Математички факултет Универзитет у Београду

Ђорђе Калањ

Примена технологије *CUDA* у прављењу Де Бројновог графа

Мастер рад

Београд, 2015.

Студент:	Ђорђе Калањ 1029/2010
Наслов:	Примена технологије *CUDA* у прављењу Де Бројновог графа
Ментор:	др Саша Малков, доцент Универзитет у Београду – Математички факултет
Чланови комисије:	др Миодраг Живковић, редовни професор Универзитет у Београду – Математички факултет, др Ненад Митић, ванредни професор Универзитет у Београду – Математички факултет
Датум:	06.10.2015

1. Увод

Биоинформатика је дисциплина рачунарства које обухвата развој и примену рачунарских технологија и сазнања у проучавању биологије и медицине. Типични задаци карактеристични за биоинформатику су прикупљање, складиштење, обрађивање и претраживање великих скупова биолошких података, развијање одговарајућих алгоритама, машинско учење, визуализација података и резултата и слично.

Један од главних задатака биоинформатике представљају алгоритми за поравнање секвенци. Њихов задатак је да врше поравнање више секвенци генетских или протеинских података, тако да се постигне максимална оцена сличности. Један од метода за поравнање и реконструкцију секвенци почива на примени Де Бројновог графа.

Тема овог рада је имплементација алгоритма за конструисање Де Бројновог графа уз помоћ графичког процесора. У контексту овог рада конструкција Де Бројновог графа се своди на препознавање чворова и њихов број понављања. У овом раду проблем препознавања чворова се врши на графичкој процесорској јединици. Данашње модерне графичке процесоре одликује паралелна архитектура, која у односу на централне процесорске јединице, за одређене врсте послова, нуди боље перформансе. Из тог разлога графички процесори се све више користе код масовних обрада података. Зато је и конструисање Де Бројновог графа уз помоћ *CUDA* технологије занимљив спој нове технологије и биоинформатичког проблема.

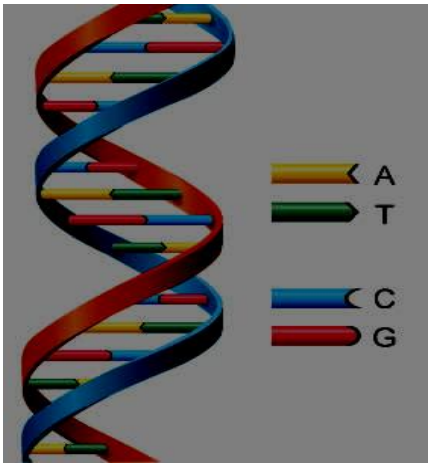
2. Де Бројнов граф и секвенцирање генома

2.1. Генетске секвенце

Геном представља целокупни генетски материјал јединке или врсте. Захваљујући томе што садржи цео скуп наследних особина и инструкција за креирање и одржавање организама, он обезбеђује преношење живота на наредне генерације. Сваки организам на Земљи има јединствени геном који се састоји од гена паковаих у хромозоме. Геном се, физички посматрано, састоји од једног или више молекула ДНК, док се функционално посматрано састоји из гена који носе запис за различите протеине и молекуле РНК. Број гена у геномима различитих организама се веома разликује. Геном може да се дефинише и као одређен редослед нуклеотида у свим молекулима ДНК из којих се састоји.

Ген је физичка и функционална јединица наслеђивања која преноси наследну поруку из генерације у генерацију. Представљена је делом молекула ДНК који се преписују у РНК и засебно је регулисан. Ген је редослед нуклеотида у молекулу ДНК који одређује хемијску структуру специфичног полипептида или молекула РНК.

ДНК је молекул који представља основни генетски материјал сваке живе ћелије. Састављен је од два упарена ланца нуклеотида који су спирално увијени један око другог (Слика 1). Сваки нуклеотид садржи три дела, шећер - дезоксирибозу, фосфатну групу и азотну базу. Дезоксирибоза и фосфатне групе имају градивну улогу и не носе информације. Азотне базе се разликују и представљају носиоце информација. У саставу ДНК могу се наћи четири азотне базе: аденин (А), гуанин (Г), цитозин (Ц) и тимин (Т). Ланац ДНК је једнозначно одређен редоследом база.



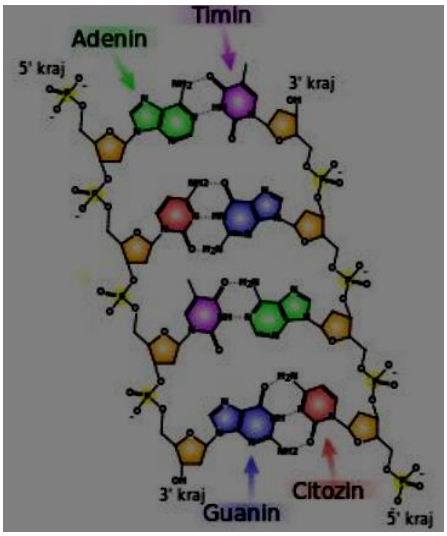
Слика 1. ДНК два упарена ланца нуклеотида[1]

Структура ДНК ланца је таква да су шећери међусобно повезани фосфатним групама. Гради се фосфодиестарска веза између трећег угљениковог атома, једног шећера и петог угљениковог атома следећег шећера (Слика 2). Пошто су

[1]Слика преузета са: http://education.technyou.edu.au/view/91/155/what-does-dna-look

фосфодиестарске везе асиметричне, сваки ДНК ланац има два краја. Ти крајеви се означавају са 5' и 3'. На 5' крају ДНК ланца налази се фосфатна група, док се на 3' крају завршава шећером. Захваљујући асиметричности ДНК ланца, могуће је одредити и његов смер. Сваки ДНК ланац се чита од 5' краја ка 3' крају.

Два ланца ДНК молекула се међусобно повезују преко азотних база. Свака база на једном ланцу повезује се са тачно једном базом другог ланца. Пуринска база (аденин и гуанин) се повезује са пиримидинском базом (тимин и цитозин). Повезивање се врши преко водоничних веза. Аденин се повезује са тимином преко две водоничне везе, док се цитозин повезује са гуанином уз помоћ три водоничне везе. Водоничне везе се лако раскидају и поново успостављају. Овакво повезивање ДНК ланаца назива се комплементарно повезивање. На основу база на једном ДНК ланцу могу се једнозначно установити и базе другог ланца у оквиру молекула. ДНК је одређена када је одређена секвенца база које чине један њен ланац. Због тога је одређивање секвенце база ДНК ланца један од најважнијих корака у изучавању генома неког организма. Поступак одређивања секвенце ДНК се назива секвенцирање генома. Азотна база је носилац генетске информације и представља једину разлику у оквиру нуклеотида.



Слика 2. Структура ДНК молекула[2]

На основу тога, ДНК секвенцу можемо посматрати као низ слова А, Ц, Г, Т, која одговарају низу база у ланцу. Један ДНК ланац преведен на нашу азбуку изгледа као на слици (Слика 3):

AGTCCGGAATACAGCT

Слика 3. Пример ДНК секвенце

Уколико се ДНК посматра на тај начин, поступак секвенцирања генома своди се на одређивање редоследа слова А, Г, Ц, Т у запису секвенце ДНК.

Секвенцирање генома је почетни корак у његовом разумевању. Разумевање генома се често пореди са декодирањем. Секвенца генома добијена секвенцирањем је само једна дуга ниска слова. Можемо је упоредити са реченицом. Да бисмо

[2]Слика преузета са: http://en.wikipedia.org/wiki/DNA

разумели значење прочитане реченице, поред разумевања појединачних слова и речи које сачињавају, потребно је разумети и значење које те речи имају када се споје у целину. Слична ситуација је и код генома код кога такође постоји унутрашње значење које треба открити.

Геном можемо да замислимо као књигу, написану без размака, знакова интерпункције, али и са додатим бесмисленим нискама слова између реченица, као и унутар њих. На тај начин можемо да закључимо колико је, заправо, тешко прочитати га и разумети његово значење. Секвенцирање генома нам не открива све његове тајне. Након секвенцирања и даље остаје посао разумевања значења добијених ниски слова. То подразумева разумевање разноврсних гена у оквиру генома, начина на који су они повезани, као и како су повезани различити делови генома. И поред тога, секвенцирање генома је веома важан део посла, како би се уопште добила секвенца коју треба "разумети".

