

**Универзитет у Београду
Математички факултет**

Петар Радовић

**Разматрање могућности примене
технологије *CUDA* за убрзавање рада
са Б-стаблима**

мастер рад

УНИВЕРЗИТЕТ У БЕОГРАДУ/
МАТЕМАТИЧКИ ФАКУЛТЕТ
195
НБ. бр.
БИБАНОВСКА

Београд

2012.

Садржјај

1 Увод	1
1.1 CUDA	1
1.2 Б-stabla	5
2 Методологија	6
3 Имплементација	8
3.1 Б-stabло у системској меморији	8
3.1.1 Бинарна претрага на централној процесорској јединици (CPU)	9
3.1.2 N-арна претрага на графичкој процесорској јединици (GPU)	10
3.1.3 Паралелна линеарна претрага на графичкој процесорској јединици (GPU)	13
3.2 Б-stabло у меморији графичке карте	15
3.2.1 Имплементација Б-stabла заснована на низовима	16
3.2.2 Претрага Б-stabла на централној процесорској јединици (CPU)	20
3.2.3 Претрага Б-stabла на графичкој процесорској јединици (GPU)	21
3.2.4 Вишеструка претрага Б-stabла на графичкој процесорској јединици (GPU)	25
4 Резултати и дискусија.....	28
4.1 Мерење трајања иницијализације језгарне функције на графичком процесору.	28
4.2 Б-stabло у системској меморији	29
4.3 Б-stabло у меморији графичке карте.....	33
5 Закључак	38
6 Референце	40

1 Увод

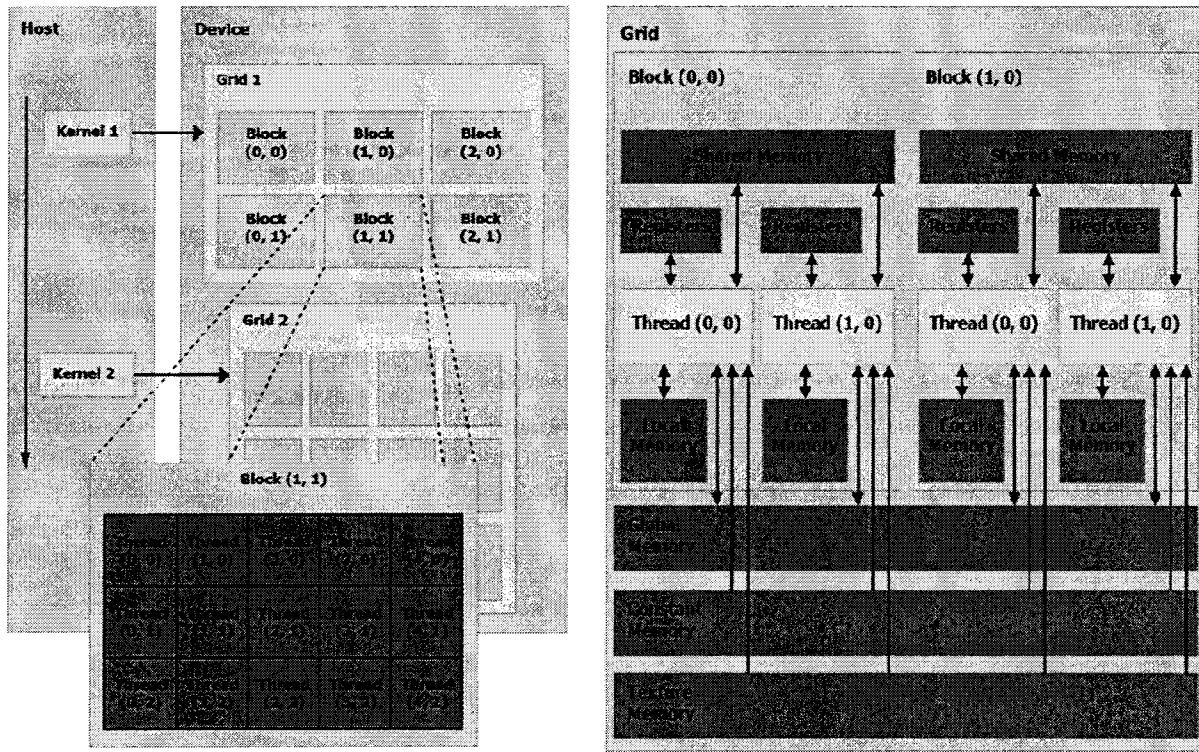
У последњих неколико година све већу популарност добија коришћење графичке процесорске јединице за обављање задатака који се традиционално обављају на централној процесорској јединици (енгл. *General-purpose computing on graphic processing unit*, *GPGPU*). Како графичке карте садрже масивно паралелну архитектуру, оне могу допринети вишеструком убрзању неких задатка. Овакав приступ обично даје добре резултате над проблемима који садрже велики број података над којима се обавља исти задатак. Б-stabla се обично примењују у системима који садрже огромне количине података, као што су базе података и системи фајлова. Исто тако, овакви системи обично за задатак имају да обављају велики број функција одједном, нпр. при извршавању упита над базама података, и то може бити процесорски веома захтеван посао. Из тог разлога, циљ овог рада је покушај убрзања рада са Б-stablima извршавањем задатака на графичкој процесорској јединици.

