-k-market-tabb-group-says.html (pristupljeno 23.3.2012.)

[9] Financial Times, http://www.ft.com/intl/cms/s/0/d5fa0660-7b95-11de-9772-00144feabdc0.html(pristupljeno 23.3.2012.)

[10] O'Hara, M. – “ Market Microstructure Theory”, Cambridge, MA, Blackwell Publishers, (1995)

[11] http://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/CURAND_Library.pdf

[12] David B. Hertz – “Risk Analysis in Capital Investment”, Harvard Business Review, 1964

[13] Phelim P. Boyle - "Options: A monte carlo approach", Journal of financial economics, 1977

[14] Fischer Black, Myron Scholes – “The pricing of options and corporate liabilities”, Journal of Finance, 1973

[15] Louis Bachelie -“ Théorie de la Spéculation”