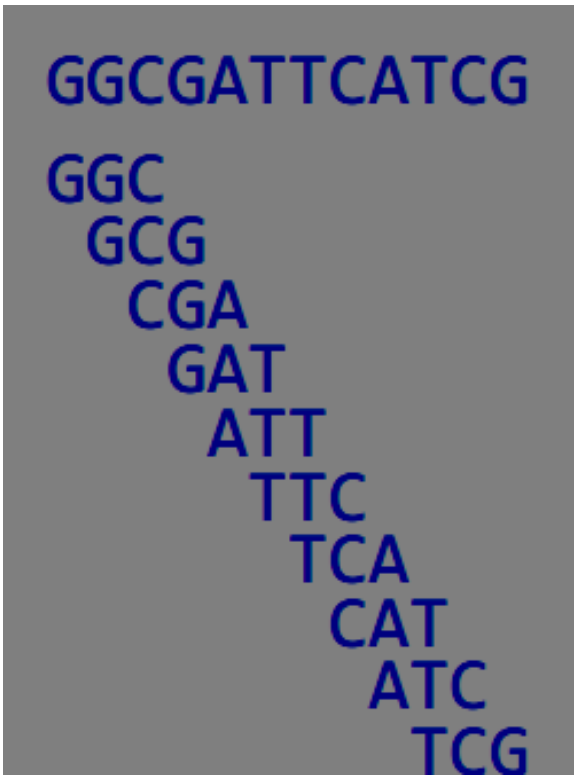
На основу добијене секвенце требало би једноставније и брже пронаћи гене у оквиру генома. Добијена секвенца такође би, требало да омогући и проучавање других делова генома. Међу њима се налазе регулаторни делови који контролишу коришћење одређених гена, и делови који представљају некодирајуће секвенце.

2.2. Секвенцирање генома помоћу де Бројновог графа

2.2.1 *K*–мера

K-мера (енг. *k-mer*) представља подниску дужине *k*. Реч *mer* потиче из грчког језика и значи *део*. Можемо за *k*-мере, рецимо, дужине $k = 3$ да кажемо да су *3-mere*. На слици (Слика 4) видимо све *3-mere* издвојене из дате секвенце.

У раду *k*-мера представља низ нуклеотида дужине *k*. Њих издвајамо из генетске секвенце. Издвајање прве *k*-мере из секвенце се врши тако што се узме првих ***к*** нуклеотида секвенце, Свака *n*-та *k*-мера се добија тако што се на *n*-1 *k*-меру на крај дода (*k*+*n*)-ти нуклеотид из секвенце, а први нуклеотид се уклони.



Слика 4. Издвајање k-мера

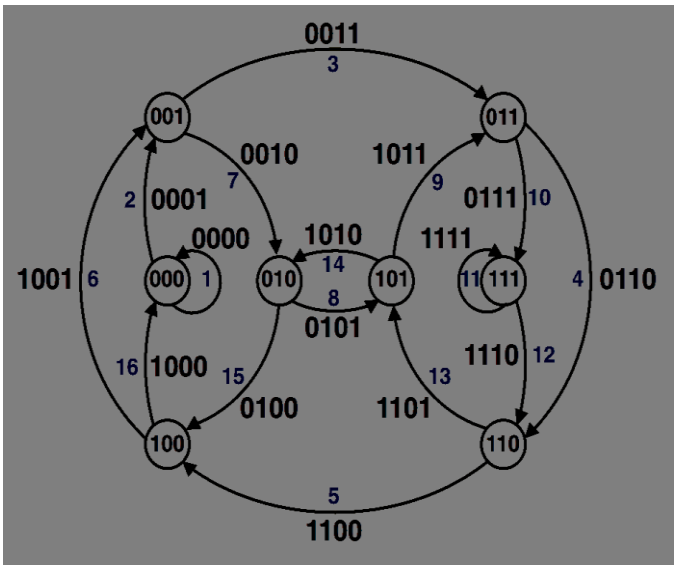
2.2.2 Де Бројнов граф

У контексту овог рада конструкција Де Бројновог графа своди се на препознавање чворова графа и њихов број понављања.

Године 1946. холандски математичар Николас де Бројн је почео да се занима за проблем суперниске. Проблем се састојао у следећем: потребно је пронаћи најмању цикличну суперниску која садржи све могуће подниске дужине к (*k*-мере) над задатом азбуком. У контексту основног проблема овог рада неће се тражити циклична суперниска, већ је у наставку узета ради објашњења самог Де Бројновог графа.

Постоји n^k k-мера у азбуци која садрже *n* симбола. На пример, уколико азбука садржи слова A, T, G и C, постоји $4^3 = 64$ тринуклеотиде. Уколико наша азбука садржи само 0 и 1, онда су 3-мере задате са осам троцифрених бинарних бројева: 000, 001, 010, 011, 100, 101, 110, 111. Кружна суперниска 0001110100 не само да садржи све 3-мере, већ је и најкраћа могућа, пошто садржи све 3-мере само једанпут. Али како конструисати такву суперниску за све *k*-мере када је задата вредност *k* и азбука? Де Бројн је одговорио на ово питање коришћењем Ојлеровог решења за проблем Кенинзбершких мостова. Укратко описано, потребно је конструисати граф за све могуће (*k-1*)-мере додељене чвору, повезати директно једну (*k-1*)-меру са другом (*k-1*)-мером уколико постоји нека *k*-мера чији је префикс прошли, а суфикс следећи чвор. Тачке Де Бројновог графа представљају све могуће *k*-мере и чине Ојлеров циклус који представља најкраћу суперниску, а она садржи сваку *k*-меру тачно једном. Уколико је улазни и излазни степен сваког чвора једнак дужини азбуке, можемо константовати да граф садржи Ојлеров циклус. Тако

можемо конструисати Ојлеров циклус користећи Ојлеров алгоритам и тиме решити проблем супернискe.



Слика 5 – де Бруинов граф

У теорији графова *n*-димензиони Де Бројнов граф од *m* симбола је усмерен граф који представља преклапање између секвенци симбола. Он има m^n темена која се састоје од свих могућих секвенци од датих симбола дужина *n*. Један симбол се може појављивати више пута у секвенци. Ако имамо скуп од *m* симбола $S = \{s_1,\ldots, s_m\}$ онда нам је скуп чворова :

$$V = S^n = \{(s_1,\ldots,s_1,s_1,),(s_1,\ldots,s_1,s_2,),\ldots, (s_1,\ldots,s_1,s_m,),(s_1,\ldots,s_2,s_1,),\ldots,(s_m,\ldots,s_m,s_m,)\}$$

Ако један од чворова може бити изражен преко другог чвора тако што ћемо све његове симболе померити за једно место у лево и додати му нови симбол на крај, онда тај други чвор има усмерену везу ка првом чвору. Онда је скуп лукова (усмерених веза) :

$$V = \{((v_1,v_2,\ldots, v_n),(v_2,\ldots, v_n,s_i)):i=1,\ldots,m\}$$

2.2.3. Алгоритам реконструкције генома применом Де Бројновог графа

У пракси покушаји да се примени Де Бројнов граф над експерименталним подацима не представљу лак посао. Следи опис неких од кључних рачунарских техника које су осмишљене да одговоре на изазове који се односе на грешке код постојећих технологија секвенцирања, као и да реше сложеност насталу великим бројем понављања узастопних генома. Де Бројнова метода се ослања на четири претпоставке:

- гарантовано можемо издвојити сваку *k*-меру присутну у геному;
- све *k*-мере су без грешке;
- свака *k*-мера се појављује највише једном у геному и
- геном се састоји од једног кружног хромозома.

Издвајање скоро свих *k*-мера присутних у геному

Читањем 100-мера издвојених узима се само један део скупа 100-мера из генома (чак и за мале узорке секвенциониране великом покривеношћу), што не задовољава кључну претпоставку Де Бројнових графова. Међутим, ако разбијемо ове мере у ситније *k*-мере, резултат *k*-мера често представља скоро све *k*-мере из генома за довољно мало *k*.

Управљање грешкама приликом читања

Метод упареног читања који подразумева да секвенцу читамо од почетка према крају и од краја према почетку. Та два читања даље можемо искористити за реконструкцију секвенце на основу преклапања та два читања.

Реконструкција генома се своди на склапање генома из више *k*-мере које су спојене директним гранама у Де Бројновом графу и чија дужина преклапања задовољава дужину преклапања која се задаје као улаз алгоритма.

Свака грешка у читању креира гране у Де Бројновом графу и компликује реконструкцију генома. Да ствар буде још лошија, у геному са разликама у поновљеном читању (на пример уколико су два региона разликују за један нуклеоид или за малу варијацију), ствара се грана у Де Бројновом графу. Упареним читањем би такође генерисало грану у Де Бројновом графу. Приступ за решавање проблема приликом читања, са којим би се грешке избегле пре самог почетка креирања графа предоложен је 2001. године и нашироко је у употреби. Грешке у читањима генеришу темена у пределима преклопа два читања стварајући тиме нове гране. У односу на број понављања *k*-мере може се одлучити које гране су носиоци греше и њиховим одстрањивањем се елеминишу грешке читања. Алгоритам је унапређен 2004. године, са неким варијацијама, и користи се у данашњим асемблерима за кратке секвенце (нпр. *EULER-SR16*, *Velvet17*, *ALLPATHS18*, *ABySS19* и *SOAPdenovo20*). Ови и остали, скоро развијени алати садрже многе алгоритме и софтверско инжењерске идеје које су отвориле пут за склапање великих генома са следећом генерацијом секвенционисања података. У основи алгоритми раде у два корака. У првом кораку из кратких читања се издвајају *k*-мере и конструише се Де Бројнов граф, затим се у другом кораку одсецају гране које не задовољавају услове алгоритма односно које се препознају као грешке по алгоритму.