Што се тиче примене Б-stabla у базама података, већ постоје неки радови у којима се добило убрзање извршавања упита, коришћењем *GPGPU*, као што је [1]. У овом раду су поређени резултати и са другим радовима који су испитивали употребу *GPGPU* за убрзање упита у базама података, али који не користе Б-stabla него структуре посебно направљене за извршавање на графичком процесору. Аутори рада су изнели аргументе и резултате који говоре да су се Б-stabla показала као боља структура података од ових специјалних структура података. У примени Б-stabala као основна операција намеће се претрага Б-stabla. Односно, уколико је претрага Б-stabla спора онда нема смисла разматрати остале операције за одржавање стабла (писање, брисање итд.). Из тог разлога, акценат рада је на убрзању претраге Б-stabla, применом графичког процесора. До краја рада биће приказани различити приступи саме имплементације Б-stabla, као и алгоритама за рад са њима.

За имплементацију на *GPGPU* примењена је технологија *NVIDIA CUDA*. Разлог томе је што је ово тренутно најзрелије комерцијално окружење за програмирање за *GPGPU*, које се може наћи на тржишту. О томе сведочи напредак саме технологије у последњих неколико година. Аутор посебну захвалност дугује господину Сави Живановићу, као и фирмама Технолошко Партнерство, за помоћ коју су пружили у оквиру практичног дела рада, уступањем *NVIDIA* хардвера и несебичном разменом искустава у примени *NVIDIA CUDA* технологије.

1.1 CUDA

CUDA је паралелна рачунарска архитектура, развијена од стране корпорације *NVIDIA*, за графичко процесирање. Исто тако, она је и програмско окружење које омогућава програмерима да производе програмски код који се извршава на графичкој карти, употребом варијанти стандардних програмских језика. *GPGPU* омогућава да графичка процесорска јединица као једну од главних намена има извршавање задатака које традиционално обавља централна процесорска јединица. Ова архитектура не омогућава потпуни паралелизам. Графичка процесорска јединица припада врстама



Слика 1. Хијерархија нити унутар CUDA окружења.

Пример употребе идентификационог броја нити и конфигурисања самог позива језгарне функције представљен је у примеру 1.

```

global_ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    // ...
}

```

Пример 1. Програмски код множења матрица, на графичкој процесорској јединици.

У претходном примеру приказан је позив језгарне функције који извршава $N * N$ нити, унутар једног блока. Ради се о множењу матрица, где свака нит, којој се

- Број нити унутар блока треба да буде производ броја 32, због величине *warp-a*.
- Број нити унутар блока треба да буде минумум 64, и то само уколико се покреће више блокова на једном мулти-процесору
- Обично је најбоље имати између 128-256 нити по блоку
- Больје је користити више мањих блокова него један већи. Ово поготово важи за језгарне функције које имају синхронизацију између нити.

1.2 Б-stabla

Б-stabло је дрвоидна структура података која садржи уређене податке и омогућава претраге, секвенцијални приступ, додавање и брисање података у логаритамском времену [4]. Разликује се од бинарног stabла, тиме што омогућава вишеструко гранање. Заправо Б-stabло обично има велики степен (максимални број деце) чвора, што доводи до велике разгратности самог stabла. Сваки чвор Б-stabla може да има раличит број деце чворова.

Б-stabло спада у групу балансираних stabала. Додавањем и брисањем елемената stabла, stabло може да изгуби своја основна својства. Како се то не би десило, морају се имплементирати функције које балансирају stabло, односно одржавају његове карактеристике. Основне карактеристике Б-stabla су максимални и минимални број кључева које чвор може да садржи. Б-stabla обично садрже много кључева у чвору, како би се смањила њихова висина. Разлог томе је што је једна од њихових основних циљева смањење броја операција читања и писања са секундарног складишта података, при претрази. Ово се постиже тако што се у току претраге у меморију учитава само онај чвор који се тренутно разматра. Додатно, вишеструко гранање над уређеним кључевима омогућава драстично смањење простора претраге у сваком кораку. Односно, претрагом сваког чвора се или налази тражени кључ, или проналази следећи чвор који је потребно учитати у меморију. Број читања и писања са секундарног складишта података током претраге Б-stabla сразмеран је висини stabла. На овај начин су Б-stabla оптимизована за системе који читају и пишу велике блокове података са секундарних складишта података, као што су системи за базе података и системи фајлова.

коришћене су функције за мерење времена из *CUDA API*-а које користе мерач времена графичког процесора.

Да би се мерило време применом мерача времена графичке процесорске јединице, потребно је користити функције из *CUDA API*-а. Унутар ове билбиотеке налазе се функције за прављење (и уништавање) догађаја (*event*), као и за очитавање протеклог времена између њих. Класа која мери време се може имплементирати као што је приказано у примеру 2.

```
void MyCudaTimer::start()
{
    CheckError(cudaEventRecord(d_begin, 0));
    return;
}

void MyCudaTimer::stop()
{
    CheckError(cudaEventRecord(d_end, 0));
    return;
}

float MyCudaTimer::elapsedTime()
{
    // Sync
    CheckError(cudaEventSynchronize(d_end));

    // Elapsed time
    float time;
    CheckError(cudaEventElapsedTime(&time, d_begin, d_end));
    return time;
}
```

Пример 2. Мерач времена, који користи графичку процесорску јединицу.

Ова класа узима у обзир асинхроност позива језгарних функција. Из тог разлога се пре очитавања протеклог времена обавезно извршава синхронизација, односно чека се да се заврше све операције на графичкој картици. На овај начин добијају се исправни подаци о времену извршавања задатака на графичкој процесорској јединици. Иста класа може се користити и за мерење извршавања рачунарске процесорске јединице, и из тог разлога је управо она употребљена за мерење времена свих тестова извршених у оквиру овог рада.

