UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Anže Škerjanc

Izvedba algoritmov računske geometrije na arhitekturi CUDA

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2013

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00414/2013
Datum: 05.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ANŽE ŠKERJANC**

Naslov: **IZVEDBA ALGORITMOV RAČUNSKE GEOMETRIJE NA ARHITEKTURI CUDA**

IMPLEMENTATION OF COMPUTATIONAL GEOMETRY ALGORITHMS ON CUDA

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

V diplomskem delu predstavite arhitekturo CUDA iz strojnega in iz programerskega zornega kota. Predstavite knjižnice za delo s CUDO ter pristope za reševanje velikih problemov in njihove časovne zahtevnosti.
V nadaljevanju se osredotočite na dva problema računske geometrije -- iskanje najbližjega para ter iskanje konveksne ovojnice danih točk v ravnini. Oba problema predstavite s teoretičnega vidika, nato ju implementirajte za izvajanje na CPE in GPE. Primerjajte hitrosti izvajanj obeh implementacij ter ovrednotite rezultate.

Mentor:

doc. dr. Tomaž Dobravec

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

IZJAVA O AVTORSTVU DOPLOMSKEGA DELA

Spodaj podpisani Anže Škerjanc, z vpisno številko **63100093**, sem avtor diplomskega dela z naslovom:

Izvedba algoritmov računske geometrije na arhitekturi CUDA

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 19. septembra 2013 Podpis avtorja:

Zahvala

Zahvaljujem se mentorju doc. dr. Tomažu Dobravcu za vso strokovno pomoč, nasvete, mentorstvo in vodenje pri izdelavi in pisanju diplomske naloge.

Prav tako gre zahvala tudi mojim staršem in vsem domačim, ki so me podpirali, spodbujali in verjeli vame v času mojega študija.

Kazalo

Kazalo slik:

Povzetek

Cilj diplomske naloge je implementacija nekaterih algoritmov računske geometrije na arhitekturi CUDA. Predstavimo pregled arhitekture CUDA, nekatere knjižnice, pristope reševanja algoritmov in njihove časovne zahtevnosti. Osredotočimo se na algoritma iskanje najbližjega para in iskanje konveksne ovojnice. Prav tako predstavimo prednosti in slabosti izvajanja algoritmov na grafični kartici v primerjavi s centralno procesno enoto. Algoritem iskanja najbližjega para poišče dve točki v množici, ki imata najmanjšo vrednost evklidske razdalje. Algoritem iskanja konveksne ovojnice poišče najkrajšo možno povezavo točk tako, da so vse točke na notranji strani konveksne ovojnice oziroma na njej. Algoritma sta implementirana tako na procesorju kot tudi na grafični kartici. Namen diplomske naloge je podati konkretne ugotovitve o dejavniku pohitritve izvajanja algoritmov na arhitekturi CUDA. Spoznanja temeljijo na praktičnih testih.

Ključne besede:

CUDA, vzporedno procesiranje, algoritmi, računska geometrija, konveksna ovojnica, najbližji par točk

Abstract

The aim of the thesis is implementation of certain algorithms in computational geometry on the CUDA architecture. We present an overview of CUDA architecture, libraries and approaches of solving the algorithms and their time complexity. The main focus of the thesis is on searching (finding) the nearest pair and convex hull. The benefits and disadvantages of the implementation of algorithms on the graphics card against the central processing unit are also introduced. The algorithm search of the closest pair finds two points in the set, that have the nearest value of the Euclidean distance. The algorithm of searching the convex hull finds the shortest possible point connection, so that all points are on the inside of the convex hull or on it. The algorithms were implemented on processor as well as on graphic card. Our set goal in this thesis was to provide concrete answers to the speedup factor in algorithm implementation on CUDA architecture. The answers are solely based on practical performance tests.

Key words:

CUDA, parallel processing, algorithms, computational geometry, convex hull, closest pair of points

Poglavje 1

Uvod in motiv

Arhitekturo CUDA je podjetje NVIDIA predstavilo širši strokovni javnosti februarja leta 2007. Na začetku sicer le za operacijska sistema Windows in Linux, leto kasneje pa še za Mac OS X. Arhitektura CUDA deluje le na grafičnih procesorjih NVIDIA. Po predstavitvi je doživela velik razcvet in vzbudila zanimanje. Podjetju NVIDIA je to takrat prineslo veliko prednost pred konkurenti.

Razvoj programiranja na grafičnih karticah je omogočil hitrejše reševanje splošnih matematičnih problemov v računalništvu in drugačen pogled na njih.

Arhitektura CUDA je najprimernejša za reševanje naslednjih problemov:

- procesiranje slik in videa,
- dinamika delcev,
- izvajanje operacij nad matrikami,
- reševanje fizikalnih problemov.

Velik razcvet med laično javnostjo pa je arhitektura CUDA doživela z drastičnim dvigom cene spletne valute Bitcoin. Ta temelji na decentralizirani, najvarnejši in najcenejši transakciji denarja. S pomočjo grafične kartice je lahko množica uporabnikov učinkovito rudarila Bitcoine [5].

Motiv diplomske naloge je raziskovanje te relativno nove arhitekture. Naš namen je podati korektne ugotovitve o prednostih in slabostih arhitekture CUDA. Spoznanja temeljijo na praktično dokazanih testih.

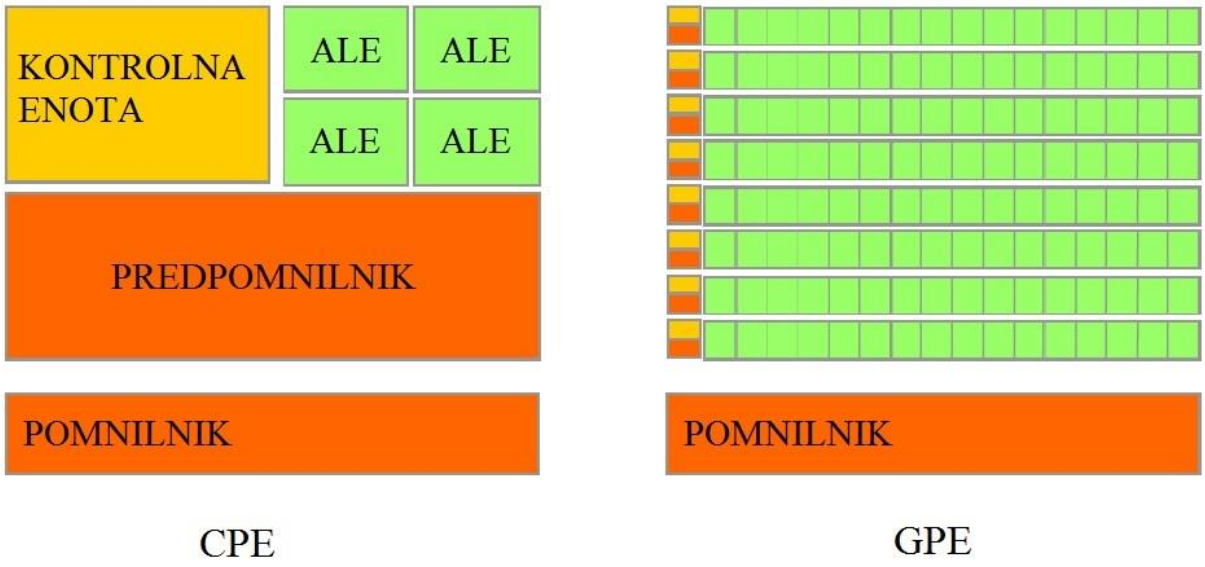
Poglavje 2

Arhitektura CUDA

CUDA je vzporedna programska arhitektura in programsko okolje za vzporedno računanje. Razvilo jo je podjetje NVIDIA in se uporablja za pospeševanje algoritmov z močjo grafične kartice. Je izredno razširljiva arhitektura.

2.1 Razvoj grafičnih kartic

Grafične kartice so zmogljive računske naprave z lastnim procesorjem in pomnilnikom. Arhitektura grafičnih kartic je močno paralelna. Grafična kartica ima v primerjavi s centralno procesno enoto veliko večje število jeder, a so ta jedra počasnejša (slika 1).



Slika 1: Arhitekturna razlika med centralno procesno enoto in grafično kartico [21]

Na začetku so bile to specifične naprave, ki so znale izrisovati le grafiko. Včasih je bilo algoritme možno izvajati le v primeru, ko se je problem lahko pretvoril v računanje z grafiko. Z leti je grafična kartica postala vedno bolj splošna. To je omogočilo razvoj arhitekture CUDA [8].

Veliko hitrejše izvajanje nekaterih algoritmov na grafični kartici je posledica njene zasnove,

ki se razlikuje od centralne procesne enote. Še vedno pa algoritmov ne moremo pospeševati v nedogled le z dodajanjem računskih jeder.

2.2 Amdalov zakon

Amdalov zakon govori o pohitritvi algoritma, če del algoritma poženemo na več procesorjih. Pohitritev se izračuna po enačbi:

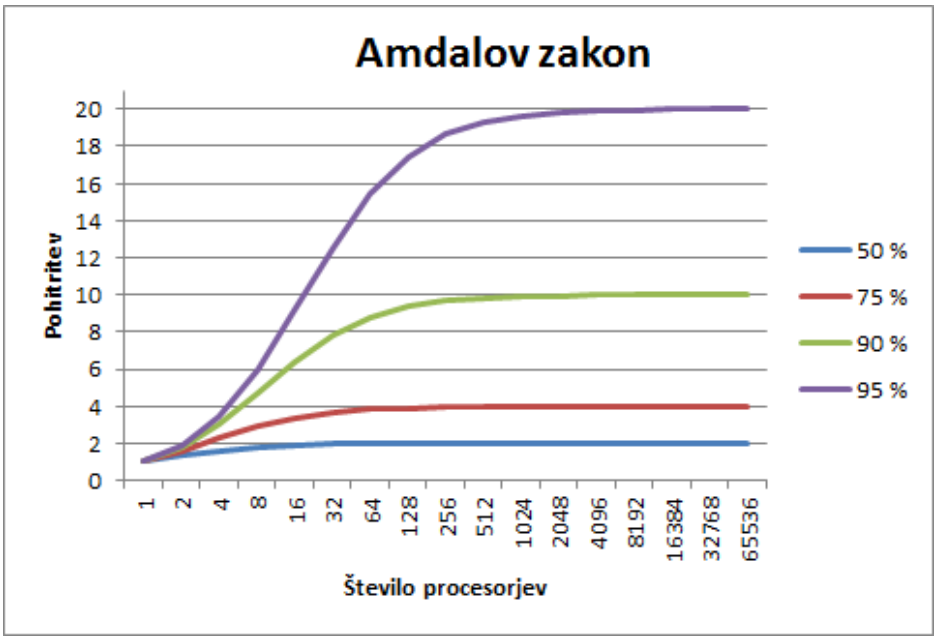
`N` – število procesorjev
`S(N)` – pohitritev algoritma ob poganjanju na N procesorskih jedrih
`f` – delež algoritma, ki se ga ne da paralelizirati
`(1 - f)` – delež, ki ga je možno paralelizirati

$$S(N) = \frac{1}{f + \frac{1-f}{N}} = \frac{N}{1 + (N-1)f}$$

V primeru, da imamo na voljo neskončno procesorjev, se pohitritev izračuna po enačbi:

$$\lim_{n \to \infty} S(N) = \frac{1}{f}$$

Iz enačbe za neskončno procesorjev je razvidno, da v primeru, če velikega dela algoritma ne moremo paralelizirati, ne moremo pričakovati velike pohitritve.



Slika 2: Amdalov graf pohitritve

Iz slike 2 je razvidno, da lahko v primeru, če paraleliziramo le 50 % algoritma, pričakujemo največ dvakratno pohitritev [3].

2.3 Centralna procesna enota

Centralna procesna enota je osrednji del računalnika, ki skrbi za obdelavo podatkov in nadzor drugih enot v računalniku. Po Flynnovi kvalifikaciji spada med procesne naprave SISD (en ukaz, en podatek). Centralna procesna enota ima podprtih veliko več kompleksnih strojnih ukazov, in to z veliko različnimi načini naslavljanja. Velik del tranzistorjev v centralni procesni enoti je namenjen predpomnilniku.

Večina današnjih procesorjev je večjedrnih. Trenutno se na trgu prodajajo procesorji z do osmimi jedri. Kljub večjedrnosti današnjih procesorjev pa števila jeder zaradi arhitekture zasnove procesorja ne moremo učinkovito dvigovati v nedogled [15].

2.3.1 Pomnilniška hierarhija

Velik del procesorja so predpomnilniki. Za hitro delovanje računalnika potrebujemo veliko hitrih predpomnilnikov. Hitrejši predpomnilniki pa pomenijo tudi višjo ceno računalnika. Pomnilniki, ki so bliže procesorju, so hitrejši, a zato manjši.

Pomnilniška hierarhija je navzven za uporabnika nevidna. Centralna procesna enota pomnilniško hierarhijo vidi kot en pomnilnik, ki ima skoraj tako hiter dostopni čas kot pomnilnik, ki je najbližji procesorju, in je tako velik, kot je velik navidezni pomnilnik v zadnjem nivoju.

Vsakič, ko beremo iz pomnilnika, se išče podatek, ki je čim bliže centralno procesni enoti. Ko ga najdemo, njegov blok po potrebi premaknemo vse do prvega nivoja pomnilniške hierarhije.