Руковање понављањима ДНК

Замислимо секвеницијална 3-базна читања цикличног генома ATGCATGC. Требало би да креирамо четири 3-мере: ATG, TGC, GCA и CAT. Међутим, водећи се дефиницијом Де Бројновог графа, реконструисали бисмо геном као ATGC. Проблем је у томе што се свака од 3-мера заправо понавља два пута у оригиналном геному. Због тога, потребно је реконструисати геном тако да пронађемо колико се пута појављује свака *k*-мера, а не само које се све *k*-мере понављају у геному. Добра вест је та што можемо да обрадимо фрагмент склапања у случају када је број понављања *k*-мере познат. Једноставно користимо исту методу за конструисање Де Бројнових графа, с тим што, уколико је број понављања *k*-мере *m*, повезујемо њен префикс са њеним суфиксом користећи *m* директних ивица (уместо само једне). Уколико приликом читања откријемо да је свака од четири 3-мере TGC, GCG, CGT и GTG има број појављивања два, и свака од шест 3-мера ATG, TGG, GGC, GCA, CAA и AAT има број појављивања један, креираћемо допунски граф. Даље, граф настао додавањем многоструких ивица је балансиран (и самим тим садржи Ојлеров циклус), јер су улазни и излазни степен сваког чвора једнаки броју понављања *(k-1)*-мера који се појављују у геному. У пракси, утврђивање информације о броју понављања *k*-мера у геному коришћењем постојећих технологија за секвенцирање, може бити веома компликована. Ипак, научници су пронашли начине за реконструкцију генома, чак и када су ови подаци недоступни. Једна таква техника користи упарена читања. Читања се генеришу у паровима секвенционисањем оба краја дугачког фрагмента ДНК чија се дужина може одредити. Уколико упарено читање представимо тако да један пролаз буде улазна грана једног чвора, а други пролаз излазна грана чвора, прочитани пар се може користити за одређивање коректности проласка кроз граф.

Руковање вишеструким и линеарним хромозомима

Код линеарног хромозома морамо да пронађемо Ојлеров пут, уместо Ојлеровог циклуса. Ојлеров пут се не мора завршити у чвору у коме је започет. Уколико имамо вишеструке линеарне хромозоме, имали бисмо пут за сваки од њих.

Руковање несеквенционисаним регионима

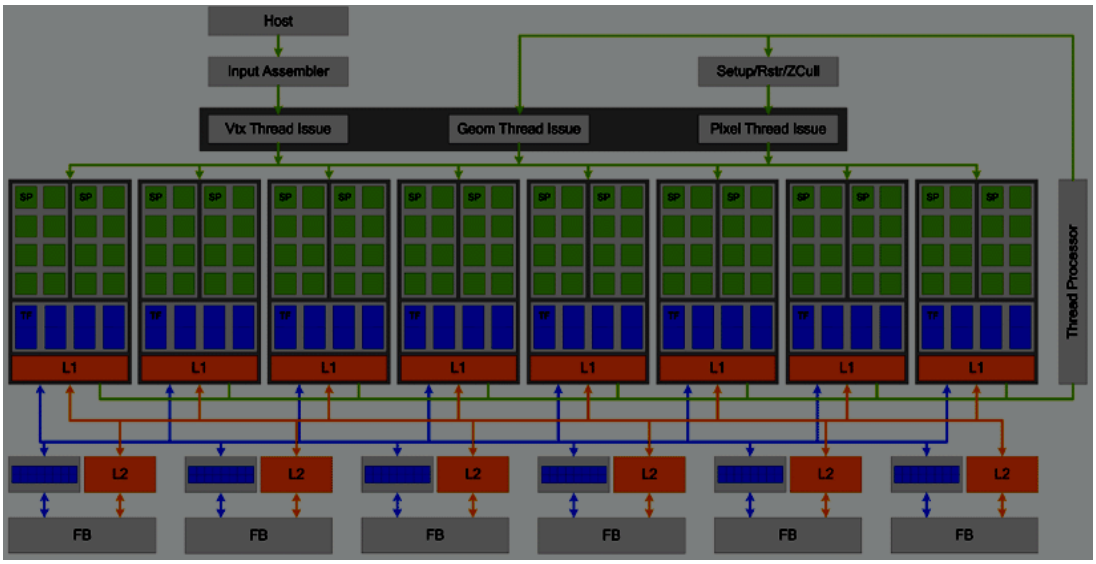
Региони који нису секвенционисани и грешке приликом секвенционисања могу додатно поделити хромозоме у повезане секвенце (секвенционисане регионе ДНК) и празнине (несеквенционисане регионе), са по једном путањом за сваку повезану секвенцу. Повећањем вредности *k* смањује се број грана и праве дугачке повезане секвенце на местима са великом покривеношћу, али се праве грешке. Чак и тако, повезане секвенце ће се разбијати у регије које имају мању покривеност. Сукцесивне повезане секвенце поред хромозома могу имати преклапања мања од *k* нуклеотида, или могу имати празнине између њих. Исправан редослед и орјентација повезаних секвенци, као и приближна величина празнина, одређују се у фази склапања. Ове фаза користи додатне информације, укључујући упарено читање, за одређивање редоследа повезаних секвенци.

3. Масовно паралелно израчунавање

Масовно паралелно израчунавање је форма израчунавања где се велики број израчунавања извршава упоредо. Главна идеја је велики проблем разбити на мање проблеме и онда мање проблеме решавати упоредо.

3.1. Архитектура графичких процесора

Као што је речено у уводу данашње графичке процесоре карактерише масовно паралелизована архитектура. На слици (Слика 6) приказана је концептуална шема модерног графичког процесора.

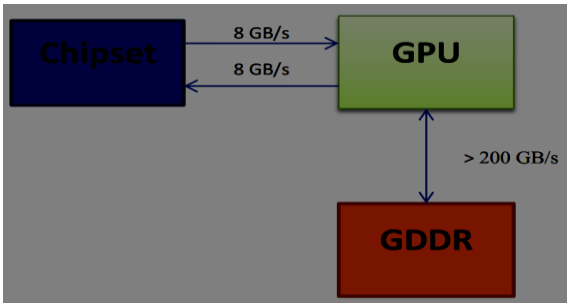


Слика 6. Шема модерног графичког процесора

Висока паралелизација графичког процесора остварена је великим бројем једноставних процесорских језгара (енгл. *Streaming Processors*). Ради једноставнијих контролних механизама и приступа меморији, језгра су груписана у веће скупове који се називају мултипроцесори (енгл. *Streaming Multiprocessors*). Свако процесорско језгро састоји се обично од једне јединице за сабирање и множење (*MAD*) и једне јединице која изводи само множење *(MUL*). Када се узме у обзир да данашње модерне графичке картице из више сегмената имају преко 3072 таквих језгара, која раде на фреквенцијама преко једног гигахерца, једноставном рачуницом може се доћи до бројке од 6 тера флопса (енгл. *Tera Floating Point Operations Per Second*), односно 6 билиона операција са помичним зарезом у секунди. Таква рачунарска снага се раније једино могла пронаћи у врло скупим супер рачунарима.

Сва та расположива снага не би била од велике користи без података над којима би се она користила. Сви подаци који ће се користити у прорачунима чувају се у главној графичкој меморији (*global memory*) чији капацитет иде и до четири гигабајта. Будући да је ситем јак колико и најслабија карика у систему, јако је битно остварити брз приступ подацима у меморији. Због тога се користе брзе *DDR* (енгл.

Double Data Rate) меморије фреквенције преко 6 гигахерца, које имају посебну ознаку *GDDR* (*Graphics Double Data Rate*), са јако широким магистралама до 384 бита. Резултат пропусности иде преко 336 гигабајта у секунди. Наравно, то је теоретски максимум која се постиже оптималним приступом меморији. Под оптималним приступом се подразумева читање података из суседних меморијских локација. У случају насумичног читања из меморије, пропусност знатно опада. Подаци се пре обраде морају на неки начин сместити у графичкој меморији. То се остварује преко *PCI Express 3.0* прикључка, који има пропусну моћ реда 8 гигабајта по секунди у оба смера (16 гигабајта код верзије 4). На први поглед то може да се чини као уско грло, али подаци се преко њега преносе релативно ређе него преко комуникације између графичке меморије и графичког процесора, те због тога и не представља посебан проблем. Комуникација са графичким процесором је приказана на слици (Слика 7).