покрене за сваки кључ, и у резултатима теста прикаже просечно време извршавања алгоритма, то је веома добар показатељ перформанси самог алгоритма и његове имплементације. Исто тако, имплементације алгоритама не садрже микрооптимизације, зато што је акценат стављен на коректност и перформансе основне имплементације алгоритама.

Један од битних корака у алгоритму тражења кључа у Б-стаблу је претрага чвора који садржи уређен низ кључева. Како би се искористио масивни паралелизам графичке процесорске јединице, величине чворова у Б-стаблу су у свим тестовима „велике“. Из тог разлога је, за потребе поређења, за централну процесорску јединицу (*CPU*) имплементирана бинарна претрага (сложености $O(\log n)$), где n представља број кључева. За графичку процесорску јединицу (*GPU*) имплементирана је N -арна претрага, као и паралелна верзија линеарне претраге.

3.1.1 Бинарна претрага на централној процесорској јединици (*CPU*)

За централну процесорску јединицу имплементирана је стандардна бинарна претрага. Као улаз се узима низ кључева чвора. Излаз алгоритма је податак о успешности (да ли је кључ пронађен), као и његов индекс у низу, уколико јесте. Додатно, пошто се ради о претрази Б-stabla, уколико кључ није пронађен алгоритам враћа индекс потенцијалног детета чвора где се може налазити кључ. Имплементирана је рекурзивна верзија алгоритма, без микро-оптимизација. За потребе овог рада то је доволично, јер је битно уочити редове величина дужине трајања на централној процесорској јединици (*CPU*), и упоредити га са трајањем извршавања алгоритма који се одвија на графичкој процесорској јединици (*GPU*).

```
int BTreeNode::BinarySearch(int start, int end, int key, bool *found)
{
    // stop condition, return index of potential child
    if(start>end)
        return (start);

    int middle = start + (end-start)/2;

    // if found return the node and index of key
    int *keys = this->Keys();
    if(keys[middle]==key)
    {
        *found=true;
        return middle;
    }

    if(keys[middle]>key)
    {
        return BinarySearch(start, middle-1, key, found);
    }
    else
        return BinarySearch(middle+1, end, key, found);
}
```

Пример 3. Бинарна претрага чвора Б-стабла

случајевима када се тражени кључ потенцијално налази у сегменту који није додељен ниједној нити, био нетачан. Из тог разлога, потребно је да број иницијализованих нити буде већи или једнак броју сегмената. С друге стране, број сегмената зависи од односа величине самог сегмента и укупног броја кључева и чвору. Циљ је имати што већи број малих сегмената који могу да се изврше паралелно. Тест је извршен са разним конфигурацијама и иако разлике нису велике, испало је најбоље имати макар 128 нити у блоку, а затим иницијализовати доволјно блокова да укупан број нити буде довољан да би се алгоритам извршио коректно. Теоријски гледано, ово делује исправно, зато што је са толико нити окупирањост блока на оптималном нивоу (предложен од стране званичне *NVIDIA CUDA* документације), па се велики број нити може извршити паралелно у исто време. Исто тако, уколико се покрене више блокова, постиже се паралелизам на нивоу блокова јер се блокови расподељују на *SM*-ове. Програмски код језгарне функције која извршава овај алгоритам представљен је у примеру 4.

Улаз у алгоритам је низ кључева чвора, као и низ у који се уписује резултат. Резултат се уписује у низ од 4 контролне цифре (целобројне вредности). Свака контролна цифра је иницијализована на -1. У прву цифру се уписује вредност индекса нађеног кључа, уколико је он пронађен. У четврту цифру се уписује индекс потенцијалног детета тренутног чвора, у коме би се могао налазити кључ, уколико кључ није пронађен у тренутном чвору који претражујемо. Друга и трећа контролна цифра служе за случајеве када је кључ који тражимо већи или мањи од свих кључева у чвору. У првом случају се мења друга контролна цифра, док се у другом случају мења трећа контролна цифра.

Као што се може приметити у примеру 4, у 10. линији кода, додата је додатна провера да ли се ради о нити чији редни број превазилази број сегмената или се ради о нити са редним број 1. Ово је потребно, зато што је могуће покренути језгарну функцију са више нити него што је потребно и у том случају би индекси низа кључева којима би те нити приступале биле веће од максималног исправног индекса низа. Зато је потребно такве нити избацити из скупа нити које ће учествовати у претрази. Исто тако, могуће је изабрати доволјно велики сегмент који би обухватио све кључеве чвора. У том случају је потребно да само нит са индексом 0 (прва нит) учествује у претрази. Овакав случај треба избегавати, зато што се онда губи смисао паралелизма графичке процесорске јединице, односно алгоритам би био извршен од стране једне нити, потпуно секвенцијално. При томе, централна процесорска јединица је много боља од графичке у извршавању оваквих задатака (који укључују гранања), што би додатно утицало на лоше перформансе графичке процесорске јединице.

С друге стране, што се изабере мањи сегмент, могуће је иницијализовати више нити да учествују у претрази. Ово би требало да доведе до бољих перформанси графичке процесорске јединице. Међутим, што је сегмент мањи, то се више губи потреба за бинарном претрагом, односно уопште претрагом логаритамске сложености. Наиме, када се смањи сегмент на 1, заправо долазимо до идеје за паралелни алгоритам који ће бити приказан у следећем поглављу.