Pomnilniška hierarhija ima dober učinek zaradi lokalnosti pomnilniških dostopov. Velika verjetnost je, da če bomo dostopali do podatka na neki lokaciji, da bomo v nadaljevanju dostopali tudi do podatkov okoli nje. To velja predvsem zaradi naslednjih dejstev:

- ukaze iz pomnilnikov se jemlje po naraščajočih naslovih,
- zaradi zank v programih,
- podatki so večinoma razdeljeni v polja in strukture.

Slabost pomnilniške hierarhije je, da močno omeji razširljivost procesorja. Problem so vsi predpomnilniki, ki so lastni procesorskim jedrom. Do problema pride v primeru, ko pišemo v predpomnilnik, ki ga ima predpomnjenega tudi drugo jedro in se mora podatek posodobiti tudi v predpomnilniku tistega jedra [22].

2.4 Grafična procesna enota

Grafična procesna enota po Flynnovi kvalifikaciji spada med naprave SIMD. To pomeni, da se en ukaz izvrši na večji količini podatkov. Grafična kartica je sestavljena iz velikega števila procesorskih jeder. V njej je zelo malo predpomnilnikov. Ker podatki niso predpomnjeni na napravi, to omogoča veliko več procesorskih jeder, ki lahko hkrati dostopajo do pomnilnika. Grafična kartica se dobro obnese pri problemih, ki imajo veliko podatkovnega paralelizma. Računanje enega dela problema mora biti čim bolj neodvisno od računanja drugega dela.

2.5 Razvoj arhitekture CUDA

Arhitektura CUDA se je razvila leta 2007. Od takrat naprej se izredno hitro razvija. V nadaljevanju naštevamo nekaj najpomembnejših pridobitev po posameznih verzijah arhitekture CUDA [12]:

- 1.0: prva uradna verzija;
- 1.1: dodane celoštevilčne atomarne operacije za dostop do globalnega pomnilnika;
- 1.2: atomarne operacije so razširjene, da lahko dostopajo do deljenega pomnilnika;

- 1.3: dodana podpora za plavajočo vejico z dvojno natančnostjo;
- 2.x: do sedaj si lahko imel polje niti v bloku le dvodimenzionalno, s to verzijo pa je dodana še tretja dimenzija. S tem je omogočena lažja preslikava podatkov na niti;
- 3.0: v tej verziji je izjemno povečano največje število blokov v vsaki dimenziji;
- 3.5: dodan je dinamični paralelizem, kar omogoča veliko lažje reševanje algoritmov – omogoča nam, da znotraj ščepca v izvajanje pošljemo nov ščepec.

2.6 Knjižnice CUDA

Na področju arhitekture CUDA se je razvilo veliko grafično pospešenih visokonivojskih knjižnic [25]:

- Thrust – splošna knjižnica, katere filozofija temelji na knjižnici STL. Vsebuje veliko vzporednih primitivnih funkcij, kot so najmanjša vrednost, največja vrednost, urejanje, redukcija itd.;
- CUBLAS – za delo z osnovno linearno algebro;
- CUSP – za delo z matrikami in reševanje linearnih sistemov;
- CUFFT – za računanje Fourierjeve transformacije.

2.6.1 Thrust

Thrust je C++ knjižnica, ki uporablja arhitekturo CUDA za pohitritev klasičnih operacij, kot so:

- urejanje,
- redukcija,
- iskanje najmanjšega in največjega elementa
- itd.

Filozofija knjižnice temelji na standardni knjižnici (STL). Thrust je visokonivojska knjižnica, ki nam zakrije podrobnosti arhitekture CUDA in omogoča njeno preprosto uporabo. Knjižnica je spisana samo v zaglavnih datotekah, zato je za uporabo ni treba naložiti. Je izredno razširljiva, saj lahko sami pišemo operatorje, ki jih kliče znotraj svoje implementacije [10].

Osnova hranjenja podatkov v knjižnici Thrust je vektor, ki ima podobne funkcije kot vektor v STL. Thrust pozna dve vrsti vektorjev:

- `host_vector` – podatke hrani v naslovnem prostoru pomnilnika,

- `device_vector` – podatke hrani v naslovnem prostoru grafične kartice.

Prenašanje podatkov med vektorjem naprave in gostiteljem je preprosto z operandom =. Preprosti primer uporabe knjižnice:

```
//generiramo vektor v naslovnem prostoru gostitelja, ki se
//napolni s 100 naključnimi številkami
thrust::host_vector<int> h_vec(100);

std::generate(h_vec.begin(), h_vec.end(), rand);

//Ustvarimo vektor, ki bo hranil podatke na napravi.
//Operator = avtomatsko poskrbi za prenos podatkov na napravo

thrust::device_vector<int> d_vec = h_vec;


//poženemo redukcijo, ki nam vrne vsoto 100 števil
int x = thrust::reduce(d_vec.begin(), d_vec.end(), 0,
     thrust::plus<int>());
```

2.7 CUDA toolkit

Za delo v arhitekturi CUDA moramo najprej naložiti CUDA toolkit. CUDA toolkit je razvijalsko okolje C in C++ za programiranje v arhitekturi CUDA. Cuda toolkit vsebuje:

- prevajalnik,
- matematične knjižnice,
- orodja za razhroščevanje in analiziranje,
- primere kod,
- programske vodiče.

CUDA toolkit se naloži v datotečno strukturo:

- bin – v tej podmapi se nahaja prevajalnik CUDA in dinamične knjižnice, ki jih potrebujemo med izvajanjem programa. V okolju Windows so to datoteke s končnico `dll`;
- include – v tej datoteki se nahajajo zaglavja funkcij, ki jih uporabimo pri programiranju z arhitekturo CUDA;

- lib – tukaj se nahajajo statične knjižnice, ki jih potrebuje prevajalnik za povezovanje programov CUDA. To so datoteke s končnico `lib` v okolju Windows in se pri prevajanju programa vključijo v sam program. Pri dinamično prevedenih knjižnicah je notri le povezava na dinamično knjižnico;
- doc – v tem direktoriju se nahaja dokumentacija. Dobimo veliko dokumentov `pdf` in kopijo spletne strani z dokumentacijo arhitekture CUDA.

2.8 Prednosti in omejitve arhitekture CUDA

Prednosti arhitekture CUDA:

- pohitritev algoritmov,
- razbremenitev centralne procesne enoto, ki lahko ta čas dela kaj drugega.

Omejitve arhitekture CUDA:

- vsi algoritmi na arhitekturi CUDA niso učinkoviti,
- arhitektura ne podpira celotnega standarda C,
- prenašanje podatkov med gostiteljem in napravo je lahko ozko grlo zaradi omejitve vodila,
- deluje le na grafičnih karticah NVIDIA,
- števila s plavajočo vejico niso v celoti podprta po standardu.

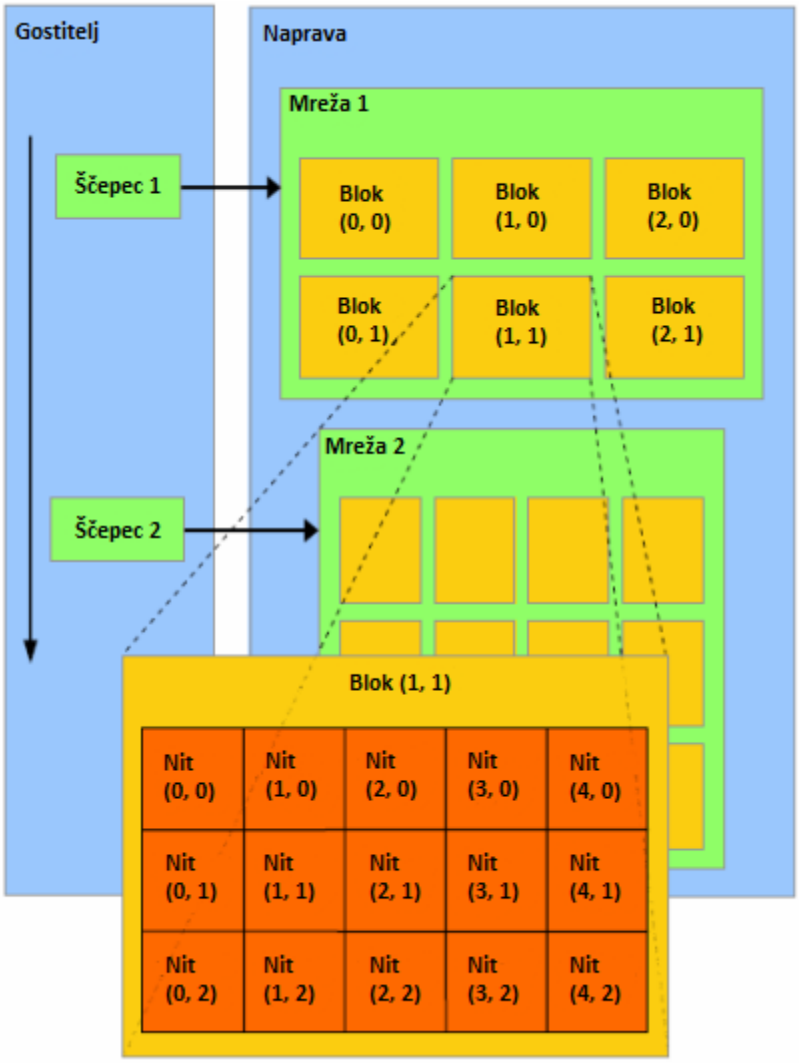
2.9 Izvajalni model

Vzporedno kodo pošljemo na grafično kartico kot ščepec (Kernel). Ščepec je funkcija, ki se poganja na napravi. Naenkrat se na grafični kartici izvaja le en ščepec. Nov ščepec lahko pošljemo na napravo le, če se je dokončal prejšnji. Kodo v ščepcu nato izvaja več niti. Niti se med sabo razlikujejo po številki ID. Ker pri vzporednih arhitekturah velikokrat govorimo o podatkovnem paralelizmu, to pomeni, da se s številko ID večinoma določi, s katerimi podatki naj nit dela.

Za boljšo predstavo si grafično kartico lahko predstavljamo kot napravo, ki zna operirati z neskončno nitmi. Organizacija niti je lahko eno-, dvo- ali trodimenzionalna. Dimenzija razporeditve niti je odvisna predvsem od organizacije podatkov.

Kot je razvidno iz slike 3, so niti združene v blok niti. Znotraj bloka se lahko niti sinhronizirajo. To je edini možni način sinhronizacije med nitmi. Sinhronizacija močno

zmanjša hitrost izvajanja. Bloki se združijo v mrežo (Grid). Trenutno arhitektura CUDA podpira izvajanje, kjer en ščepec izvaja eno mrežo. Spodnja slika prikazuje primer razporeditve niti znotraj ščepca. Niti so tako znotraj bloka kot tudi znotraj mreže razporejene dvodimenzionalno [6].



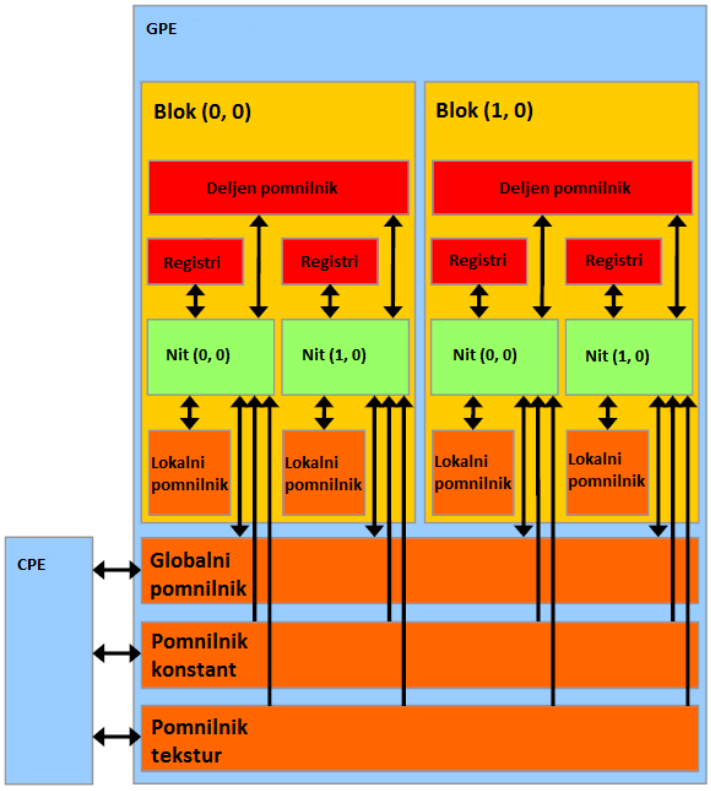
Slika 3: Primer izvajalnega modela arhitekture CUDA [19]

2.10 Pomnilniški model

Grafična kartica ima v primerjavi s centralno procesno enoto izjemno malo predpomnilnikov. Prav to v veliki večini pripomore k tako veliki razširljivosti grafičnih kartic. Arhitektura

CUDA pozna naslednje programsko vidne pomnilniške prostore (slika 4):

- registri,
- deljeni pomnilnik,
- globalni pomnilnik,
- lokalni pomnilnik,
- pomnilnik konstant,
- pomnilnik tekstur.



Slika 4: Pomnilniški model arhitekture CUDA [19]

2.10.1 Registri

Registri so najhitrejši pomnilnik. Dostopen je v eni urini periodi. Registri so lokalni za posamezno nit in jih je zato malo.