Слика 7. Шема комуникације према графичком процесору и брзине комуникације

3.2. Технологија *CUDA*

Идеја коришћења графичких процесора у сврху обраде општих података је постојала и пре појаве архитектуре *CUDA*. Највећи проблем који је спречавао ширу употребу је било ограничење комуникације са графичким процесором преко графичких програмских интерфејса *OpenGL, DirectX*. То је доносило са собом бројне проблеме. Резултате обраде није било могуће вратити у неком прикладном нумеричком облику, јер резултат обраде преко графичких интерфејса је био низ пиксела, односно слика. Велики напредак у *GPGPU* у виду увођења подршке за израчунавање општа намене на графичким процесорским јединицама догодио се 2006. године представљањем *CUDA* архитектуре. Део графичке процесорске јединице је био креиран тако да омогућава да се на њему извршавају делови програмског кода, и тиме је олакшано програмирање графичких процесора. Уведен је посебан хардверски додатак и развијани су сви потребни алати за ефикасно паралелно програмирање, односно извршавање на графичким процесорским јединицама. Од 2006. године до данас је продат огроман број графичких картица са *CUDA* архитектуром.

Технологија *CUDA (*енгл. *Compute Unified Device Architecture)* је платформа за паралелну обраду података и модел програмирања развијен од стране произвођача графичких картица *Nvidia,* да би се омогућила паралелизација на самим графичким картицама. Ова технологија је применљива на процесорским јединицама које се налазе на неким моделима графичких картица фирме *Nvidia*, на свим новијим моделима графичких картицама ове фирме налази се подршка. Ова технологија омогућава да процесори који су се раније користили само за приказивање слика на екранима сада могу да се користе за израчунавања, као што

се користи и централна процесорска јединица. Она обухвата графичке процесорске јединице са својом архитектуром, модел програмирања и готове библиотеке које помажу приликом развијања и писања апликација.

Ова технологија се највише развијала захваљујући играма које имају велико тржиште и које захтевају све више израчунавања података због комплексности и квалитета слике која се играчима приказује приликом играња видео игара. Сама технологија није развијана само због подршке за видео игре, већ је развијана и због примене у такозваним „Тесла серверским станицама“. Тесла серверске станице су моћни рачунари који главну обраду података врше на графичким процесорским јединицама. Модуле за Тесла сервере такође развија фирма „Енвидија“ (енгл. *Nvidia*) и у њих уграђује своје чипове развијене за графичке процесорске јединице, само што су модификовани за Тесла сервере и носе ознаку *Тесла*.

Захваљујући великим перформансама које пружају Тесла сервери и графичке процесорске јединице које подржавају технологију *CUDA*, *она* је нашла примену у бројним областима:

- Биоинформатика и природне науке
- Рачунарска хемија
- Рачунарски електромагнетизам
- Рачунарска електродинамика
- Рачунарска финансија
- Динамика флуида
- Рачунарство
- Масовна обрада података
- Обрада слика и рачунарска визија
- Молекуларна динамика
- Рачунарске симулације
- Анализа и обрада базе података
- Моделирање објеката
- Симулације протока гаса
- И многе друге

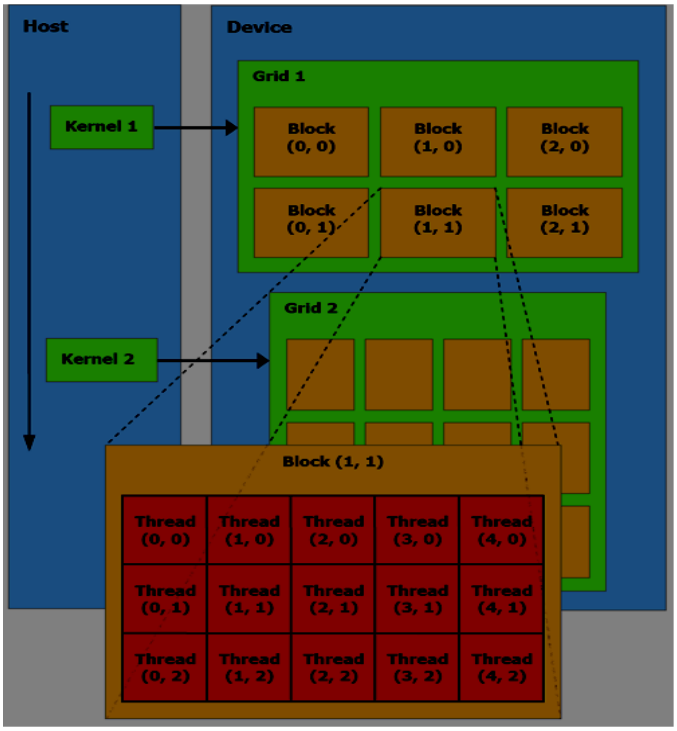
3.2.2. Логичка структура

Код израде апликација са паралелним извршавањем коришћењем технологије *CUDA* могуће је разликовати две врсте програмског кода:

1. Код који ће се извршавати на централној процесорској јединици (*host code*)
2. Код који ће се извршавати на графичкој процесорској јединици (*device code*)

Основна идеја код дизајнирања система је да се део посла који се мора секвенцијално извршавати прилагоди за извршавање на централној процесорској јединици, а да се део посла који укључује извршавање основних аритметичких операција над великом количином података истовремено прилагоди за извршавање над графичком процесорском јединицом.

Програмски код који се извршава на графичком процесору састоји се од низа функција које се називају језгра (*Kernel*). Приликом позива таквих функција све створене нити извршавају исте инструкције, али над различитим подацима што је карактеристично за *SIMD* (енгл. **Single instruction, multiple data**) архитектуру. Податке које нит користи одређују идентификатори *BlockID i ThreadID*. Све створене нити су организоване у два нивоа. На нижем нивоу налазе се блокови (*Grid*). Блокови су скупови нити које могу директно комуницирати преко заједничке меморије (engl. Shared memory). На вишем нивоу налази се мрежа блокова. Нити које се налазе у одвојеним блоковима могу комуницирати искључиво преко главне графичке меморије (*Global memory*). Логичка структура приказана је на слици (Слика 8).



Слика 8. Логичка организација графичког процесора.

Главни разлози овакве логичке структуре су перформансе. На овај начин максимизира се искоришћеност процесорских језгара, односно минимизира се време које језгра проводе чекајући на обављање неке високо латенте операције попут приступа меморији. У тренутку извршавања неког језгра сваки блок нити се дели на скупове од по 32 нити. Ти скупови се називају ворпови (енгл. *warp*). Сваки ворп се може директно изводити на једном мултипрецесору. У сврху постизања максималних перформанси, број нити у једном блоку би требао бити дељив са 32, јер у супротном ворп није потпун и неће у потпуности искористити сва расположива процесорска језгра.

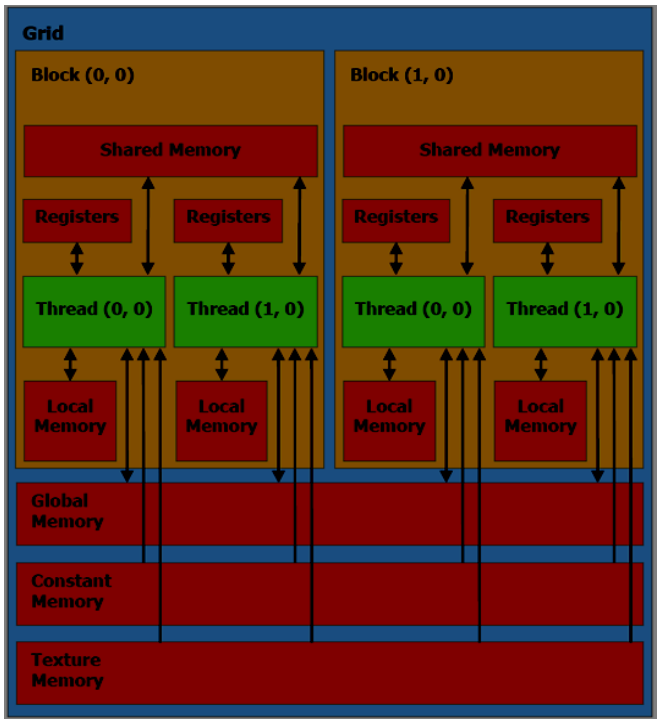
3.2.3. Меморијска хијеарархија

Код чувања података кориснику су на располагању следеће меморијске локације (наведени по брзини приступа у опадајућем редоследу)

1. Регистри
2. Дељена меморија

3. Меморија за константе
4. Меморија за текстуре
5. Глобална меморија

На слици (Слика 9) приказана је шема меморијске хиеарархије.