3.1.3 Паралелна линеарна претрага на графичкој процесорској јединици (GPU)

У претходном поглављу је размотрена паралелна верзија бинарне претраге, зато што је бинарна претрага природан избор алгоритма за централну процесорску јединицу, у случају претраге сортираног низа кључева. Међутим, мора се приметити пар недостатака у имплементацији паралелне верзије овог алгоритма. Први недостатак је што графичка процесорска јединица праву снагу извлачи у позиву великог броја нити одједном, које извршавају неки задатак који има што мање гранања унутар самог секвенцијалног дела алгоритма. Исто тако, алгоритам је сам по себи логаритамске сложености, и редукција простора кључева се дешава веома брзо. Већ у пар итерација тај простор је обично смањен на величину сегмента који је изабран. Из тог разлога се поставља питање колико је заправо снаге графичке процесорске јединице искориштено у оваквом алгоритму. У претходном поглављу, напоменуто је да се број нити повећава како се величина сегмента (партиције) смањује. Уколико смањимо величину сегмента на 1, свака нит би добила задатак да провери једну вредност која одговара том сегменту, и још две вредности пресека суседних сегмената. Међутим, то не би било потребно, зато што је заправо доволно да свака нит провери једну вредност и једну суседну вредност. Прва и последња нит би, у том случају, биле задужене за проверу и граничних вредности (уколико је кључ који се тражи већи или мањи од свих кључева у низу који се претражује). На овај начин би се могло извршавати више нити у исто време. Додатно, број гранања, унутар једне нити, би био знатно смањен.

Улаз у алгоритам би био низ кључева, док би излаз био индекс нађеног кључа или индекс потенцијалног детета који садржи кључ, уколико он није пронађен. Када се врши претрага N кључева, број деце чвора је $N + 1$. Уколико кључ није пронађен у чвору, прво дете може садржати задати кључ у случају да је тражени кључ мањи од свих кључева тренутног чвора. У случају да је тражени кључ већи од свих кључева задатог чвора, онда се такав кључ потенцијално може налазити у $(N + 1)$ -ом детету тренутног чвора. Као и у претходном алгоритму, ово се може проверити од стране централне процесорске јединице, пре покретања језгарне функције. Наиме, потребно је да $N - 1$ нит провери по један кључ, са индексом једнаким редном броју нити, и упише у резултат (у одговарајућу контролну цифру) свој редни број уколико пронађе кључ на том месту. Додатно, уколико не пронађе кључ, у истој нити се проверава да ли је тражени кључ већи од кључа на претходном индексу, а мањи од кључа на посматраном индексу. Уколико је ова провера тачна, нађен је индекс потенцијалног детета чвора које може садржати тражени кључ. У том случају се у резултат (одговарајућу контролну цифру) уписује индекс потенцијалног детета посматраног чвора. Пре свега, додата је додатна провера да ли се ради о нити са најмањим редним бројем. Ова нит само проверава кључ са својим индексом, јер мањи индекс није исправан, а граничне вредности су проверене пре самог позива функције. Обзиром да графичка процесорска јединица, са архитектуром *CUDA*, може да извршава велики број нити у истом времену, преко различитих проточних мулти-процесора, очекује се да овај алгоритам даје боље резултате од претходног, иако се ради о варијацији линеарне претраге. Исто тако, у овај алгоритам би било лакше, уколико је то потребно, убацити и граничне провере (када је тражени кључ мањи или већи од свих кључева у чвору).

паралелно. Једине две нити које немају исти резултат провера су прва нит, или нит која евентуално пронађе тражени кључ, односно потенцијално дете чвора које би могло да садржи тражени кључ. Још једна корисна карактеристика овог алгоритма је што брзина претраге не зависи од расподеле кључева.

3.2 Б-stabло у меморији графичке карте

Као што је приказано, када се Б-stabло налази само у системској меморији извршавање претраге једног чвора, на графичкој процесорској јединици, није донело побољшање у перформансама. Из тог разлога испитана је и могућност похрањивања Б-stabла у меморију графичке карте. То отвара могућност паралелизована читаве претраге и њеног извршавања на графичкој карти. С друге стране, копирање целог стабла у меморију графичке карте се јавља као додатни проблем. Мада је то потребно урадити само једном, на почетку, ипак је потребно имати могућност слања промена на стаблу (брисања и писања) између системске меморије и меморије графичке карте. Поред алгоритама за претрагу, у овом делу ће бити размотрена и специјална имплементација стабла која би одговарала оваквим захтевима.

У разматрању паралелне верзије алгоритма напоменуто је да би се могла размотрити претрага више кључева у исто време на графичкој процесорској јединици. То није имало смисла у ситуацији када је Б-stabло било похрањено у системској меморији. Разлог томе је што је претрага једног чвора само један корак у претрази целог стабла, те би синхронизација разних претрага и резултата претрага појединачних чворова постала превише компликована. Током претраге, правац би се гранао, па би било потребно учитавати разне чворове са дрвета, а показано је да је копирање чвора (у односу на извршавање претраге) веома временски скуп задатак. Међутим, када је Б-stabло похрањено у меморију графичке карте, овакав приступ има смисла зато што се претрага целог стабла врши од стране графичке процесорске јединице. Самим тим, могуће је, и има смисла, покренути претрагу за више кључева одједном.

У овом одељку ће бити размотрена специјална имплементација Б-stabла која би одговарала ситуацији када се оно налази и у меморији графичке карте. Исто тако биће представљено време потребно да се Б-stabло похрани у меморију графичке карте. Поред овога, биће представљен алгоритам за претрагу Б-stabла, који се извршава од стране графичке процесорске јединице. На крају, размотрена је и ситуација претраге више кључева паралелно на графичкој карти. Предпоставка је да, због своје особине масивног паралелизма, графичка процесорска јединица може да победи перформансе рачунарарске процесорске јединице у извршавању оваквог задатка. Резултати ће, као и у предхоном поглављу, бити представљени заједно како би се лакше уочио однос, и направило поређење, перформанси различитих алгоритама.