2.10.2 Deljeni pomnilnik

Deljeni pomnilnik je hiter in je skupen bloku niti. Prek njega si lahko podatke izmenjujejo niti

znotraj bloka. Lahko ga uporabljamo tudi za predpomnjenje podatkov. To pride v poštev pri podatkih, do katerih niti v bloku pogosto dostopajo.

2.10.3 Globalni pomnilnik

Globalni pomnilnik je počasen, a največji. Uporablja se za branje in pisanje, do njega pa lahko dostopajo vse niti in centralna procesna enota. Na napravi ni nikjer predpomnjen.

2.10.4 Lokalni pomnilnik

Lokalni pomnilnik je lokalen za vsako nit. Uporablja se ga v primerih, če zmanjka prostora v registrih. Preslikan je v globalni pomnilnik, kar pomeni, da je počasen.

2.10.5 Pomnilnik konstant

V pomnilniku konstant se hrani programska koda. Preslikan je v globalni pomnilnik. Na napravi je predpomnjen. Predpomnilnik se briše, ko centralna procesna enota piše v napravo.

2.10.6 Pomnilnik tekstur

Pomnilnik tekstur je preslikan v globalni pomnilnik. Optimiziran je za dostop podatkov, ki so razdeljeni v 2D-polje. Na napravi je predpomnjen.

2.11 Preslikava izvajalnega modela v fizičnega

Za optimalnejše delo z grafično kartico potrebujemo tudi nekaj znanja o njeni dejanski fizični sestavi.

Grafična kartica vsebuje več podatkovno-pretokovnih multiprocesorjev (Streaming multiprocesor). Znotraj njih pa je več podatkovno-pretokovnih procesorjev (Streaming processor). Vsak blok niti se dodeli enemu podatkovno-pretokovnemu multiprocesorju. Od tega vsak vsebuje svoj deljeni pomnilnik in izvajalno enoto, ki skrbi za izvajanje podatkovno-pretokovnih procesorjev. Vsak podatkovno-pretokovni procesor hkrati izvaja eno nit. Ker se izvajalna enota nahaja na nivoju podatkovno-pretokovnih multiprocesorjev, to pomeni, da hkrati vsi podatkovno-pretokovni procesorji izvajajo enak ukaz. S tega vidika sledi, da je grafična kartica naprava z več enotami SIMD (en ukaz, več podatkov) enot. Z vidika arhitekture CUDA je to kar SIMT (en ukaz, več niti). Prav zato mora biti koda v ščepcu čim bolj enaka za vse niti. S tega sledi, da mora biti v optimalno napisanem programu čim manj

vejitev.

2.12 Programiranje v arhitekturi CUDA

Arhitekturo CUDA programiramo v razširjenem jeziku C/C++. Kodo, ki jo želimo prevesti s prevajalnikom NVCC, pišemo v datoteko s končnico `.cu`. Vsa koda, ki ni namenjena grafični kartici, se prevede klasično, torej kot navaden program. Koda, ki se bo izvajala na napravi, pa prevajalnik CUDA prevede v vmesno kodo PTX. S tem dosežemo prenosljivost kode med različnimi grafičnimi karticami.

Klasični program za računanje z grafično kartico je sestavljen iz kode, ki se izvaja na gostitelju, in kode, ki se izvaja na napravi. Katera funkcija se kje izvaja, povemo prevajalniku prek dodatnih oznak:

- `__global__`: funkcijo prevede prevajalnik CUDA in se izvede na napravi. Funkcijo požene centralna procesna enota. Pri novejših grafičnih karticah, ki podpirajo dinamični paralelizem (arhitektura CUDA verzije 3.5), lahko to funkcijo kličemo tudi znotraj ščepca;
- `__device__`: to funkcijo prav tako prevede prevajalnik CUDA. Kličemo jo lahko le iz naprave;
- `__host__`: funkcijo prevede klasični prevajalnik. Kličemo jo lahko le na gostitelju. Privzeto so vse funkcije tega tipa.

2.12.1 Koda gostitelja

Kodo gostitelja izvede centralna procesna enota in je serijska. Ta koda poskrbi za prenos podatkov na napravo. To naredimo tako, da najprej rezerviramo prostor na napravi z ukazom:

```
int *device_array;
cudaError err = cudaMalloc((void**)&device_array,
     sizeof(int)*numOfElements);
```

Funkcija `cudaMalloc` kot prvi argument dobi kazalec na kazalec podatkov, kot drugi argument pa povemo število bajtov, ki jih želimo alocirati. Kazalec je veljaven samo v pomnilniškem prostoru naprave. Funkcija nam vrne kodo napake.

Podatke prenašamo med gostiteljem in napravo z ukazom `cudaMemcpy`. Ukaz prav tako vrne kodo napake. Kot prvi argument navedemo destinacijo kopiranja podatkov, kot drugi

argument vir podatkov in kot tretji argument smer prenosa.

Na voljo imamo naslednje smeri prenosov:

- `cudaMemcpyHostToHost`: gostitelj -> gostitelj. Enako naredi tudi ukaz `memcpy`;
- `cudaMemcpyHostToDevice`: gostitelj -> naprava;
- `cudaMemcpyDeviceToHost`: naprava -> gostitelj;
- `cudaMemcpyDeviceToDevice`: naprava -> naprava;
- `cudaMemcpyDefault`: v primeru, ko si naprave delijo en navidezni pomnilniški prostor, se smer prenosa določi glede na lokacijo, na katero kažejo kazalci. Trenutno to podpirajo le grafične kartice Tesla.

Prenos podatkov na napravo:

```
int* device_array = 0;
int* host_array = 0;

cudaError err = cudaMemcpy(device_array, host_array,
      sizeof(int) * numOfElements, cudaMemcpyHostToDevice);
```

Prenos podatkov iz naprave:

```
cudaError err = cudaMemcpy(host_array, device_array,
      sizeof(int) * numOfElements, cudaMemcpyDeviceToHost);
```

Tudi pri grafičnih karticah moramo paziti na pravilno delo s pomnilnikom. Vsak rezervirani prostor na napravi moramo tudi sprostiti. Rezervirani prostor na grafični kartici sprostimo z ukazom:

```
cudaError err = cudaFree(device_array);
```

Ko imamo podatke na napravi, lahko ščepec pokličemo skoraj enako kot funkcijo. Dodati moramo še izvajalno konfiguracijo:

```
Dim3 grid(2, 4);
Dim3 block(10, 20);
Kernel<<<grid, block>>>(arg1, arg2);
```

Z zgornjo kodo smo pognali ščepec z imenom `Kernel`, ki kot argument dobi `arg1` in `arg2`. Vsi kazalci, ki jih pošljemo v argumentih, morajo kazati na naslovni prostor naprave. Ščepec bo izvajala mreža, ki ima 2 x 4 bloke. Vsak blok bo izvajal 10 x 20 niti.

2.12.2 Koda naprave

Koda znotraj naprave se s prevajalnikom CUDA prevede v vmesni jezik. Uporabljamo lahko le osnovni jezik C, brez knjižnic STL.

Znotraj ščepca lahko uporabimo avtomatično definirane spremenljivke, ki nam pomagajo določiti delo, ki ga mora posamezna nit narediti:

- `gridDim`: dimenzija mreže,
- `blockDim`: dimenzija bloka,
- `blockIdx`: ID bloka, v katerem se izvaja trenutna nit,
- `threadIdx`: ID niti znotraj bloka.

Spodnja koda prikazuje preprost ščepec, ki nam izračuna vsoto dveh vektorjev. Ščepec kot argument dobi vektor `a` in vektor `b`, ki ju mora sešteti in rezultat shraniti v vektor `c`. Število komponent, ki jih moramo sešteti, pa dobimo kot argument `n`. Vsaka nit v ščepcu izračuna vsoto le za en istoležni element. Informacijo, katerega mora sešteti, pa dobi na podlagi njene globalne številke ID, ki se izračuna na podlagi avtomatično definiranih spremenljivk. Ker arhitektura CUDA požene vse bloke z enako dimenzijo, se nam lahko podatki, ki jih mora nit obdelati, ne izidejo. Prav zato moramo obvezno preveriti, ali je izračunan globalni ID niti manjši od števila elementov. Če tega ne preverimo, lahko pri dostopu do tabel prekoračimo njeno velikost in pride do sesutje ščepca.

```
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
     // Get our global thread ID
     int id = blockIdx.x * blockDim.x + threadIdx.x;
     // Make sure we do not go out of bounds
     if (id < n)
          c[id] = a[id] + b[id];
}
```

Poglavje 3

Algoritmi računske geometrije

Področje računske geometrije se ukvarja z reševanjem praktičnih geometrijskih problemov. Med te probleme spadajo [23]:

- iskanje konveksne ovojnice,
- iskanje najbližjega para,
- iskanje Delaunayeve triangulacije,
- iskanje presečišča med telesi
- itd.

V tem delu se posvetimo algoritmoma iskanje konveksne ovojnice in iskanje najbližjega para.

3.1 Načini reševanja velikih problemov

V nadaljevanju opišemo dva pristopa, ki ju uporabimo pri programiranju. Pristopa deli in vladaj ter redukcija se pogosto uporabljata pri reševanju problemov.

3.1.1 Deli in vladaj

Zelo znan način reševanja velikih problemov je pristop deli in vladaj. Najbolj znan problem, rešen s tehniko deli in vladaj, je urejanje z algoritmom hitro iskanje (`QuickSort`).

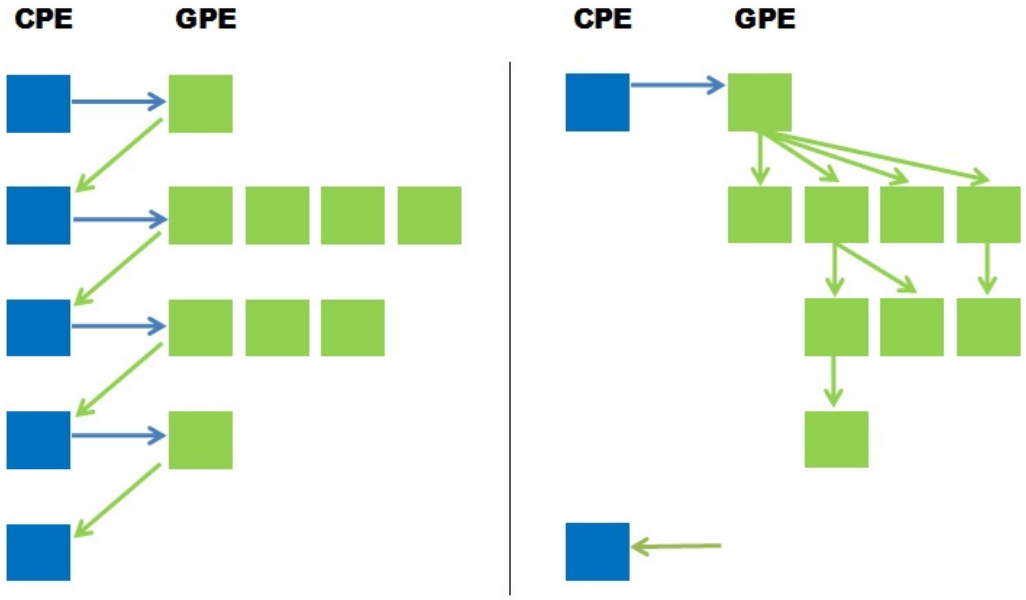
Problem rešimo rekurzivno, pri čemer na vsakem nivoju naredimo tri osnovne korake:

- deli – problem razdelimo na manjše instance istega problema;
- vladaj – za vsak podproblem poiščemo rešitev. Če je podproblem majhen in obvladljiv, ga rešimo, če je velik, pa najprej razdelimo;
- združi – rešitve podproblemov združimo nazaj v rešitev problema.

Prejšnja stanja rekurzije si zapomnimo avtomatsko s programskim skladom. Slabost rekurzije je veliko število klicev funkcij in omejena velikost programskega sklada (omejeno število skokov). Prav zato lahko namesto rekurzije uporabljamo sklad.

Tehnika deli in vladaj zaradi same rekurzije ni primerna za arhitekturo CUDA, zato se je treba take implementacije algoritma izogibati. V prihodnosti tega problema ne bo. Že danes najnovejše grafične kartice z arhitekturo CUDA verzije 3.5 omogočajo dinamični paralelizem.

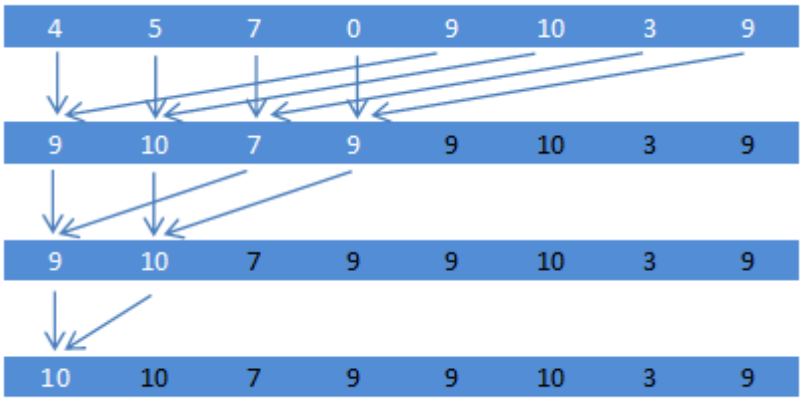
To pomeni, da lahko ščepec v izvajanje pošlje nov ščepec. Še vedno pa je pri tem veliko sinhronizacije, tako da v primeru, če obstaja algoritem za rešitev istega problema v isti časovni zahtevnosti, raje uporabimo drugega. Slika 5 prikazuje rešitev rekurzije z in brez dinamičnega paralelizma [7].