Слика 9. Шема меморијске хиеарархије

Кључ за постизање високих перформанси је ефикасно коришћење расположиве меморије. Глобална меморија има велики капацитет, али врло дуго време приступа. У идеалном случају користи се за иницијални пренос података и за враћање резултата обраде.

Меморија за константе део је глобалне меморије који је посебно индексиран да омогућава бржи приступ (искључиво читање). Величина је ограничена на 4 килобајта по мрежи блокова, а унос података могућ је само од стране централног процесора.

Још један тип меморије, који се може само читати од стране графичког процесора, су текстуре. Текстуре по својим својствима представљају комбинацију глобалне меморије, јер имају велики капацитет, и меморије за константе, јер им је индексиран приступ и искључиво су за читање. Индексирање је изведено на посебан начин како би се убрзао локални приступ меморији.

Дељена меморија дефинисана је на нивоу блокова и врло је ограничене величине до 48 килобајта по блоку, али омогућава бржи приступ подацима (око 3-4 циклуса). Ефикасније руковање дељеном меморијом је нужно како би се заобишло ограничење меморијске пропусности.

Задња, уједно и најбржа, меморијска локација јесу регистри. Регистри су дефинисани на нивоу блокова нити, и укупно их има 65536 по блоку. Расположиви

регистри се деле на једнаке делове међу нитима у блоку. У случају да нити користе више регистарских локација него што би смеле, смањује се број расположивих ворпова који се могу извршавати на једном мултипроцесору. Сви једноставни типови података (*char, int, float...*) складиште се у регистре, што омогућава врло брз приступ подацима (1 циклус). Сложени типови података (низови, ниске ...), ако није другачије назначено, смештају се у глобалну меморију, те о томе треба водити рачуна приликом програмирања.

3.3. Развојно окружење

За раду у технологији *CUDA* потребнан је хардвер, односно рачунар са графичком картицом са *CUDA* архитектуром. За покретање апликација неопходне су графичке картице са више од 256 мегабајта графичке меморије. Са мање меморије такође је могуће покренути неке апликације са једноставнијим операцијама над малом количином података.

Затим је неопходно преузети са Енвидијине интернет презентације драјвер програме за *CUDA* архитектуру и *CUDA* развојне алате (енгл. *CUDA Toolkit*) који укључују потребне преводиоце и разне математичке библиотеке и још неке додатне алате. За тестирање поста постављеног окружења могуће је покренути примере који долазе са *CUDA SDK* (енгл. *Software Developer Kit*).

Развој *CUDA* апликација врши на програмском језику *C* са специфичним проширењима за *CUDA*. Развој је могућ у било ком интегрисаном развојном окружењу за језик *C*.

Приликом израде апликативног дела рада коришћени су следећи алати и њихове допуне које су у међувремену излазиле.

1. ***Microsoft - Visual Studio 2012 Professional***. Ово развојно окружење је лиценцирано преко Мајкрософтовог програма подршке за студенте. Оно омогућава развој програма писаног у "*C*" програмском језику, компајлирање и дебаговање програма.

2. ***Nvidia - Nsight Visual Studio Edition*** - Алат направљен од стране Енвидије као подршка за мајкрософтово развојно окружење *Visual Studio*. Овај алат омогућава коришћење библиотека кода писаних за подршку над Енвидијиним графичким процесорима који омогућавају извршавање *CUDA* програма. Овај алат такође омогућава превођење *CUDA* програма и њихово дебаговање.

4. Паралелна имплементација препознавања чворова Де Бројновог графа

Улазне секвенце представљене су као низови нуклеотида. Као такве, сваки нуклеотид се може представити са два бита, а свака *k*-мера може се представити са *2*k* бита. У зависности од узорка и величине генома минимална граница за дужину *k*-мере иде од 16 до 27 да би се смањила вероватноћа случајног проналажења узастопно поновљених *k*-мера. То можемо искористити као погодност за представљање, јер сваку *k*-меру можемо представити као 64-битни податак (unsigned long long), ако узмемо да нам је горња граница за дужину *k*-мере 32.

Секвенцијални алгоритам

Вредност *k*-мере иницијално се постави на 0. Издвајање прве *k*-мере се своди на узимање нуклеотида редом из секвенце. Сваки нуклеотид претвара се у број (А у 0, Т у 1, Ц у 2, Г у 3). Затим се ради битовско померање у лево за два места тренутне *k*-мере и на добијен број изврши се битовска операција *или* са нуклеотидом претходно претвореним у број. То се понавља ***k-1*** пута. Тиме смо издвојили прву *k*-меру. Свака следећа *k*-мера издваја се тако што се на претходну *k*-меру понови поступак померања у лево и битовске операције *или* са следећим нуклеотидом из секвенце.

Паралелни алгоритам

Паралелизација секвенцијалног алгоритма постиже се тако што се у више нити извршава секвенцијални алгоритам паралелно над различитим секвенцама за сваку нит.

4.1. Имплементација

Потребно је направити програм који ће из текстуалног фајла, који садржи геномске секвенце, сваку секвенцу раставити на *k*-мере. *K*-мере је потребно да буду фиксне дужине не веће од 32 карактера. Приликом издвајања *k*-мера из секвенци неопходно је податке обрађивати паралелно. За паралелизацију обраде секвенци потребно је користити *CUDA* технологију. Главни циљ је пребројати понављање сваке *k*-мере и резултат сачувати у излазну датотеку.

4.2. Алгоритам

Програм се састоји од:

Од улазних података:

1. Датотека са секвенцама
2. Дужина *k*-мере

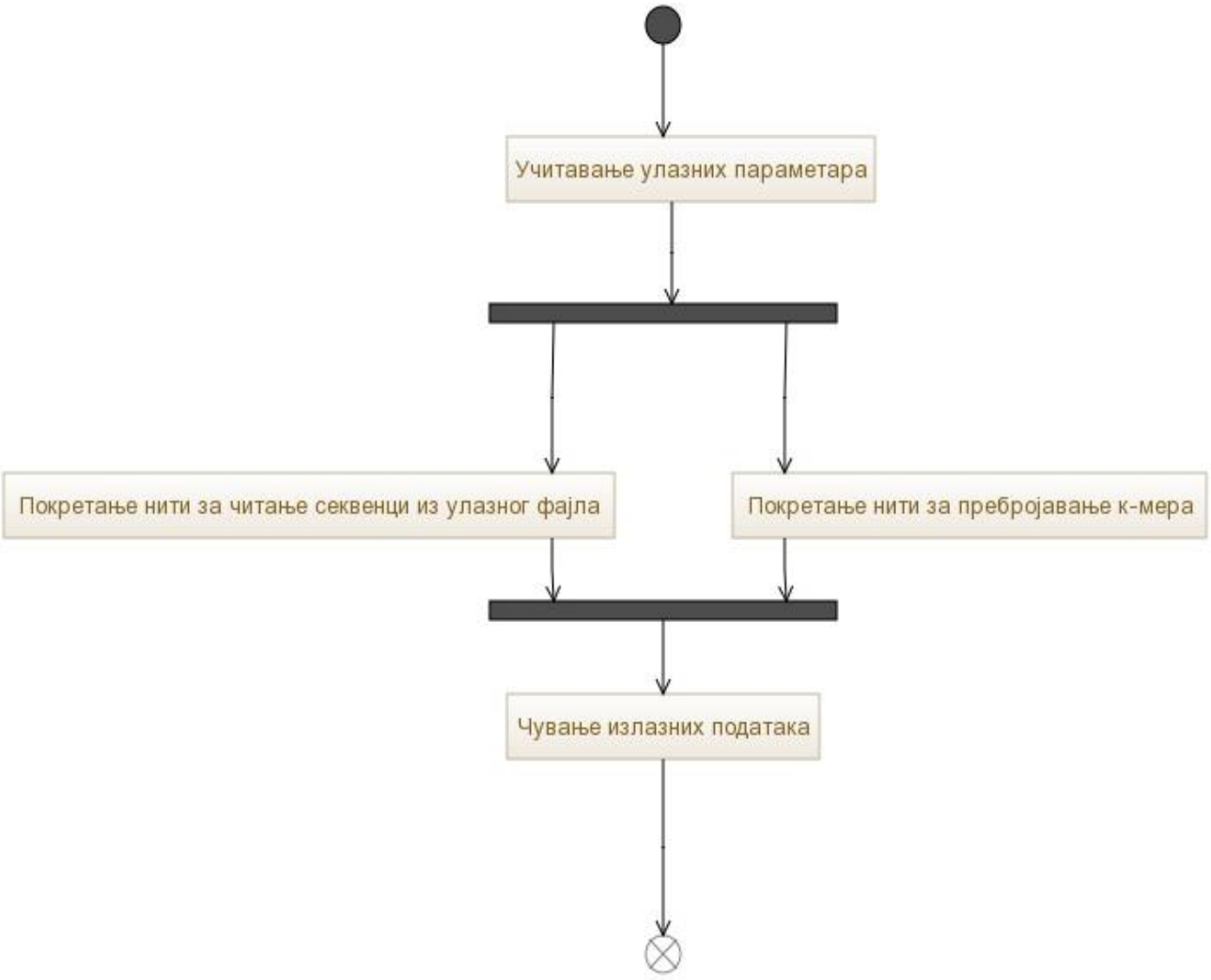
3. Максимална дужина секвенци улазне датотеке – представља максималну могућу дужину секвенце из улазне датотеке.
4. Величина блока за паралелну обраду – одређује број паралелних нити на графичкој процесорској јединици

Излазних података:

1. Листа парова (кључ - вредност) где нам је кључ *k*-мера, а вредност број понављања те *k*-мере у узорку.