За извршавање тестова, узет је исти скуп случајних кључева као и за претходне тестове. На исти начин, сваки тест је покренут за сваки могући кључ које стабло садржи, и представљена су просечна времена дужине извршавања одговарајућих функција. Додатно, сви алгоритми су имплементирани тако да су сви кораци јасно

```

class Stablo
{
    int indeksPoslednjegCvora;
    int indeksKorenaStabla;
    Cvor *cvoroviStabla;
    int *indeksiDeceCvorova;
    int *kljucevi;

    /*
        info
    */
}

class Cvor
{
    int indeksCvoraUStablu;

    /*
        info
    */
}

```

Пример 6. Псеудо код класе Чврор и класе Стабло.

У примеру 6 су приказане само основне разлике у односу на стандардну имплементацију Б-стабла. Наиме, интерфејс класа је исти (уз евентуалне одговарајуће промене аргумента и повратних вредности функција). Приметимо да су сада сви чворови инстацирани секвенцијално у меморији унутар једног низа. Исто то важи и за кључеве и индексе деце чворова. Овим се губи потреба за директне показиваче на чворове, већ се уместо показивача њима приступа преко одговарајућег индекса низа у којем се они налазе. Исто тако, простом аритметиком се може доћи до одговарајућег индекса или кључа неког чвора. Нпр, уколико чврор има индекс I, његови кључеви се налазе (у низу кључева стабла) на индексима x за које важе $I * N \leq x < I * N + M$, где N означава максималан број кључева дозвољен у једном чврору, а M означава тренутан број кључева у том чврору. Наиме, уколико је чврор попуњен онда једначина гласи овако $I * N \leq x < (I+1) * N$. На сличан начин индекс чврора детета зависи од индекса самог чврора (унутар низа чворова у стаблу), максималног броја деце чворова унутар чврора и броја кључева у чврору (максималан број деце може бити за један више од броја кључева).

Све методе интерфејса претходне имплеметације Б-стабла су имплементиране и у овом случају. Ово омогућава да се Б-стабло без икаквих проблема иницијализује и попуни од стране централне процесорске јединице. Имплементације самих метода интерфејса, ове и претходне имплеметације, су еквивалентне. Једина разлика је у приступу (и додавању) нових чворова и кључева. На местима где су се користили показивачи у прошлој имплеметацији, у овој се користе индекси одговарајућих низова. Овде постоји мала разлика у перформансама ове и претходне имплеметације.

```

// cuda tree
class CudaTree
{
public:
    int numberOfTreeNodes;
    int device_head_index;
    BTTreeNodeArrayBased *device_treeNodes;
    int *device_childrenIndexes;
    int numberOfChildrens;
    int *device_keys;
    int numberOfKeys;
    int device_numberOfKeysPerNode;
    int device_numberOfChildrensPerNode;
    int treeNodesSize;

    __host__ __device__ void Init(int headIndex,
                                int numberOfChildrensPerNode,
                                int numberOfKeysPerNode,
                                int treeNodesSize,
                                int numberOfChildrens,
                                int numberOfKeys);
    __host__ __device__ void GetHeadIndex(int *headIndex);
};

void CudaTree::Init(int headIndex,
                    int numberOfChildrensPerNode,
                    int numberOfKeysPerNode,
                    int treeNodesSize,
                    int numberOfChildrens,
                    int numberOfKeys)
{
    // allocation
    this->device_head_index = headIndex;
    this->device_numberOfChildrensPerNode = numberOfChildrensPerNode;
    this->device_numberOfKeysPerNode = numberOfKeysPerNode;
    this->treeNodesSize = treeNodesSize;
    this->numberOfKeys = numberOfKeys;
    this->numberOfChildrens = numberOfChildrens;

    CheckError( cudaMalloc((void**) &device_treeNodes, treeNodesSize));
    CheckError( cudaMalloc((void**) &device_childrenIndexes,
                           numberOfChildrens * sizeof(int)));
    CheckError( cudaMalloc((void**) &device_keys,
                           numberOfKeys * sizeof(int)));
}

```

Пример 7. Програмски код класе која садржи Б-stabло у меморији графичке карте

CUDA C садржи библиотеку функција за рад са меморијом, које чине иницијализацију и копирање Б-stabла у меморију графичке картице веома једноставном. То се може постићи библиотечком функцијом *cudamemcpuy*, којој се може предати низ које се копира као и дестинација, односно низ у који се копира први низ. Стандардна *CUDA* документација препоручује да се број позива ове функције сведе на минимум. Као што је већ поменуто, много је брже позвати ову функцију за један велики меморијски сегмент, него је позвати више пута за неколико малих

тренутни чвр лист стабла. Чвр је лист уколико нема потомака. Ако је чвр лист, алгоритам је прошао све релевантне чврове и враћа неуспех. Уколико чвр није лист, иста функција се позива рекурзивно за чвр дете чији је индекс враћен од стране функције претраге чвра. Значи, критеријуми заустављања алгоритма су или успех функције претраге једног чвра, или наилажење на лист услед неуспеха те претраге. Може се приметити да је број потенцијалних претраживања чврова једнак висини стабла. Како Б-стабло обично има велики коефицијент гранања (број деце), висина стабла обично није много велика. Ово наводи на идеју да треба покушати паралелизовати претрагу унутар једног чвра уколико је то могуће, на сличан као што је већ урађено у претходним поглављима.

```

bool BTeeArrayBased::BTeeSearch(int rootNode,
                                int key,
                                int *resultNode,
                                int *index)
{
    bool found=false;
    *index=this->BinarySearchNode(rootNode, key, &found);

    if(found)
    {
        *resultNode=rootNode;
        return true;
    }

    BTeeNodeArrayBased *root = &this->treeNodes[rootNode];

    if(root->IsLeaf())
    {
        return false;
    }

    else
    {
        return BTeeSearch(root->GetChildTreeIndex(*index, this),
                          key,
                          resultNode,
                          index);
    }
}

```

Пример 9. Програмски код претраге Б-стабла на централној процесорској јединици.