Slika 5: Rekurzija z in brez dinamičnega paralelizma

3.1.2 Redukcija

Na grafični kartici bi lahko združevanje rezultatov posameznih niti izvajala ena sama nit, medtem pa bi druge čakale, da konča. Enako bi lahko rezultate združili na gostitelju. Če je število niti `N`, to pomeni, da bi morali narediti `N - 1` združitev. Ta postopek lahko pohitrimo z redukcijo. Pri redukciji prva polovica niti združi svoje rezultate z istoležnim rezultatom v drugi polovici. Z vsako ponovitvijo se nam število združitev razpolovi. Iz tega sledi, da iz `N - 1` združitev potrebujemo $\log_2(N)$ korakov. Slika 6 prikazuje redukcijo, pri kateri rezultate združujemo tako, da vsakič vzamemo največji element.



Slika 6: Redukcija, pri kateri iščemo največje število

Koda redukcije [18]:

```
results[threadIdx.x] = someResult; //izračunaj rezultat
                                   //posamezne niti
__syncthreads(); //počakaj, da se izračuna rezultat za vse
                 //niti znotraj bloka

int i = blockDim.x / 2;
while(i != 0)
{
     if (threadIdx.x < i)
     {
          results[threadIdx.x] =
               mergeResults(results[threadIdx.x] +
               results[threadIdx.x + i]); //združitev rezultata
     }
     __syncthreads(); //počakaj, da vse ostale niti združijo
                      //rezultate
     i = i/2;
}
```

Ker moramo niti sinhronizirati z ukazom `__syncthreads()`, to pomeni, da redukcijo lahko delamo le znotraj bloka. Rezultate blokov moramo nato združiti s serijsko kodo ali s

ponovitvijo redukcije, kjer poženemo redukcijo rezultatov iz blokov. To ponavljamo, dokler ne poženemo le še enega bloka in je rezultat bloka enak končnemu rezultatu.

3.2 Iskanje najbližjega para

Dano imamo množico točk v dvodimenzionalnem prostoru. Poiskati moramo par, ki je najbliže skupaj (slika 7). Za računanje razdalj med točkami uporabimo evklidsko razdaljo.

Imamo seznam točk `P`, ki ima `n` elementov:

$$|P| = n$$

$$P = \{ p_1, p_2, p_3 \dots p_n \}$$

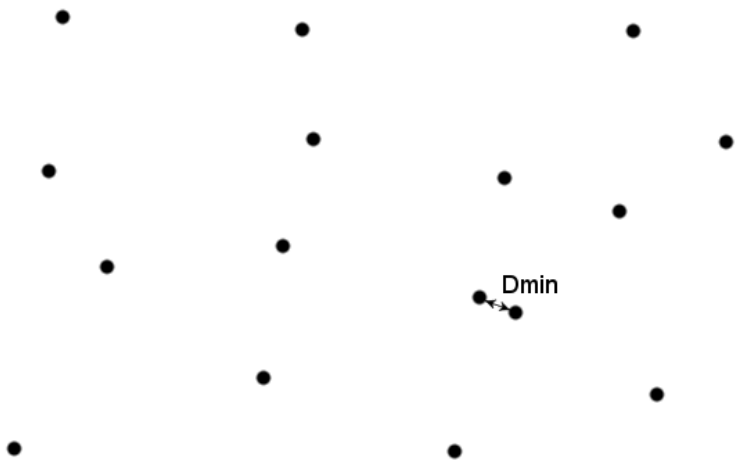
Vsaka dvodimenzionalna točka v seznamu ima koordinati `x` in `y`:

$$p_i = (x_i, y_i)$$

Med dvema točkama izračunamo evklidsko razdaljo z enačbo:

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Algoritem poišče taka `i` in `j`, da je vrednost `d(p`$_i$`, p`$_j$`)` najmanjša.



Slika 7: Najbližji par

Praktični primer uporabe algoritma je nadzor zračnega ali vodnega prometa. Algoritem nam v tem primeru vrne najbližji ladji oz. letali, med katerima potencialno lahko pride do trka.

3.2.1 Pristopi (algoritmi) pri reševanju problema najbližjih parov

Najpreprostejši pristop pri reševanju tega problema je algoritem, kjer preverimo vse pare. Psevdokoda algoritma iskanje najbližjega para:

```
Vhod:
P - seznam točk
Izhod:
Najmanjša razdalja

najblizjiParTock (P)       //P je seznam točk

d = razdalja(p1, p2)

for i od 1 do n - 1
      for j od i + 1 do n
            če d(pi, pj) < d
                  posodobi d       //si jo zapomnimo

 Vrni d
```

Vsako točko primerjamo z vsemi njenimi naslednjimi točkami v seznamu. Če z n označimo število točk, to pomeni, da moramo narediti:

$$(n-1)+(n-2)+(n-3)+\cdots+2+1=\frac{n(n-1)}{2}$$

izračunov in primerjav. Tak algoritem ima zato časovno zahtevnost $O(n^2)$.

3.2.2 Strategija deli in vladaj

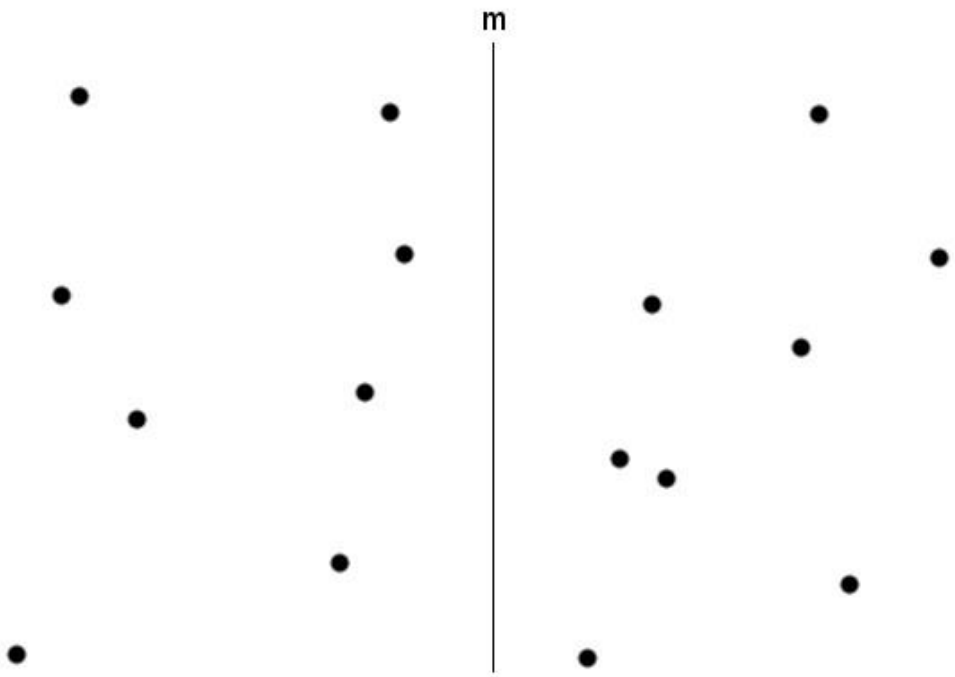
Problem lahko hitreje rešimo s strategijo deli in vladaj, pri kateri je časovna zahtevnost $O(n \log n)$ [1].

Faze strategije deli in vladaj:

- deli: množico točk razdelimo na dva dela, pri čemer je v vsakem delu polovica točk. Trenutno predpostavimo, da imamo sodo število točk;
- vladaj: za vsak podproblem poiščemo rešitev. Kaj naredimo, je odvisno od števila točk `n`, ki jih imamo v podproblemu:
  - `n > 3`: problem ponovno razdelimo na podprobleme;
  - `n = 3`: izračunamo razdalje med preostalimi tremi točkami in vrnemo najkrajšo razdaljo;
  - `n = 2`: vrnemo razdaljo med preostalima točkama;
- združevanje: v tej fazi moramo rešitvi podproblema združiti. Pri tem je lahko najbližji par iz levega podproblema, desnega podproblema ali iz para, sestavljenega iz obeh množic.

3.2.2.1 Faza delitve

V fazi delitve razdelimo množico točk na dve številčno enaki množici, ki ju ločuje navpična črta `m` (slika 8). Pri tem mora veljati, da imajo vse točke v levi množici manjšo koordinato `x` in točke v desni množici večjo koordinato `x`.



Slika 8: Delitev problema na dva podproblema

Najlažje to naredimo tako, da točke uredimo po koordinati `x`. Tako v levo množico padejo točke, ki se nahajajo v prvi polovici seznama, v desno množico pa padejo točke, ki se nahajajo v drugi polovici seznama. Urejanje točk po koordinati `x` lahko izvedemo pred rekurzijo, ker naprej v posameznih fazah rekurzije to ni več potrebno, saj točke v fazi delitve ostanejo urejene.

Koordinato `X` navpične črte `m` izračunamo tako, da vzamemo povprečje koordinate `x` tiste točke, ki je najbolj desno v levi množici, in točke, ki je najbolj levo v desni množici. Zaradi urejenosti točk po osi `x` to pomeni, da vzamemo zadnjo točko v levem seznamu in prvo točko v desnem seznamu.

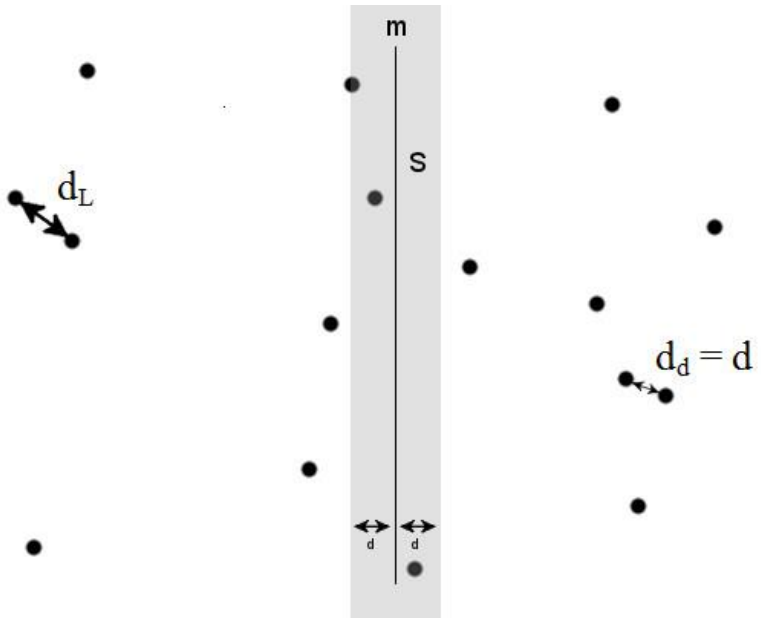
3.2.2.2 Faza združevanja

V fazi združevanja moramo upoštevati, da je lahko najbližji par v levi množici ali pa v desni množici, lahko pa je to tudi par, ki je sestavljen iz leve in desne množice.

Najprej iz leve in desne množice vzamemo par, ki ima manjšo medsebojno razdaljo. Tako dobimo najmanjšo razdaljo `d` iz levega in desnega dela.

Sedaj moramo pregledati še pare točk, ki so čez mejo. V smeri `x` lahko iskanje omejimo na točke, ki se nahajajo v sredinskem pasu (slika 9). Ta pas je območje, kjer so točke po osi `x` oddaljene od navpične črte `m` za manj ali enako kot `d`. Točka `p` je v sredinskem pasu, če velja:

$$p_x \geq m - d \; in \; p_x \leq m + d$$

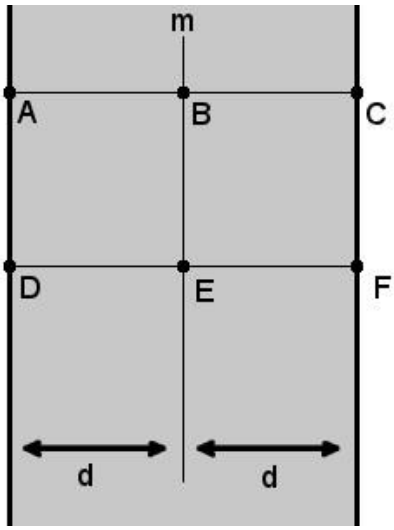


Slika 9: Sredinski pas

V najslabšem primeru lahko pride do tega, da vse točke ležijo v sredinskem pasu, kar pomeni, da imamo ponovno kvadratno časovno zahtevnost.

Prav zato moramo omejiti iskanje tudi v smer `y`, da nam ni treba preveriti vseh parov v sredinskem pasu. Vemo, da morajo biti točke v sredinskem pasu na vsaki strani oddaljene vsaj za razdaljo `d`. Točke v sredinskem pasu uredimo po koordinati `y`. Da nam tega ni treba vedno urejati (časovna zahtevnost `O(n log n)`), se lahko temu izognemo tako, da vsak rekurzivni klic vrne že po koordinati `y` urejen seznam. Vse, kar nam preostane, je, da z zlivanjem združimo urejen seznam točk iz leve in desne množice. Zlivanje ima časovno zahtevnosti `O(n)`.