Алгоритам се може раставити на четири целине, које се извршавају у две одвојене процесорске нити. Те целине су:

1. Читање секвенци из улазне датотеке и покретање *CUDA* блока
2. Издвајање *k*-мера у *CUDA* блоку
3. Пребројавање броја појављивања *k*-мера
4. Складиштење резултата у излазну датотеку



Слика 10. УМЛ Дијаграм активности - Главна нит

4.2.1.Читање секвенци из улазне датотеке и покретање *CUDA* блока

У овом делу апликације неопходно је прочитати све секвенце из улазне датотеке. Прочитане секвенце је затим неопходно послати на обрађивање на графичку процесорску јединицу, на раздвајање на ***k*-мере.** Секвенце копирамо у меморију графичке процесорске јединице као низ ниски чија је величина једнака величина *CUDA* блока. Како се читање секвенци из фајла и издвајање *k*-мера покрећу из исте процесорске нити, а бројање појављивања *k*-мера се изводи у другој процесорској нити, неопходно је обезбедити конкурентни приступ *k*-мерама између те две нити. За то користимо модификовану структуру *Ред* (енгл. *Queue*) са кантацима да би смо обезбедили конкурентан приступ.

Улаз у алгоритам:

- Текстуална датотека са секвенцама
- Величина блока који ће се извршавати на графичком процесору

Корак читања секвенци из улазне датотеке се одвија на следећи начин:

```
nizSekvenci = prazan niz sekvenci duzine velicina_cuda_bloka
while  (citacFajla.nijeKrajFajla())
begin
        novaSekvenca = citacFajla.procitajSekvencu();
        nizSekvenci.dodajNovuSekvencu(novaSekvenca);
        if (nizSekvenci.brojElemenata() == velicina_cuda_bloka)
        begin
                //ova komanda se izvrsava na GPU
                kopirajNizSekvenciUGPUMemoriju(nizSekvenci)
                cudaIzdvojiKMere(velicina_cuda_bloka);
                listaKMera = kopirajListuKMeraSaGPU();
                konkurentniRed.dodajListuKMera(listaKmera);
        end
end
```

Слика 11. УМЛ Дијаграм активности - Читање улазног фајла

4.2.2. Издвајање *k*-мера у *CUDA* блоку

У овом делу апликације нам је циљ да из сваке секвенце издвојимо *k*-мере. Улаз у овај део програма је низ секвенци чији број је једнак величини блока који ће се паралелно извршавати на ГПЈ.

У овом кораку потребно је сваку секвенцу раставити на *k*-мере и добијене *k*-мере сместити у низ. Растављање на *k*-мере се врши на следећи начин: како имамо секвенце које су редом записане једна за другом у низ карактера, а свака нит ће приступати истом низу каратера, потребно је да за сваку нит одредимо почетак њене секвенце у низу. Када у тренутној нити имамо почетак секвенце која ће се обрађивати у тој нити, онда знамо целу секвенцу, јер знамо почетак и максималну дужину секвенце из датотеке. Затим свака нит у *CUDA* блоку приступа издвајању *k*-мера из секвенце према приложеном алгоритму:

Улаз у програм

- Секвенца
- Максимална дужина секвенце
- Дужина *k*-мере

```
sekvenca = uzmiSekvencuZaTrenutnuNit()
//Niz kmera
lKmera[sekvenca.length – duzinaKMere + 1]
kmeraBr = 0;
kodKMere = 0;
for (i =0 ; i < sekvenca.length ; i++)
begin
   kodBaze = prebaciUIntVrednost(sekvenca[i])
   if (i  < duzinaKMere)
      lKmera[kmeraBr++] = kodKMere = kodKMere << 2 | kodBaze
   else
      lKmera[kmeraBr++] = kodKMere << 2 | kodBaze
end
```

Код дела програма који се извршава на ГПЈ имамо могућност да део програма покренемо у блоку одређене величине. Та величина нам говори колико ћемо покренути засебних нити које ће паралелно извршавати исти део програма. Свака нит види другу вредност идентификатора нити. Вредности идентификатора иду од 0 до **n-1**, где је **n-1** број нити.

K-мере ћемо записивати као шездесетчетворобитне неозначене бројеве. То је могуће у нашем примеру јер смо узели да дужина *k*-мера не буде већа од 32, односно да је ***к <= 32***, а свако слово А Т Ц или Г представљамо са два бита. Слово А представљамо са 00 бинарно односно 0 декадно, слово Т са 01 бинарно односно

1 декадно, слово Ц са 10 бинарно односно 2 декадно и слово Г са 11 бинарно односно 3 декадно. Тако представљена слова из секвенци сада заузимају само два бита по слову у односу на 4 бита која су заузимала док су била представљена као карактери.

4.2.3. Пребројавање броја појављивања *k*-мера

У овом делу програма нам је циљ да пребројимо *k*-мере које су издвојене у претходном делу програма. ***kMeraBrojPojavljivanja*** је каталог у који смештамо *k*-мере и њихов број појављивања.

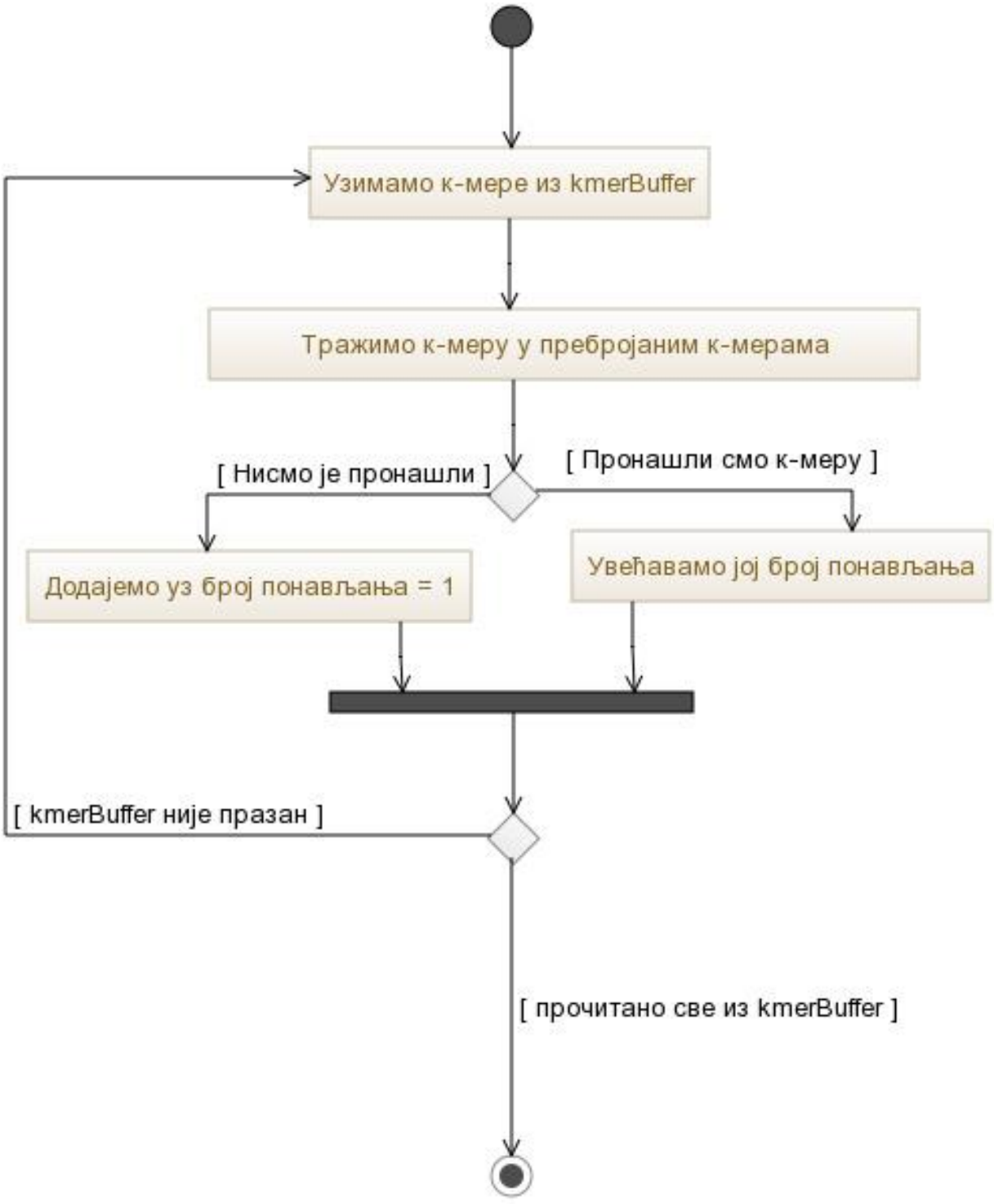
За пребројавање *k*-мера у основи се користи структура каталога. Као кључ се чува *k*-мера записана као шездесетчетворобитни број, а вредност представља број појављивања те *k*-мере. Када први пут *k*-меру додамо у каталог поставимо јој број појављивања на један. Уколико *k*-мера већ постоји у каталогу онда увећамо њен број појављивања за један.