3.2.3 Претрага Б-стабла на графичкој процесорској јединици (*GPU*)

Као основа за паралелни алгоритам претраге Б-стабла, користи се стандардни алгоритам претраге Б-стабла описан у претходном поглављу. Ако се изанализира претходни алгоритам примећује се да је један важан корак могуће паралелизовати. То је претрага чвра. У претходним поглављима су већ описане N-арна претрага и паралелна верзија линеарне претраге једног чвра. У имплементацији претраге Б-

```

__device__ __host__ void CudaParallelSearch(int tid,
                                            int key,
                                            int *keys,
                                            int start,
                                            int end,
                                            bool *success,
                                            bool *found,
                                            int *keyIndex)
{
    if(tid <= end - start + 1)
    {
        if(tid == 0)
        {
            if(key <= keys[start])
            {
                sharedPotentialChildNodeIndex = tid;
                *success = true;

                if(key == keys[start])
                {
                    *found = true;
                    *keyIndex = start;
                }
            }

            return;
        }

        if(tid == end - start + 1)
        {
            if(key > keys[end])
            {
                sharedPotentialChildNodeIndex = tid;
                *success = true;
            }

            return;
        }

        int index = tid + start;
        if(keys[index]==key)
        {
            *success = true;
            *keyIndex = index;
            sharedPotentialChildNodeIndex = tid;
            *found = true;
        }
        else if((keys[index]) > key && (keys[index - 1]) < key)
        {
            *keyIndex = -1;
            sharedPotentialChildNodeIndex = tid;
            *success = true;
        }
    }
}

```

Пример 10. Паралелна претрага једног чвора

једног чвора, овај индекс се може уписати у дељену меморију која је бржа од глобалне меморије. Уколико је потребно користити нити више блокова, онда се мора користити глобална меморија, којој могу приступити различити блокови, али која је уједно и много спорија од дељене меморије. Из тог разлога тежња је да се користи конфигурација која би при претрази једног чвора користила нити једног блока. Ово додаје додатно ограничење да чвр не може бити већи од максималног броја нити које је могуће иницијализовати унутар једног блока. На графичкој карти на којој су извршени тестови ова граница је 1024 нити по блоку. С обзиром на то да за величину чвора од 1023, већ дрво висине 2 може садржати преко 1,000,000 кључева, у даљим тестовима ће се разматрати чворови величине до 1023. Из истог разлога језгарна функција, која је приказана у примеру 10, је имплементирана са претпоставком да се за претрагу једног чвора користе нити унутар истог блока.

Кључни корак за коректност ове имплементације је наредба „*syncthreads()*“, којом се поставља баријера на којој нити чекају док све нити не достигну ову тачку у коду. Након тога, једна једина нит је или нашла чвр, или пронашла место детета чвора које може садржати тражени кључ. У другом случају, индекс детета чвора је уписан у заједничку променљиву. Сам упис у заједничку променљиву није потребно синхронизовати (закључати), зато што ће увек само једна нит да уписује у њу индекс. После баријере за синхронизацију следи провера да ли је алгоритам достигао критеријум заустављања. Уколико није, свака нит чита из заједничке променљиве индекс који означава следећи чвр који је потребно посетити. Поставља се питање да ли је потребна још једна баријера синхронизације како би очитани индекс био исправан. Међутим, како увек само једна нит долази до резултата (поставља индекс потенцијалног детета или враћа индекс пронађеног кључа), може се једино десити да неке нити прескоче непотребне кораке, док је резултат увек коректан.

3.2.4 Вишеструка претрага Б-stabla на графичкој процесорској јединици (*GPU*)

С обзиром на то да се претходни алгоритам извршава од стране нити које припадају истом блоку, отвара се могућност претраживања различитих кључева на различitim блоковима. Графичка процесорска јединица је у могућности да извршава више блокова у исто време, у зависности од заузетости меморијских ресурса (регистара, дељене меморије итд.) и броја „*SM*“-а. Из тог разлога размотрена је претрага више кључева одједном, зато што таква претрага још више користи масивни паралелизам графичке процесорске јединице. Исто тако, веома је интересантна могућност коришћења SIMT архитектуре за имплементацију неке врсте класичног (разгранатог) паралелизма. Као што је већ примећено, претходни алгоритам је поприлично „уоквирен“, односно извршава се уоквиру једног блока, и користи само једну заједничку променљиву, која је похрањена у меморију која је заједничка само за један блок. Ово омогућава потпуно природно проширење алгоритма. Наиме, потребно је само променити улазне податке да садрже низ кључева за претрагу, док излазни подаци садрже по један низ за индикаторе успешности одговарајућих претрага и индексе одговарајућих кључева, уколико су пронађени.

Као што се може приметити у програмску коду приказаном у примеру 12, алгоритам за вишеструкту претрагу Б-stabla је скоро потпуно исти као и алгоритам за претрагу једног кључа. Једина разлика је што сада у претрази учествују и нити које припадају различитим блоковима, где је сваки блок задужен за претрагу једног кључа. Синхронизација је и даље потребна само за нити унутар истог блока. Синхронизација између блокова није потребна, зато што различити блокови не користе исте меморијске ресурсе и не учествују у претрагама истих кључева. Једноставно, *CUDA* окружење иницијализује за сваки кључ независну претрагу, и извршава што је могуће више претрага одједном, у зависности од слободних ресурса на хардверу.