Za vsako točko v sredinskem pasu lahko omejimo območje iskanja na pravokotnik, ki je širok `2d` in visok `d` (slika 10). Vse točke izven tega pravokotnika ne morejo biti bliže od trenutne razdalje `d`. Izkaže se, da lahko v tak pravokotnik spravimo le osem točk in moramo zato pri vsaki točki pogledati le nadaljnjih sedem točk. Ker je največje število primerjav pri vsaki točki konstantno, to pomeni, da je za pregled vseh točk v sredinskem pasu potrebna linearna časovna zahtevnost [11].



Slika 10: Točke v sredinskem pasu

3.2.3 Časovna analiza algoritma

Pred izvedbo rekurzije moramo izvesti urejanje točk po koordinati `x`. Dober algoritem za urejanje ima časovno zahtevnost `O(n log n)`.

Vsaka faza rekurzije naredi:

- razdelitev na dve podmnožici `O(1)`,
- rekurzivni klic za levo in desno podmnožico,
- združitev urejenih seznamov točk po osi `y` `O(n)`,
- določitev točk v sredinskem pasu `O(n)`,
- preveritev točk v sredinskem pasu `O(n)`.

Vsaka faza rekurzije ima časovno zahtevnost:

$$O(n)$$

Za izračun časovne zahtevnosti algoritma moramo zapisati rekurzivno enačbo v obliki:

`a` – število podproblemov
`n` – število elementov
`b` – za koliko zmanjšamo problem na vsakem nivoju rekurzije

n^k – časovna zahtevnost delitve problema in združevanja podproblemov

$$T(n) = a * T\left(\frac{n}{b}\right) + O(n^k)$$

Po Master teoremu lahko tako izračunamo časovno zahtevnost po enačbi [24]:

$$T(n) = \begin{cases} O\left(n^{\log_b a}\right), \text{če je } a > b^k \\ O(n^k \log n), \text{če je } a = b^k \\ O(n^k), \text{če je } a < b^k \end{cases}$$

V primeru algoritma najbližji par je rekurzivna enačba naslednja:

$$T(n) = 2 * T\left(\frac{n}{2}\right) + O(n^1)$$

Po Master teoremu to pomeni, da je časovna zahtevnost našega algoritma enaka:

$$O(n \log n)$$

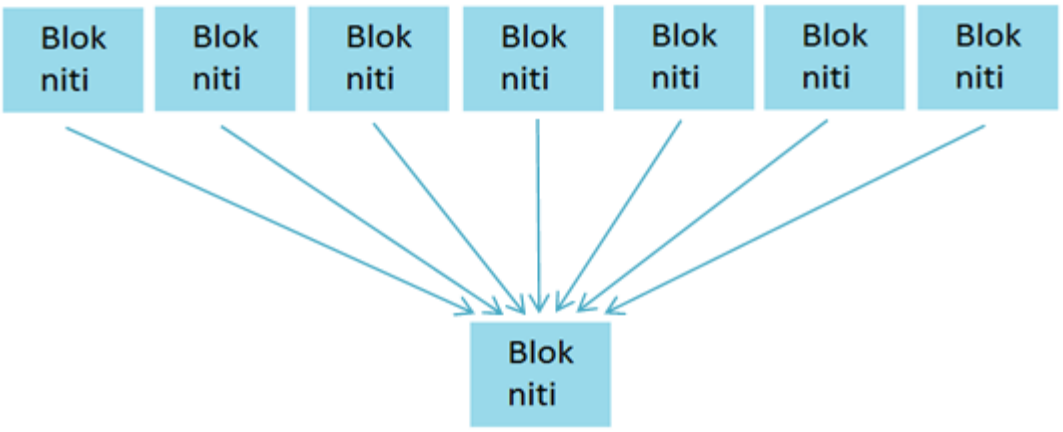
3.2.4 Implementacija algoritma na grafični kartici

Ker pristop deli in vladaj zaradi rekurzije ni pisan na kožo grafičnim karticam, smo problem rešili tako, da smo delitev naredili brez rekurzije. V ta namen smo urejene točke po koordinati `x` ročno razdelili v skupine po dve točki skupaj. V zadnji skupini so lahko tri točke, a trenutno predpostavljajmo, kot da sta povsod samo dve točki.

Nato smo pognali inicializacijski ščepec, ki naredi:

- izračuna razdaljo med točkama;
- izračuna najmanjšo in največjo koordinato `x`. Ta podatek potrebujemo v nadaljevanju za izračun pokončne črte `m`;
- uredi točki po koordinati `y`.

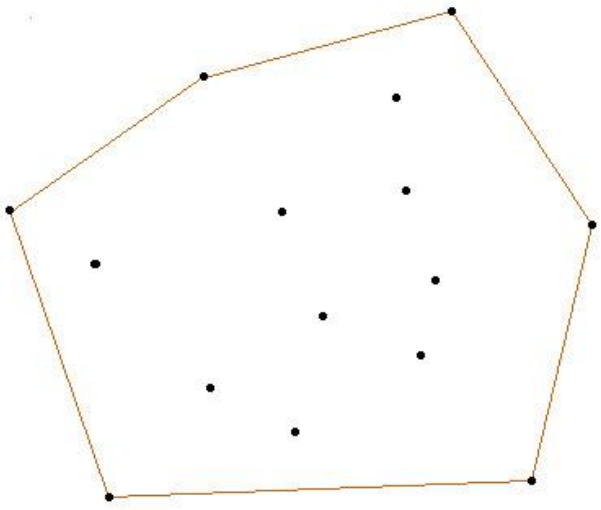
V naslednjem koraku smo pognali ščepec, ki množice združuje (faza združevanja). Združevanje delamo z redukcijo, pri čemer je izredno pomembno, da redukcija združuje sosednji množici. Redukcijo zaradi sinhronizacije lahko delamo le znotraj bloka. Prav zato moramo rezultate redukcij iz blokov združevati in ponovno pognati ščepec. To delamo toliko časa, dokler ne poženemo ščepca z enim blokom, rezultat bloka pa predstavlja rezultat algoritma. Združevanje blokov prikazuje slika 11.



Slika 11: Redukcija blokov niti

3.3 Iskanje konveksne ovojnice

Konveksna ovojnica je najmanjši konveksni poligon, ki vsebuje vse točke v množici. Točke ležijo znotraj konveksne ovojnice ali pa so del nje. Vsaka točka v množici, okoli katere se konveksna ovojnica obrne, predstavlja točko konveksne ovojnice (slika 12).



Slika 12: Konveksna ovojnica

Algoritem se lahko uporablja v računalniški grafiki, kjer kompleksnim telesom izračunamo

konveksne ovojnice. Na podlagi konveksnih ovojnic predmetov v prostoru je preverjanje trkov veliko učinkovitejše. Ta pristop uporabljajo tudi fizikalni in igralni pogoni. Eden takih igralnih pogonov je Bullet Physics [14].

Bullet Physics je odprtokodna knjižnica, ki jo lahko uporabimo na platformah:

- Windows,
- Linux,
- Mac OS X,
- Xbox 360,
- PlayStation 3,
- Android,
- iPhone,
- Nintendo Wii.

Bullet Physics je profesionalna 3D-knjižnica, ki nam preverja trke med objekti in njihovo dinamiko.

3.3.1 Pristopi (algoritmi) pri reševanju problema konveksne ovojnice

Obstaja več algoritmov reševanja problema konveksne ovojnice:

- Jarvis march,
- Graham scan,
- QuickHull,
- Chan's algoritm
- itd.

V osnovi se algoritmi razlikujejo po tem, ali je njihova časovna zahtevnost odvisna le od števila točk v množici ali tudi od števila točk konveksne ovojnice [2].

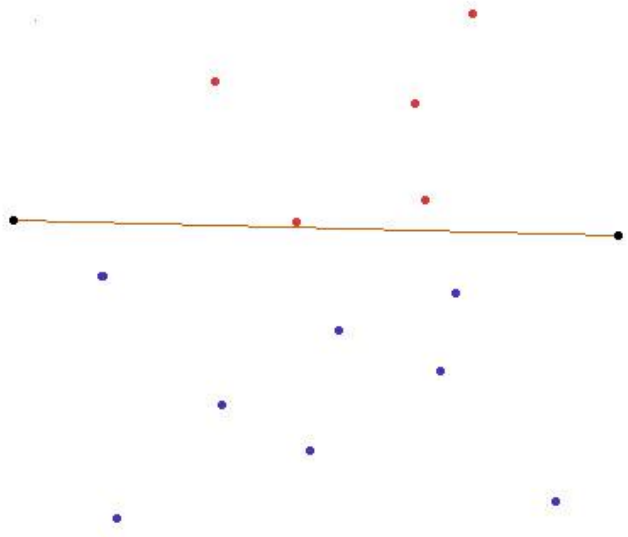
Na začetku smo se lotili implementacije algoritma na centralno procesno enoto z metodo Graham scan. Nato smo poskusili problem rešiti s hitro konveksno ovojnico (QuickHull). Izkazalo se je, da je hitra konveksna ovojnica veliko hitrejša in lažja za implementacijo na arhitekturi CUDA.

3.3.2 Hitra konveksna ovojnica

Hitra konveksna ovojnica uporablja pristop deli in vladaj. Sam algoritem je zelo podoben

hitremu urejanju (Quick sort) in iz tega izhaja tudi ime algoritma.

Na začetku najdemo točki z najmanjšo in največjo koordinato `x` (slika 13). S tem smo dobili prvi dve točki konveksne ovojnice. Konveksna ovojnica med njima (daljica) ločuje prostor na dva dela.



Slika 13: Delitev na podproblem

V fazi delitve za vsako na novo nastalo daljico rekurzivno poiščemo najbolj oddaljeno točko (slika 14). To točko poiščemo z enačbo:

`p1` – prva točka daljice

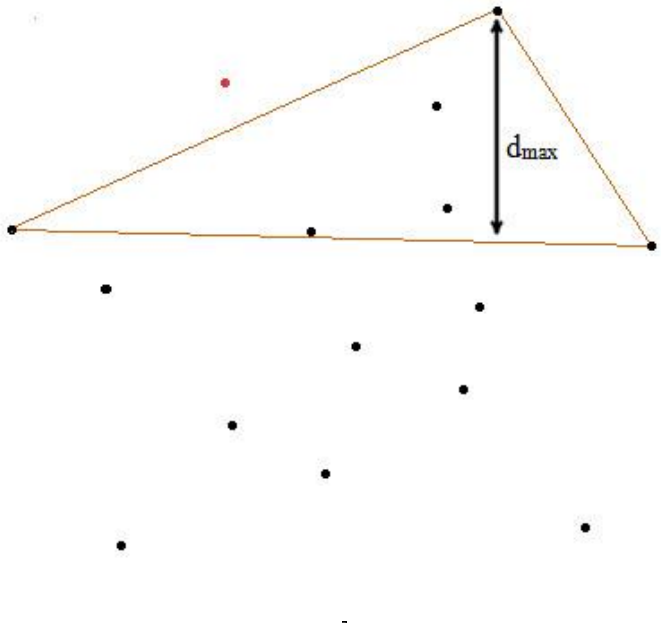
`p2` – druga točka daljice

`p` – točka, za katero gledamo oddaljenost od daljice

$$d(p1, p2, p) = (p1_y - p2_y)(p_x - p1_x) + (p2_x - p1_x)(p_y - p1_y)$$

Najbolj oddaljena točka tako zagotovo postane del konveksne ovojnice. Točke znotraj nje lahko v nadaljevanju zanemarimo, saj ni več možno, da bi bile njeni člani.

Rekurzijo oziroma fazo delitve ponavljamo toliko časa, dokler imamo nad daljico še kakšno točko.



Slika 14: Iskanje najbolj oddaljene točke

Psevdokoda algoritma hitre konveksne ovojnice:

```
Vhod:
points – seznam točk

Izhod:
Vrne točke konveksne ovojnice

getConvexHull(points):
     pMin = findMinXPoints(points)
     pMax = findMaxXPoints(points)
     r1 = buildConvexHull(Line(pMin, pMax),points)
     r2 = buildConvexHull(Line(pMax, pMin),points)
     vrni zdruzi(r1,r2)

buildConvexHull(line, points):
     p = findMostDistantPointFromLine(line, points)
     //poiščemo točko, ki je najbolj oddaljena od daljice
     //poleg iz seznama points izločimo vse točke, ki so pod
     //črto
     če points ima točke:
```

```
        r1 = buildConvexHull(Line(line.p1, p),points)
        r2 = buildConvexHull(Line(p, line.p2),points)
        vrni zdruzi(r1,r2)
    če points nima točk:
        vrni line.p1
```

3.3.3 Časovna analiza hitre konveksne ovojnice

Pred izvedbo rekurzije moramo poiskati točki z najmanjšo in največjo vrednostjo koordinate `x`. Časovna zahtevnost je `O(n)`.

V vsaki fazi rekurzije poiščemo najbolj oddaljeno točko od daljice. Časovna zahtevnost je `O(n)`.