Као каталог коришћена је структура *unordered_map* која се налази у библиотеци *unordered_map*. Како су перформансе каталога знатно опадале када је у њој смештено више од 10^7 елемената морао сам да модификујем структуру. У основи се и даље налази *unordered_map* али сада имамо низ каталога дижине *n*, где је $n = 2^s$. Параметар ***s*** предстаља улазни параметар који је препоручено да буде већи од 12. Овако модификована структура може да обрађује 2^s пута више елемената од претходне, док не дође до тренутка када перформансе знатно опадну.

Структура функционише тако што уместо једног великог каталога имамо ***s*** мањих каталога једнаке величине које се старају о ***s*** пута мањем опсегу *k*-мера. Пошто су нам *k*-мере представљене као бројеви од 0 до 2^{64}, и број појављивања им је равномерно распоређен на целом опсегу (закључио сам на основу узорака) можемо имати следећу политику распоређивања *k*-мера по каталозима:

Како имамо *n* каталога, *i*-ти каталог се стара о *k*-мерама из опсега од $i*2^{64-s}$ до $(i+1)*2^{64-s}$ да памти њихов број појављивања.

```
kMeraBrojPojavljivanja[ ]
while nisuSveSekvenceProcitane or konkurentniRed.nijePrazan()
begin
        listaKMera = konkurentniRed.uzmiListuKmera();
        for kmera in listaKMera
        begin
                if(kmera ne postoji u kMeraBrojPojavljivanja)
                        kMeraBrojPojavljivanja[kmera] = 0
                kMeraBrojPojavljivanja[kmera]++
        end
end
```



Слика 12. УМЛ Дијаграм активности - Пребројавање k-мера

4.2.4. Складиштење резултата у излазну датотеку

Резултате је неопходно сачувати у виду парова (*k*-мера, број појављивања). У овом делу програм мора да прође кроз све парове (*k*-мера, број појављивања) и редом их запише у излазну датотеку.

4.3. Имплементација алгоритма

Апликација за издвајање Де Бруиновог графа је имплементирана да се извршава на *Windows* оперативним системима. Мада је могуће исти програм компајлирати да се извршава на *Linux* оперативном систему. Тренутна верзија програма се покреће из командне линије.

4.3.1. Издвајање *k*-мера

Овај корак се извршава на графичкој процесорској јединици и извршава се паралелно у блоку. У овом контексту блок представља **n** нити које се паралелно извршавају. У нашем случају број нити који се извршава у једном блоку једнак је броју секвенци које ћемо паралелно обрађивати и тај број се задаје као улазни параметар. Што даље имплицира да ћемо имати за сваку секвенцу по једну нит која ће је обрађивати. Функција која се извршава у том блоку, зове се genKElements као улазне параметре ова функција прима:

- char* sekvence
- unsigned long long* kmere
- int kmerLength
- int max_seq_length

Секвенце су записане у фиксној дужини једна за другом. Дужина секвенце задата је као улазни параметар програма.

Променљива *threadIdx* је доступна у свакој нити и њен параметар threadIdx.x јединствено означава нит. Тај параметар иде од 0 до броја нити који се извршава у блоку, на основу тога почетак секвенце у низу је једнак threadIdx.x * *max_seq_length*. Променљива *max_seq_length* представља највећу могућу дужину секвенце. У нашем случају све секвенце су исте дужине и она износи 114, тако да и највећа могућа дужина секвенце износи 114.

Прву *k*-меру генеришемо тако што редом од почетка секвенце узимамо по један карактер и претварамо га у два бита. Та два бита додајемо на крај *k*-мере. К мера је иницијално 0, додавање два бита на крај *k*-мере вршимо тако што радимо битовско померање улево и онда битовско *или* са *k*-мером и та два бита. Овај поступак понављмо ***k-1*** пута и тиме смо издвојили прву *k*-меру. Сваку следећу *k*-меру генеришемо тако што на предходну *k*-меру урадимо битовско померање у лево и затим урадимо битовско или са следећим карактером у секвенци. Ово радимо све док не дођемо до краја секвенце и тиме смо издвојили ***duzina_sekvence - (k - 1)*** *k*-мера. *K*-мере се у низ смештају на следећи начин: пошто знамо да имамо ***duzina_sekvence - (k - 1)*** *k*-мера на крају ће нам излазни низ *k*-мера бити дужине ***(duzina_sekvence - (k - 1)) * broj_cuda_niti***. Свака нит, због тога, *k*-мере које је

издвојила, мора редом да смешта у низ, тако што прва *k*-мера у нити долази на позицију ***duzina_sekvence - (k - 1) * threadIdx.x***. Свака следећа *k*-мера долази на за један већу локацију од претходне.

У наставку је приказан део кода који врши издвајање *k*-мера из секвенци.

```
__global__ void genKElemets(
                char* sek,
                unsigned long long* kmere,
                int kmerLength,
                int max_seq_length)
{
  int startSeq = threadIdx.x * (max_seq_length);//pocetak sekvence
  int startKmera = threadIdx.x * (max_seq_length - kmerLength -1 );
  kmere[0] = 100;
  unsigned long long kmera = 0;
  for(int i=0,kmerCount=0;i<(max_seq_length);i++)
  {
    if(i<kmerLength)
    {
      kmera = kmera << 2 | ZipCharValue(sek[startSeq+i]);
      if(i == kmerLength -1 )
      {
        kmere[startKmera] = kmera;
      }
    }
    else
    {
      kmere[startKmera+(++kmerCount)]
       = kmere[startKmera+kmerCount-1] << 2 | ZipCharValue(sek[startSeq+i]);
    }
  }
}
```

Пример издвајања *k*-мера:

Секвенца:
AATTTTCCCGGATAACCGGGACTTTGGGGGAACCCTTTGGGGGGGGAAACCT
ACAAAAAACCTTACCCAAAAACCCAATTTGGGGGGAACCCAATTTTTTACCA
TTAA

Издвојене *k*-мере

Прва *k*-мера из примера

AATTTTCCCGGATAACCGGGACTTTGGGGGAA

Друга *k*-мера из примера

ATTTTCCCGGATAACCGGGACTTTGGGGGAAC

…

Последња *k*-мера из примера

AATTTGGGGGGAACCCAATTTTTTACCATTAA

Прва *k*-мера бинарно би изгледала

000001010101101010111100010000101011111100100101011111111111110000

Односно, када би се представила као шездесетчетворобитни број у декадном запису

385827712954367984

Друга *k*-мера би бинарно изгледала

000101010110101011111000100001010111111001001010111111111111000010

Односно, као шездесетчетворобитни број у декадном запису

1543310851817471938

4.3.2. Ред са приступом из више нити

Проблем који се природно намеће јесте смештање *k*-мера у одређени део меморије када се прочитају из улазног фајла и пре него што се обраде уз помоћ бројача *k*-мера. Како имамо структуру којој је неопходно приступити из две нити: из читајуће нити и из нити за пребројавање. Потребно је омогућити закључавање тог дела меморије да не би долазило до недозвољеног приступа. Такав приступ би се десио када би обе нити приступиле истој мемориској локацији и мењали је.

Ред са приступом из више нити је реализован уз помоћ катанца и библиотеке *queue.*

Ред се понаша као фифо. Елемент који уђе први у ред из њега ће изаћи први. Елементи који се смештају у ред су показивача на листе *k*-мера.