Овакав приступ додаје још једно ограничење. Претходни алгоритам је додао ограничење на број кључева у чвору, у зависности од максималног могућег броја нити. То ограничење и даље важи, с тим да је максималан број паралелних претрага једнак максималном броју блокова који се могу иницијализовати од стране *CUDA* окружења. Овај број може да варира у зависности од конкретне графичке карте, и може се променити у будућности. У позиву овакве језгарне функције, на перформансе могу да много да утичу саме могућности графичке карте на којој се језгарна функција извршава, као и конкретна конфигурација позива језгарне функције у односу на распоред нити и блокова. У резултатима ће акценат бити на промени величине стабла и самих чворова стабла, а конфигурације позива ће бити прилагођаване овим величинама. Наравно, у реалној примени морало би се додатно експериментисати, као и направити код који узима у обзир и графичку карту самог рачунара на којима се одвијају језгарне функције.

саме иницијализације и покретања језгарне функције која се извршава на графичком процесору, јер се може испоставити да је у неким задацима то време извршавања велико у односу на само време потребно да се изврши задатак. Зато је, у првом тесту, измерено време извршавања веома просте језгарне функције, која не извршава никакав задатак, Програмски код функције представљен је у примеру 13.

```
__global__ static void DummyCUDA()
{
    // empty
}

int main()
{
    timer.start();
    DummyCUDA<<<N, M>>>();
    timer.stop();
}
```

Пример 13. Проста језгарних функција.

У коду је функција позвана за конфигурацију (N, M), где N означава број блокова који садрже по M нити. Тест је извршен са разним комбинацијама броја блокова и броја нити. Дошло се до закључка да сама конфигурација не утиче много на време саме иницијализације језгарне функције. Међутим, када се језгарна функција позива први пут (поготово ако је то први позив било које језгарне функције од када је покренуто CUDA окружење), иницијализација језгарне функције траје дуже.

Редни број извршавања	1	>1
Просечна дужина трајања(ms)	0.007	0.004

Табела 3. Време дужине иницијализације језгарних функција

4.2 Б-stabло у системској меморији

Резултати сва три алгоритма ће бити представљени и разматрани заједно. Разлог томе је што, у оквиру овог рада, нису битне перформансе ових алгоритама појединачно, него је циљ упоредити перформансе ових алгоритама у оквиру тестног окружења. Додатни разлог је што је битно упоредити перформансе извршавања на

t	64	128	512	1024
Просечно време извршавања (ms)	0.003865	0,003845	0.003910	0.003927

Табела 5. Време извршавања N-арне претраге, за сегмент $s = 5$, изражено у ms. ($2t - 1$) - број кључева у чвору.

Време потребно да се изврши претрага на графичкој процесорској јединици је отприлике 3 пута веће од времена извршавања централне процесорске јединице. Поред тога, на то је још потребно додати време копирања чвора у меморију графичке картице. Међутим, ако се обрати пажња на однос времена извршавања графичке процесорске јединице и времена извршавања празне језгарне функције (која не извршава никакав задатак), може се приметити да су та времена скоро иста. Шта више, може се десити и да време извршавања алгоритма буде брже од времена извршавања празне функције. Разлог томе је што у зависности од претходног извршавања, *CUDA* окружење припрема извршавање следеће језгарне функције (чишћење меморије, реалокација ресурса, итд.). Самим тим, уколико су ова два времена извршавања упоредива, то значи да се заправо сама функција претраге изврши веома брзо (као и на централној процесорској јединици), али је само време покретања језгарне функције веће од читавог извршавања задатка. Поред ових мана, у резултатима се може видети да време претраге на графичкој процесорској јединици више расте у односу на величину чвора, него време претраге на централној процесорској јединици. Разлог томе је вероватно, што бинарна претрага престаје да има смисла на тако малим сегментима који се користе, док се са повећањем величине сегмента смањује њихов број, и губи се смисао паралелне имплементације. Из тог разлога има смисла размотрити и резултате паралелне верзије линеарне претраге, зато што она активира много већи број нити, док је сам програмски код који извршава једна нит много једноставнији и примеренији за извршавање на графичкој процесорској јединици, од бинарне (N-арне) претраге. Резултати ових тестова приказани су у табели 6.

t	64	128	512	1024
Просечно време извршавања (ms)	0.003095	0,003090	0.003177	0.003069

Табела 6. Време извршавања паралелне верзије линеарне претраге, изражено у милисекундама.

Као што се види из приложеног резултата, овај алгоритам се показао боље од претходног, из више разлога. Први, а уједно и најочигледнији је, краће време извршавања. Међутим, још важније од тога је да се овом алгоритму не повећава знатно

овим приступом се не би добило ништа везано за разматрање овог рада, јер је овде потребно претраживати један чвор који је део саме претраге целог стабла. Свакако, ово наводи на идеју претраге целог Б-стабла за више кључева одједном на сличан начин. Такав приступ ће бити испитан у другом делу овог рада. Тренутно је акценат стављен на ситуацији у којој се Б-стабло налази у системској меморији, а не у меморији графичке карте.

Овим је закључено да приступ држања Б-стабла у системској меморији и коришћења графичке карте за претрагу по једног чвора није исплатив. Разлог томе је што брзина саме иницијализације језгарне функције и пребаџивања чвора у меморију графичке карте, превазилази време извршавања претраге једног чвора на централној процесорској јединици. Како је ово најзахтевнији корак алгоритма претраге Б-стабла, закључено је да нема смисла више испитивати алгоритме везане за овај приступ у коме се читаво стабло налази само у системској меморији.