V primeru da nam vsaka nova točka konveksne ovojnice razdeli preostale točke na dva enaka dela, je rekurzivna enačba:

$$T(n) = 2 * T\left(\frac{n}{2}\right) + O(n^1)$$

Pa Master teoremu iz tega sledi, da je časovna zahtevnost algoritma enaka:

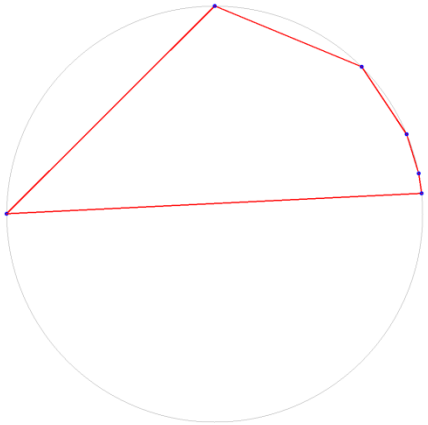
$$O(n \log n)$$

3.3.3.1 Časovna analiza algoritma v najslabšem primeru

Prav tako kot hitro urejanje ima tudi hitra konveksna ovojnica v najslabšem primeru časovno zahtevnost $O(n^2)$. Oba algoritma imata problem v primeru, ko posamezna rekurzija ne razdeli problema na dva čim bolj enaka dela. Pri hitri konveksni ovojnici pride do tega v primeru (slika 15), ko so točke razporejene na krožnici po kotu [17]:

$$\alpha = \frac{\pi}{2^i}$$

V tem primeru na eni polovici prostora ostanejo vse točke, na drugi pa nobena, zato je tako kot pri hitrem urejanju časovna zahtevnost enaka $O(n^2)$.



Slika 15: Najslabši primer hitre konveksne ovojnice

Enak problem nastane pri hitrem urejanju ob izbiri napačnega pivota. Pri hitrem urejanju se slabemu scenariju lahko izognemo s pametnejšo izbiro pivota. Pri hitri konveksni ovojnici pa se tega problema ne moremo rešiti, saj mora biti pivot najbolj oddaljena točka.

3.3.4 Implementacija algoritma na grafične kartice

Ker naša grafična kartica ne podpira dinamičnega paralelizma, je tudi pri tem algoritmu nastal problem, kako se izogniti rekurziji. Rekurziji smo se izognili tako, da je zanjo skrbel procesor, ki je v izvajanje grafični kartici pošiljal posamezne daljice. Na podlagi rezultata pa je procesor razdelil daljico na dve daljici, ki ju je v izvajanje poslal naslednji krog rekurzije.

Posamezni ščepec je za vsako preostalo točko izračunal njeno razdaljo do daljice. Razdalje med daljicami smo nato z redukcijo združili. Redukcija je združevala po večji oddaljenosti od daljice.

Poskusili smo tudi drugi način reševanja rekurzivnega problema, kjer en ščepec izračuna rezultate za celotni nivo rekurzivnega drevesa. Ta rešitev se zaradi težjega računanja indeksov ni obnesla in so bili rezultati slabši.

Poglavje 4

Testiranje

Vsak algoritem smo spisali za centralno procesno enoto in arhitekturo CUDA ter primerjali rezultate.

Za vsako testiranje smo na začetku pripravili testno množico. Nad testno množico smo nato pognali algoritem, implementiran na centralni procesni enoti ter arhitekturi CUDA. Velikost testne množice je odvisna od testa, saj smo hoteli videti in primerjati rezultate za različne velikosti množice.

Na vsaki velikosti testne množice smo izvedli petdeset ponovitev. Na podlagi izmerjenih časov smo nato izračunali povprečje, ki se izračuna po enačbi [4]:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{1}{n}(x_1 + \cdots + x_n)$$

Ker so rezultati meritev približno normalno razporejeni, smo izračunali tudi standardni odklon. Ta nam pove razpršenost meritev. Izračuna se po enačbi [20]:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}}$$

4.1 Opis izvajalnega okolja

Vsako testiranje je potekalo na istem računalnik. Računalnik vsebuje procesor Intel i7-3770K. Procesor ima štiri jedra. Vsako jedro lahko hkrati izvaja dve niti. Normalna frekvenca procesorja je 3,5 GHz, ki se lahko ob obremenitvi za nekaj časa poveča do 3,9 GHz. Do pomnilnika ima dve pomnilniški vodili za pomnilnike tipa DD3-1333/1600. Procesor ima vgrajeno grafično kartico Intel HD 4000. Kljub že vgrajeni grafični kartici pa imamo prek vodila PCI Express 3.0 možnost vgraditi tudi zunanjo [16].

Za grafično kartico smo uporabili Gigabyte GeForce GTX 660. Grafična kartica vsebuje dva pomnilnika GB GDDR5. Za povezovanje s procesorjem uporablja vodilo PCI Express 3.0. Grafična kartica je že tovarniško navita. Tako je osnovna hitrost grafične kartice iz 980 MHz dvignjena na 1033 MHz. Grafična kartica podpira arhitekturo CUDA z računsko

kompatibilnostjo 3,0. Vsebuje 960 jeder CUDA [9].

4.2 Gradnja projekta

Za gradnjo projekta smo uporabili CMake. Uporabljali smo generator za Visual Studio 2010, saj v času izdelave diplomske naloge CMake še ni podpiral arhitekture CUDA za Visual Studio 2012.

4.2.1 CMake

CMake je odprtokodni, večplatformni sistem za gradnjo projekta. Podpira generatorje projektov za:

- Visual Studio,
- Unix Makefiles,
- Eclipse,
- CodeBlocks,
- Borland Makefiles
- itd.

CMake združi izvorno kodo v projekt prek konfiguracijskih datotek z imenom CMakeLists.txt. Projekt nastavimo v grafičnem uporabniškem vmesniku, kjer nastavljamo parametre projekta. Kaj lahko nastavimo, je odvisno predvsem od konfiguracijskih datotek. CMake privzeto dobimo že z veliko skriptami, ki nam pomagajo nastaviti zunanje knjižnice. Omogočena je preprosta razširljivost, in sicer z dodajanjem lastnih skript.

Pisanje konfiguracijskih datotek poteka enako kot pisanje programa v preprostem jeziku. Ta preprosti programski jezik vsebuje ukaze za delo s spremenljivkami:

```
SET(VARIABLE ${a})
```

Spremenljivki `VARIABLE` nastavimo vrednost spremenljivke `a`.

```
SET(VARIABLE a b c)
```

Spremenljivki `VARIABLE` nastavimo seznam z vrednostnimi `a`, `b` in `c`.

Imamo tudi ukaze za kontrolo toka programa:

– pogojni stavek:

```
IF(POGOJ)
     Ukazi…
ENDIF(POGOJ)
```

– zanke:

```
FOREACH(f ${VAR})
     message(${f})
ENDFOREACH(f)
```

4.2.1.1 Osnovne funkcije v programu CMake

Nekatere izmed osnovnih funkcij programa CMake:

– `CMAKE_MINIMUM_REQUIRED(VERSION 2.8)`: navedemo najmanjšo zahtevano verzijo programa CMake, ki ga moramo imeti naloženega;

– `PROJECT(IME_PROJEKTA)`: navedemo ime projekta, ki je lahko poljubno, saj ime projekta ne spreminja funkcionalnosti:

– `FIND_PACKAGE(LIB_NAME REQUIRED)`: s tem ukazom v projekt dodamo zunanjo (3rdParty) knjižnico. Ukaz poišče skripto z imenom `FindLIB_NAME.cmake`. Z drugim parametrom `REQUIRED` pa povemo, da je knjižnica obvezna. V primeru, da je knjižnica obvezna in skripta ne najde te knjižnice, nam CMake vrne napako;

– `ADD_SUBDIRECTORY(DIRNAME)`: v projekt dodamo direktorij, ki prav tako vsebuje konfiguracijsko datoteko;

– `INCLUDE_DIRECTORIES(DIR_LIST)`: knjižnici oziroma izvršilnemu programu dodamo poti, po katerih išče vključena zaglavja v projektu;

– `ADD_EXECUTABLE(EXECUTABLE_NAME FILE_LIST)`: naredimo izvršilno datoteko z imenom EXECUTABLE_NAME, ki jo zgradimo z datotekami v seznamu `FILE_LIST`;

– `ADD_LIBRARY(LIBRARY_NAME FILE_LIST)`: naredimo knjižnico z imenom LIBRARY _NAME, ki jo zgradimo z datotekami v seznamu `FILE_LIST`;

– `TARGET_LINK_LIBRARIES(NAME LIBRARIES_LIST)`: s tem ukazom povežemo

našo knjižnico oz. izvršilni program z drugimi knjižnicami.

4.2.1.2 Sestava programa CMake

CMake je sestavljen iz štirih delov:

- CMake – osnovni modul, ki skrbi za gradnjo projekta,
- CTest – skrbi za pisanje in izvajanje testov v programu,
- CDash – spletni strežnik, na katerem vidimo rezultate testiranja,
- CPack – iz programa naredimo namestitveno datoteko za poljuben sistem.

V našem projektu smo uporabili le osnovni CMake. CTest in CDash sta pomembna v primeru, ko na projektu dela več razvijalcev. Ker je rezultat našega dela knjižnica in ne samostojni program, v ta namen ne potrebujemo paketa CPack.

4.2.2 Hierarhija projekta

```
PROJECT
|   CMakeLists.txt
|
+---Base
|       Base.cpp
|       Base.h
|       CMakeLists.txt
|
+---ClosestPairCuda
|       ClosestPair.cu
|       ClosestPair.h
|       CMakeLists.txt
|
+---ClosestPairTest
|       ClosestPairTest.cpp
|       CMakeLists.txt
|
+---ConvexHullCuda
|       CMakeLists.txt
|       ConvexHull.cu
|       ConvexHull.h
|
+---ConvexHullTest
        CMakeLists.txt
        ConvexHullTest.cpp
```

Slika 16: Hierarhija projekta

V vsakem nivoju datotečne strukture se nahaja `CMakeLists.txt` (slika 16), v katerem se nahajajo ukazi za gradnjo projekta.

V direktoriju `Base` se nahajajo strukture, ki jih uporabljajo vse naše knjižnice. Tako imamo

notri implementirano strukturo `Point` in računanje razdalje med točkami.

V direktoriju `ClosestPairCuda` imamo implementiran algoritem najbližji pari na arhitekturi CUDA. Algoritem pokličemo s funkcijo:

```
Result shortestPathCalculation(Point* points, int size);
```

Funkcija za argument dobi kazalec na seznam točk in število točk. Kot rezultat vrne strukturo `Result` s točkama, ki sta si najbliže.

V direktoriju `ConvexHullCuda` imamo napisan algoritem za iskanje konveksne ovojnice. Algoritem pokličemo s funkcijo:

```
Result convexHull(Point* points, int size);
```

Tudi ta funkcija za argumente dobi kazalec na seznam točk in število točk. Kot rezultat nam vrne seznam točk konveksne ovojnice.

Teste imamo spisane v datotekah `CovexHullTest` in `ClosestPairTest`, kjer primerjamo rezultate algoritmov, izvedenih na procesorju in grafični kartici.

Povsod, kjer uporabljamo arhitekturo CUDA, smo kodo razdelili na zaglavno datoteko (`.h`) in na datoteko z implementacijo (`.cu`). V zaglavni datoteki se nahaja deklaracija funkcij in struktur. V datoteki z implementacijo pa implementacija teh funkcij.

Z ločitvijo kode na deklaracijo in implementacijo smo dosegli, da uporabniku te knjižnice v svojem programu ni treba vključevati arhitekture CUDA in je s stališča njegovega programa ne vidi.

4.2.3 Konfiguracijska datoteka

Kot je že bilo omenjeno, se v datotekah CMakeLists.txt nahajajo ukazi za gradnjo projekta. Naša korenska konfiguracija vsebuje:

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)
option(BUILD_TESTS "Build tests" ON)
PROJECT(DIPLOMSKA_NALOGA)
FIND_PACKAGE(CUDA REQUIRED)
SET(CUDA_NVCC_FLAGS "-arch=sm_20")
add_subdirectory(Base)
add_subdirectory(ClosestPairCuda)
```

```
add_subdirectory(ConvexHullCuda)

IF(BUILD_TESTS)

add_subdirectory(ClosestPairTest)
add_subdirectory(ConvexHullTest)

ENDIF(BUILD_TESTS)
```

V tej konfiguracijski datoteki smo navedli ime projekta, ki je `DIPLOMSKA_NALOGA`. Najmanjša verzija, s katero lahko gradimo projekt, je kar verzija, ki jo uporabljamo. Nato smo vključili knjižnico CUDA in dodali poddirektorije projekta. V primeru, da uporabnik izbere opcijo `BUILD_TESTS`, v projekt dodamo tudi testa.

Konfiguracijska datoteka testa za najbližji par je:

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

include_directories (${CMAKE_SOURCE_DIR})

ADD_EXECUTABLE(ClosestPairTest ClosestPairTest.cpp)

target_link_libraries(ClosestPairTest Base ClosestPairCuda)
```

Ta konfiguracija iz datoteke `ClosestPairTest.cpp`, ki vsebuje `main` funkcijo, naredi izvršilno datoteko. Izvršilno datoteko smo povezali s knjižnicama `Base` in `ClosestPairCuda`.

4.3 Testiranje algoritma najbližji par

Najprej smo testirali algoritem najbližji par. Pravilnost rešitve na centralni procesni enoti in grafični kartici smo preverjali s preprostim algoritmom, ki gre čez vse pare točk in poišče par z najmanjšo medsebojno razdaljo.