Приложен је део кода класе *lockedQueue* која врши додавање новог елемента у ред са закључавањем. Променљива *queueMutex* катанац који служи за закључавање *push* и *pop* метода које зову исте те методе над редом *queue* из исте класе.

```
void push(const T& item)
{
  std::unique_lock<std::mutex> mlock(queueMutex);
  queue_.push(item);
  mlock.unlock();
  cond_.notify_one();
}
```

Приложен је део кода који врши узимање првог елемента из листе

```
void pop(T& item)
{
  std::unique_lock<std::mutex> mlock(queueMutex);
  while (queue_.empty())
  {
    cond_.wait(mlock);
  }
  item = queue_.front();
  queue_.pop();
}
```

4.4. Формат записа излазних података

Излазни подаци програма се уписују у бинарну датотеку. Бинарна датотека састоји се од слогова који представљају следећи бинарни запис структуре *element*:

```
struct element{

        unsigned long long k_mera;

        unsigned long count;

}
```

Променљива *k_mera* представља архивиран запис једне *k*-мере представљен шесдесетчетворобитним неозначеним бројем, што ће у датотеци бити записано са 8 бајтова. Променљива ***count*** представља број појављивања дате *k*-мере, који ће у датотеци бити записан са 4 бајта. Тиме добијамо бинарну датотеку у коју су уписани слогови величине 12 бајтова.

5. Резултати и дискусија

У првим верзијама покушано је да се пребројавање *k*-мера изврши уз помоћ каталога имплементираних у *c++*. Како су перформансе знатно опадале са већим бројем елемената, у каталогу за пребројавање, у неком тренутку програм би додавао и по пет минута нову *k*-меру. Ови проблеми су превазиђени модификацијом каталога. Направљена је нова структура која садржи низ каталога од којих се сад сваки од каталога стара о мањем броју *k*-мера, што је објашњено у одељку 4.2.3.

5.1. Брзина извршавања

Мерење брзине извршавања извршено је у односу на број ГПЈ нити, односно величине блока који се извршава на графичкој процесорској јединици у виду *CUDA* блока са истом датотеком која садржи секвенце. Приликом овог мерења део програма, који је вршио складиштење, био је искључен, те време извршавања тог дела програма није утицало на укупно време извршавања из мерења.

Конфигурација рачунара на ком је вршено мерење:

1. Процесор: *Intel i5-2500k 3.6 Ghz*
2. Рам меморија: *DIMM DDR3 4GB 1600MHz Patriot Signatur* x 4 укупно 16GB
3. Графичка картица: *VGA GeForce GTX560 Leadtek 1GB/DDR5*
4. Тврди диск са ког је покретан програм и на ком се налазила датотека са секвенцама *SSD SATA3 250GB Samsung 840 EVO 540/520MB/s*

Улазни подаци:

- Дужина *k*-мера = 31
- Дужина секвенци = 114
- Датотека са секвенцама величине 2,4 гигабајта са 20974189 секвенци
- Број каталога за пребројавање *k*-мера је 8912 (овај параметар не утиче на мерење, јер се део програма за пребројавање *k*-мера не извршава у датом мерењу)

Број ГПЈ нити	Време извршавања у секундама
1	13859
2	7032
4	3782
8	1943
16	1246
32	826
64	572
128	474
256	414
512	309

1024	282

Табела 1. Времена извршавања програма над истим тестним подацима у односу на број нити.

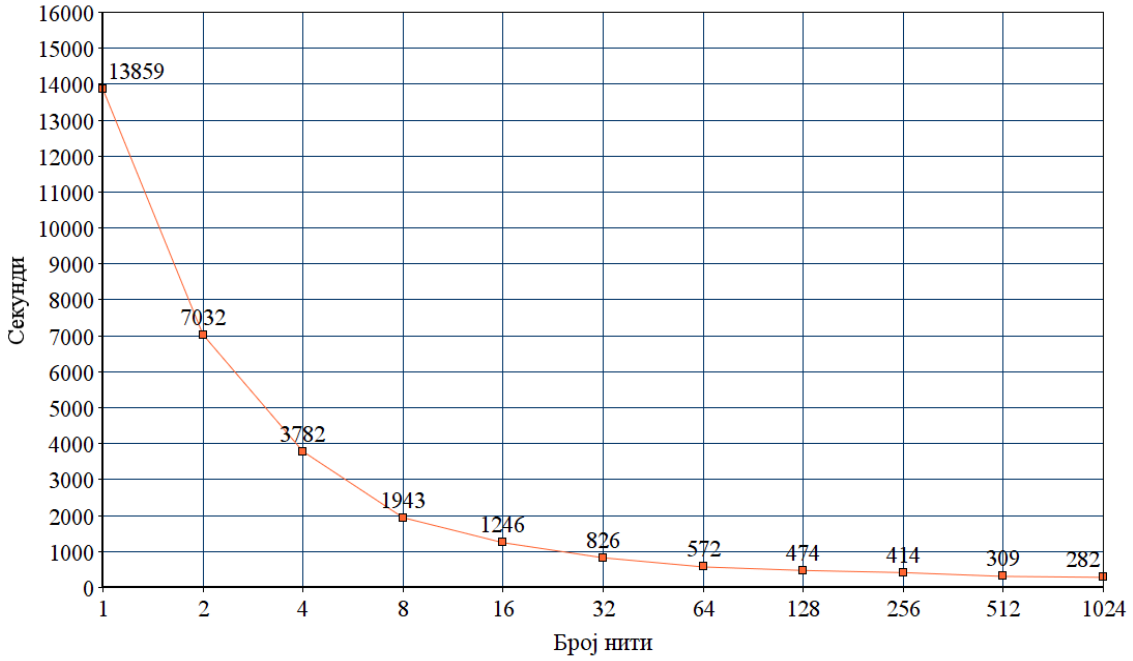


График 1 . Времена извршавања програма над истим тестним подацима у односу на број нити.

5.2. Идеје за даљи рад

Главни проблем код обрађивања геномских секвенци је био превелик број *k*-мера. Самим тим и пребројавање броја појављивања *k*-мера је врло захтевно и изискивало је доста времена да се тај део програма изврши. Проблем је до неке мере превазиђен разбијањем једног каталога, у ком су смештани резултати, на низ каталога. Тиме је у сваки од каталога смештан мањи број *k*-мера, што је до велике мере довело до убрзања програма. Сама идеја да се пребројавање великог броја *k*-мера растави на мање потпроблеме довела је до идеје да се сам проблем може решити ако удружимо више рачунара у решавању тог дела проблема.

Идеја:

Побољшање перформанси бих покушао да постигнем удруживањем више рачунара у кластер да решавају проблем пребројавања *k*-мера. Како је у том делу најуже грло, мислим да би паралелизација тог проблема довела до побољшања перформанси. Међутим, достигнуте перформансе сасвим су довољне.

Проблем пребројавања *k*-мера бих покушао да решим технологијом *CUDA*, односно да пребројавање *k*-мера вршим на графичкој картици.

6. Закључак

У овом раду показано је како је могуће искористити графичке процесорске јединице за прављење Де Бројновог графа коришћењем CUDA модела. Описан је алгоритам за конструисање Де Бројновог графа који се делимично извршава на графичкој процесорској јединици, а делом на централној процесорској јединици. Описана је архитектура графичких процесора који подржавају CUDA, и програмски модел *CUDA* који је коришћен приликом израде.

CUDA је коришћена приликом издвајања *k*-мера из једне секвенце. И на графику *График 1* се види да се програм брже извршава уколико се користи више паралелних нити на ГПЈ. Са растом перформанси графичких картица проблем израде Де Бројновог графа ће бити убрзан у делу где програм ради издвајање *k*-мера на графичком процесору.

7. Референце

1. Интернет извор: *J. Craig Venter Institute, Genome sequencing*, http://www.genomenewsnetwork.org/resources/whats_a_genome/Chp2_1.shtml#chp2#2
2. *Richard C. Deonier, Simon Tavaré, Michael S. Waterman, Computational Genome Analysis, An Introduction, Springer, 1st ed. 2005.*
3. Интернет извор: Википедија, http://en.wikipedia.org/wiki/De_Bruijn_graph
4. *Syed F. Mahmood, Huzefa Rangwala, GPU-Euler: Sequence Assembly using GPGPU,* http://cs.gmu.edu/~tr-admin/papers/GMU-CS-TR-2011-1.pdf
5. Интернет извор: *Rise of the Graphics Processor,* http://ieeexplore.ieee.org/ieee_pilot/articles/96jproc05/96jproc05-blythe/article.html
6. *Phillip E C Compeau, Pavel A Pevzner & Glenn Tesler, How to apply de Bruijn graphs to genome assembly* http://www.nature.com/nbt/journal/v29/n11/pdf/nbt.2023.pdf
7. Интернет извор: Википедија, http://en.wikipedia.org/wiki/Genome
8. *Daniel Robert Zerbino, Genome assembly and comparison using de Bruijn graphs, PhD thesis, University of Cambridge,* September 2009