Међутим, било би интересантно направити слично истраживање на другим технологијама за масивно паралелно израчунавање. На пример, нови процесори *AMD-a* имају графичке процесорске јединице интегрисане у централни процесор. Последица овога је уједначенија употреба меморије. Односно, ово би могло да омогући брже копирање чворова или читавог Б-стабла у меморију графичке процесорске јединице, или чак рад са Б-стаблом који би потпуно искључио потребу његовог копирања у меморију графичке процесорске јединице.

4.3 Б-стабло у меморији графичке карте

Пре самог почетка расправе о перформансама самих алгоритама претраге, прво ће бити приказана потребна времена за копирање Б-стабала различитих величине у меморију графичке карте. Сам процес копирања Б-стабла у меморију графичке карте описан је раније у поглављу „Имплементација“, одељку „Б-стабло у системској меморији“.

n\t	64	128	512	1024
1,000	1.15	1.9	6.37	9.55
10,000	6.73	11.5	50.34	93.22

Табела 8. Време копирања и иницијализације Б-стабла у меморију графичке карте. n - број чворова, (2t - 1) - број кључева у чвору.

Сегмент кода садржи прво иницијализацију Б-стабла на графичкој карти, а затим копирање читавог Б-стабла у меморију графичке карте. Време извршавања целог овог сегмента је мерено за различите величине Б-стабла и резултати су приказани у

n\ t	32	256	512
100,000	0.00732	0.00551	0.00643
1,000,000	0.00654	0.00632	0.00653

Табела 10. Времена извршавања претраге Б-stabла, на графичкој процесорској јединици. n - број кључева у stabлу, (2t - 1) - број кључева у чвору.

Из резултата приказаних у табелама 9 и 10, види се да просечно време извршавања претраге Б-stabла не зависи много од његове величине, па чак ни од величине чворова. Перформансе централне процесорске јединице су готово без промена, док се перформансе графичке процесорске јединице мењају, али су веома сличне (истог реда величине, а разликују се за мање од пола микросекунде). Ово нам говори, колико се у ствари претрага Б-stabла, за један кључ, брзо одвија. Чак и у стаблима различите висине, и различите конфигурације, временна извршавања су готово иста. Могла би се урадити додатна истраживања, као на пример упоредити минимуми и максимуми, а не само просечне вредности времена. Међутим, то није толико битно, јер приметићујемо да централна процесорска јединица побеђује перформансе графичке процесорске јединице у овом задатку. Разлика је око 4 - 6 пута, у корист централне процесорске јединице.

Иако су резултати тестова боли на страни централне процесорске јединице, веома је важно што су резултати тестова обе стране упоредиви, односно истог реда величина. Ово чињеница даје наду да се може добити побољшање уколико се покрене претрага више кључева одједном паралелно на страни графичке процесорске јединице. Рачунарска процесорска јединица ће овде једноставно портрошити онолико пута више времена колико се кључева тражи, док са друге стране графичка процесорска јединица може да паралелизује претраге ових кључева. Питање је заправо, колико тачно претрага може да се одвија паралелно на графичкој процесорској јединици и за колике величине чворова. Величине и број претрага овде играју велику улогу, зато што обе ствари утичу на то колико меморијских ресурса је потребно одвојити од стране графичке карте, а самим тим од тога зависи и број паралелних нити које се могу одвијати. Из тог разлога, у табелама 11 и 12, представљамо временна вишеструких претрага и на страни централне процесорске јединице и на страни графичке процесорске јединице, у зависности од величине стабла, величине једног чвора и различитог броја претрага.

оптимизује алгоритам претраге једног чвора и смање ресурси које он заузима. Исто тако, боља графичка картица од ове на којој су рађени тестови (која дефинитивно није професионална, и не би се користила у индустријске сврхе), могла би да покрене много више претрага одједном и за веће чворове. Најважнија је чињеница да графичка процесорска јединица у обављању оваквог задатка даје боље перформансе од централне процесорске јединице. У овом тестном окружењу и са оваквим (уопштеним и не-оптимизованим) језгарним функцијама, највеће убрзање добија се за чвор величине 63 (за који *CUDA* окружење покреће само 2 „*warp-a*“) за претрагу од 40,000 кључева одједном. А за чвор величине 63, перформансе графичке процесорске јединице су за око 9 – 12 пута боље од централне процесорске јединице. Врло важна ствар је да су ово времена која укључују потребан саобраћај између меморија рачунара и графичке карте, што чини ове тестове веома комплетним и блиским потенцијалним реалним ситуацијама.

Сама вишеструка претрага на централној процесорској јединици није паралелно имплементирана, и чињеница је да би таква имплементација дала боље резултате. Теоријске перформансе таквог алгоритма могу бити за онолико пута брже колико има слободних језгара на процесору. С друге стране, на професионалним графичким картама *NVIDIA CUDA*, које су намењене за извршавање оваквих задатака, могу се очекивати далеко боље перформансе. Такве графичке карте, имају много више меморије, брже су, и имају могућност покретања више нити истовремено, у односу на графичку карту која је употребљена у овом тестном окружењу. Исто тако треба узети у да *NVIDIA CUDA* окружење допушта паралелно коришћење више графичких карти за обављање истог задатка, што пружа још један начин да се вишеструко повећају перформансе и могућности самог хардвера.

убрзавања перформанси делова система која користе Б-stabла, као што су на пример базе података или системи фајлова.