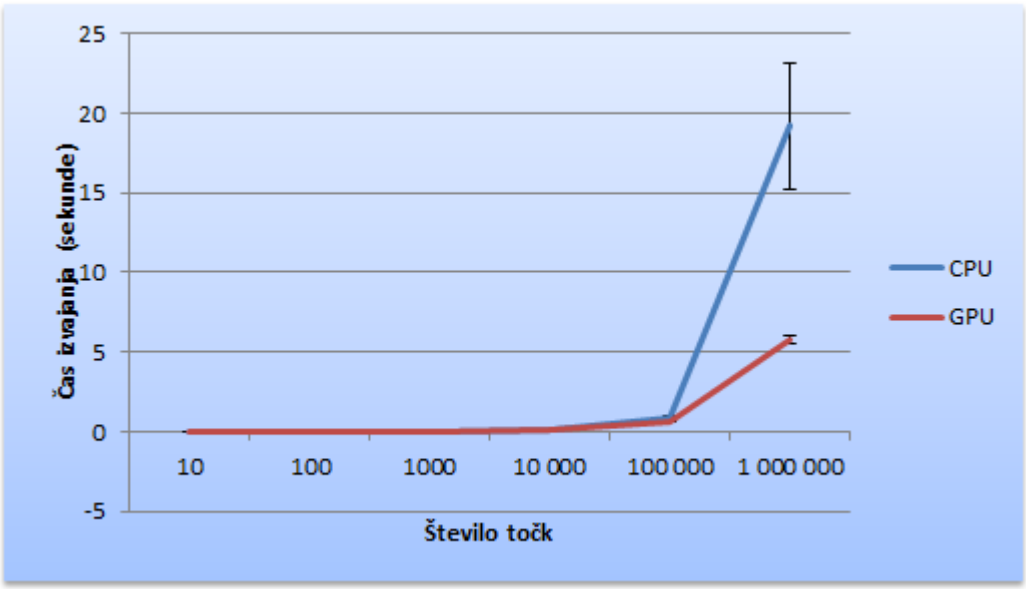
4.3.1 Testiranje na naključnih točkah

Za prvo testiranje smo testne točke pripravili tako, da smo obe koordinati v prostoru generirali naključno.

V spodnji tabeli so prikazani povprečni časi in standardni odklon glede na število točk.

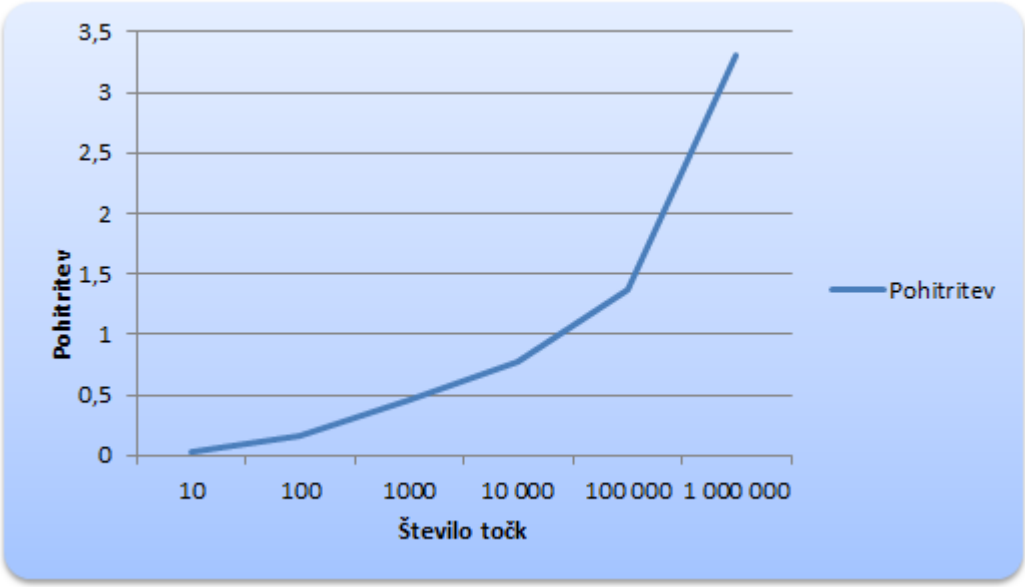
Naprava	Centralna procesna enota		Grafična kartica	
n	povprečje [s]	standardni odklon [s]	povprečje [s]	standardni odklon [s]
10	0,00003	0,00017	0,0011	0,0051
100	0,00036	0,0005	0,0023	0,00064
1 000	0,0054	0,0016	0,0118	0,00076
10 000	0,068	0,034	0,088	0,0033
100 000	0,84	0,19	0,62	0,017
1 000 000	19,2	4	5,8	0,25

Na podlagi rezultatov smo nato izrisali graf (slika 17), ki nam pokaže čas izvajanja algoritma na centralni procesni enoti in grafični kartici za različno število točk. Prikazani so povprečni časi in standardni odklon.



Slika 17: Graf izvajanja v sekundah za algoritem najbližji par na naključnih točkah

Pripravili pa smo tudi graf, ki prikazuje pohitritev algoritma, pognanega na grafični kartici, v primerjavi s centralno procesno enoto (slika 18). Na ordinatni osi je prikazana pohitritev. Pohitritev, manjša od 1, nam pove, da se je bolje odrezala centralna procesna enota, pohitritev, večja od 1, pa nam pove, da se je bolje odrezala grafična kartica.



Slika 18: Graf pohitritve za algoritem najbližji par

Iz grafa je dobro razvidno, da se je grafična kartica veliko bolje odrezala pri veliki testni množici. Pri majhnih testnih množicah se namreč ne obnese dobro, saj moramo pred izvajanjem poslati podatke na napravo in po končanem izvajanju ščepca tudi iz nje.

Presenetil nas je standardni odklon, ki je pri grafični kartici manjši. Majhen standardni odklon nam pove, da se čas izvajanja algoritma malo spreminja. To pride v poštev pri realnočasovnih problemih, kjer je važno, da poznamo čim natančnejši in konstanten čas izvajanja algoritma.

Glede na rezultate testiranja se splača za manjše množice točk vzeti implementacijo za procesor, za velike množice točk pa implementacijo za arhitekturo CUDA. Za določitev meje za preklop bi potrebovali več testiranj na različnih sistemih.

4.3.2 Testiranje na normalni porazdelitvi točk

Testirali smo tudi testne podatke, kjer so bile točke razporejene normalno. To pomeni, da smo za vrednost koordinat `x` in `y` točke generirali z normalno razporeditvijo.

Za generiranje testnih števil po normalni porazdelitvi smo uporabili kar standardno knjižnico:

```
std::default_random_engine generator;
std::normal_distribution<double> normal(0,50);
//povprečje je 0, standardni odklon je 50
double value = normal(generator); //vrednost
```

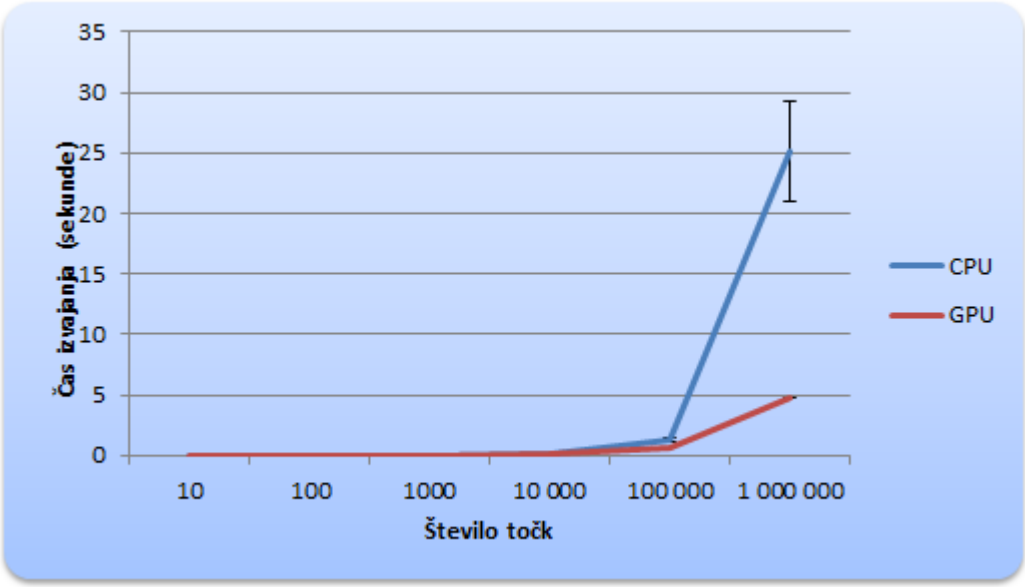
Za velike testne množice to pomeni:

- da je povprečna vrednost koordinat enaka 0,
- da se na intervalu [–50, 50] nahaja 68,2 % vrednosti koordinat.

Tabela, ki prikazuje povprečje in standardni odklon na CPE in grafični kartici za različna števila točk:

Naprava	Centralna procesna enota		Grafična kartica	
n	povprečje [s]	standardni odklon [s]	povprečje [s]	standardni odklon [s]
10	0,00004	0,0002	0,00094	0,00061
100	0,00047	0,0005	0,002	0,0005
1 000	0,0062	0,0012	0,012	0,0008
10 000	0,08	0,056	0,086	0,004
100 000	1,28	0,14	0,6	0,0064
1 000 000	25,14	4,15	4,8	0,045

Graf, ki prikazuje povprečni čas in standardni odklon za različno število točk na normalni porazdelitvi:



Slika 19: Graf povprečnega časa in standardnega odklona za normalno porazdeljene točke

Rezultati pri normalni porazdelitvi so zanimivi s tega vidika, da se je procesor obnesel slabše kot pri naključnih točkah, medtem ko se je na grafični kartici obnesel bolje.

Sklepamo, da se je na procesorju obnesel slabše, ker se večina točk (68,2 %) nahaja znotraj ozkega intervala [–50, 50]. Ker so tam točke približno enako skupaj, to pomeni približno enako najmanjšo razdaljo levega in desnega dela pri fazi združevanja. Ker sta obe najmanjši razdalji približno enaki, to pomeni povprečno več števil v srednjem pasu.

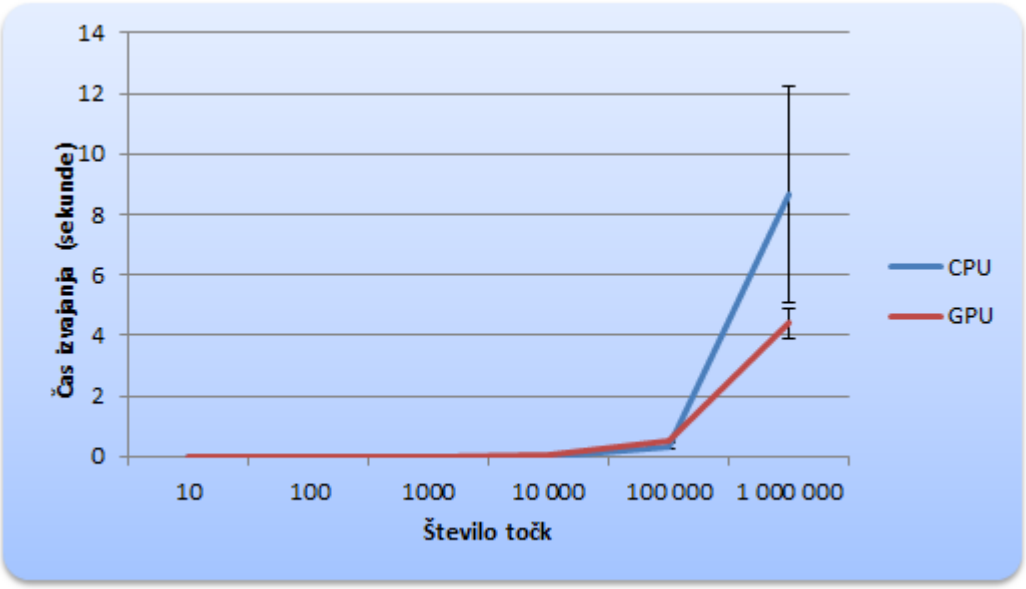
4.3.3 Testiranje na točkah, porazdeljenih navpično

Pri tem testu smo hoteli preveriti algoritem pri točkah, pri katerih so koordinate `y` enake. S tem smo dosegli, da v fazi delitve ne naredimo ničesar pametnega in vso nalogo iskanja najbližjega para prevzame faza združevanja. Ker je razlika koordinate `y` pri vseh točkah enaka 0, to pomeni, da vse točke padejo v sredinski pas.

Rezultati testiranja na točkah, porazdeljenih navpično, so prikazani v tabeli:

Naprava	Centralna procesna enota		Grafična kartica	
n	povprečje [s]	standardni odklon [s]	povprečje [s]	standardni odklon [s]
10	0	0	0,00096	0,0004
100	0,00008	0,00027	0,0017	0,0004
1 000	0,0017	0,0006	0,009	0,00086
10 000	0,02	0,02	0,07	0,004
100 000	0,35	0,11	0,51	0,009
1 000 000	8,65	3,58	4,4	0,5

Graf, ki prikazuje povprečni čas in standardni odklon za navpično porazdeljene točke:



Slika 20: Graf povprečnega časa in standardnega odklona za navpično porazdeljene točke

Rezultati na centralni procesni enoti so za polovico manjši kot pri naključnih številkah. Tudi grafična kartica se je odrezala bolje. Glavno pohitritev pripisujemo temu, da je pri računanju razdalje med točkami razlika med točkami koordinate `y` vedno enaka 0. Zaradi tega je računanje razdalje med točkama hitrejše, saj procesor pri kvadriranju števila 0 porabi manj ciklov.

Zaradi velike pohitritve izvajanja na centralni procesni enoti je bila grafična kartica boljša le

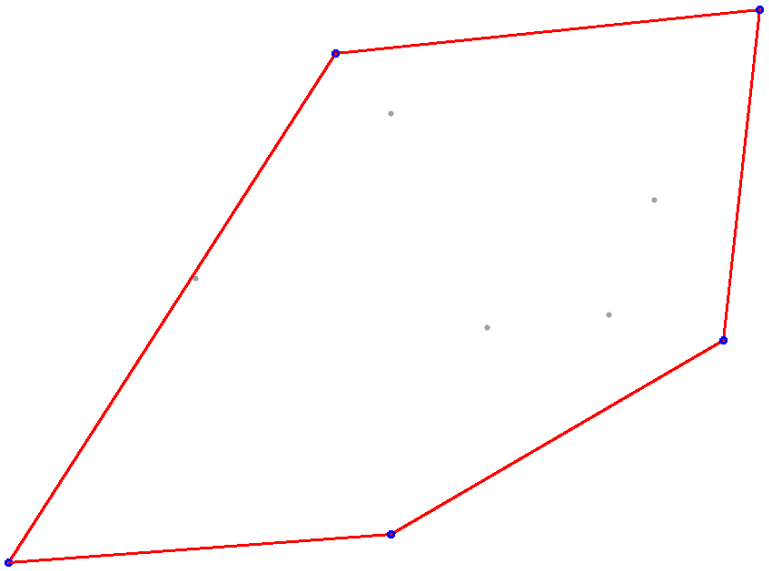
pri testu z 1 000 000 točkami.

Smo pa pri tem testiranju imeli probleme z meritvijo hitrosti na procesorju pri velikem številu točk. Če smo pognali petdeset meritev zaporedoma, se je čas izvajanja s številom meritev drastično zmanjševal in tako je bil standardni odklon nenormalno velik. Zaradi tega smo morali meritve delati s petimi hkratnimi meritvami in rezultate na koncu združiti.

4.4 Testiranje algoritma konveksne ovojnice

Pri tem testu smo testirali algoritem konveksne ovojnice z implementacijo na centralni procesni enoti in grafični kartici.

Pravilnost rezultatov smo preverili s knjižnico OpenCV. OpenCV je obsežna odprtokodna knjižnica za računalniški vid. Ker pa jo uporabljamo samo v testih, nas njena prevelika obsežnost ne moti. Z njo smo narisali točke in označili konveksno ovojnico (slika 21) [13].



Slika 21: Slika konveksne ovojnice, narejena z OpenCV

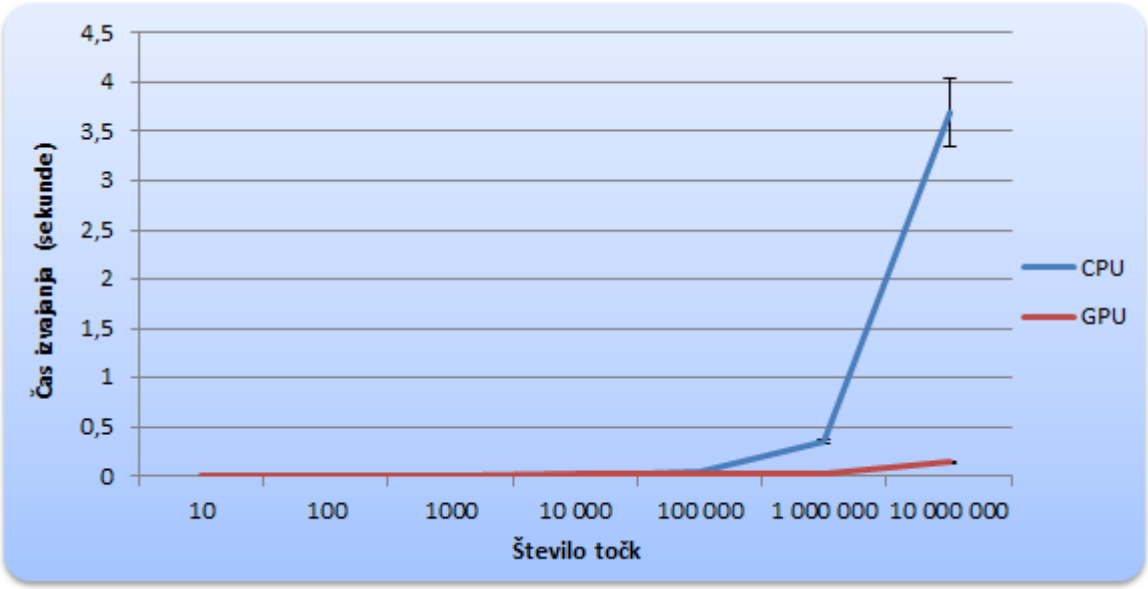
4.4.1 Testiranje na naključnih točkah

Pri tem testu smo točke generirali tako, da smo za točke koordinat `x` in `y` vzeli naključno

vrednost. V spodnji tabeli so prikazani povprečni časi in standardni odklon glede na število točk.

Naprava	Centralna procesna enota		Grafična kartica	
n	povprečje [s]	standardni odklon [s]	povprečje [s]	standardni odklon [s]
10	0,0002	0,0004	0,0044	0,00095
100	0,000333	0,000741	0,00827	0,002
1 000	0,000933	0,000249	0,0134	0,005
10 000	0,0046	0,000712	0,0195	0,0034
100 000	0,039	0,0036	0,029	0,0073
1 000 000	0,35	0,025	0,036	0,0077
10 000 000	3,695	0,348	0,1422	0,009

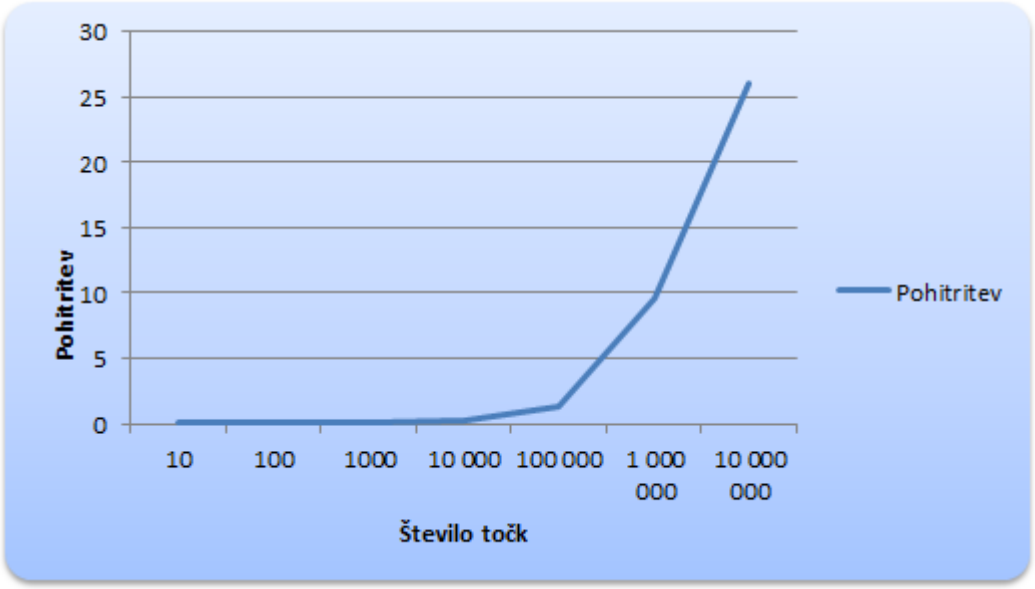
Na podlagi rezultatov smo nato izrisali graf (slika 22), ki nam pokaže čas izvajanja algoritma na centralni procesni enoti in grafični kartici za različno število točk. Prikazani so povprečni časi in standardni odklon.



Slika 22: Graf izvajanja algoritma v sekundah za algoritem hitre konveksne ovojnice

Pripravili smo tudi graf (slika 23), ki prikazuje pohitritev algoritma, pognanega na grafični

kartici, v primerjavi s centralno procesno enoto.

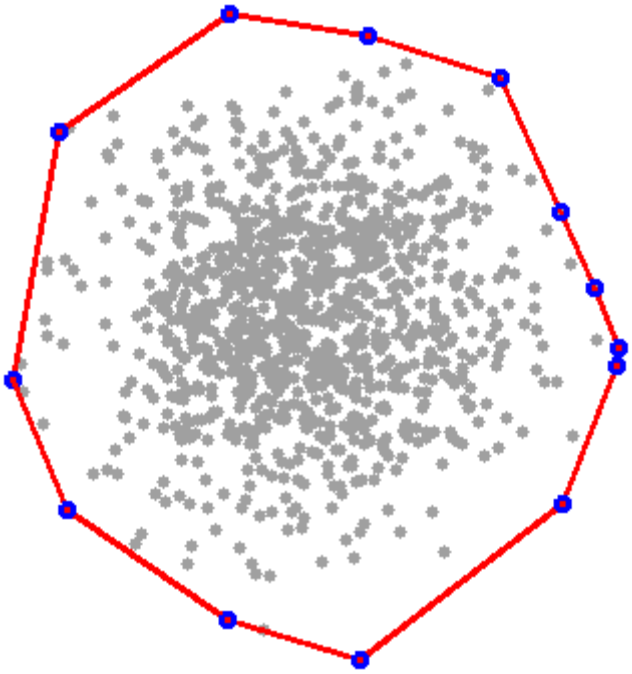


Slika 23: Graf pohitritve za hitro konveksno ovojnico

Iz grafa je dobro razvidno, da se je tudi tu grafična kartica bolje odrezala pri velikih testnih množicah, kjer režija procesorja ne pride več tako do izraza. Prav tako je manjši standardni odklon, kar nakazuje na bolj podobne rezultate med testiranjem. Tudi v tem primeru se pri majhnih problemih splača uporabiti implementacijo za centralno procesno enoto, za velike probleme pa implementacijo na arhitekturi CUDA.

4.4.2 Testiranje na normalni porazdelitvi točk

Pri tem testu smo imeli točke razporejene normalno (slika 24). To pomeni, da smo koordinati `x` in `y` izračunali po normalni porazdelitvi. Povprečje normalne razporeditve je 0, s standardnim odklonom 50.

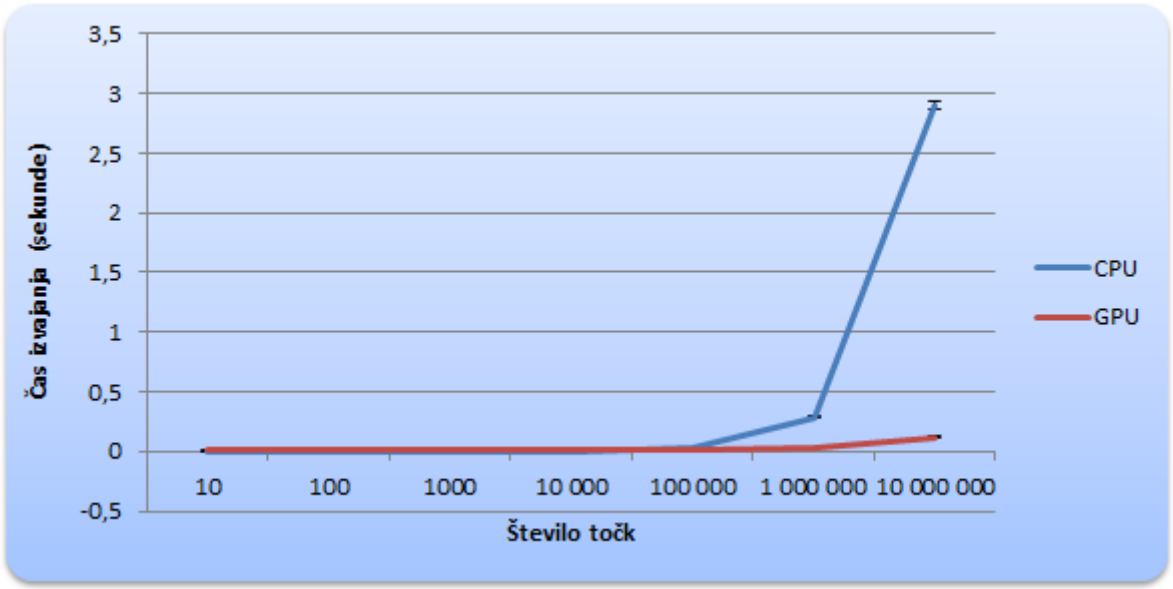


Slika 24: Normalno porazdeljene točke

Rezultati testiranja na normalni porazdelitvi so:

Naprava	Centralna procesna enota		Grafična kartica	
n	povprečje [s]	standardni odklon [s]	povprečje [s]	standardni odklon [s]
10	0,0014	0,00035	0,005	0,0014
100	0,00018	0,0038	0,006	0,002
1 000	0,0008	0,0004	0,01	0,003
10 000	0,0036	0,0005	0,011	0,005
100 000	0,03	0,0006	0,0013	0,003
1 000 000	0,29	0,005	0,027	0,003
10 000 000	2,9	0,026	0,12	0,004

Graf povprečnega časa pri normalni porazdelitvi:



Slika 25: Graf časa izvajanja algoritma in standardnega odklona pri normalni porazdelitvi

Kot je razvidno iz grafa, so časi izvajanja boljši na obeh arhitekturah. Boljši rezultati so zaradi tega, ker se večino točk nahaja okoli povprečja normalne porazdelitve. Zaradi tega se večina točk nahaja okoli sredine in tako hitro padejo znotraj začasne konveksne ovojnice in izpadejo kot možni kandidati za člana ovojnice.

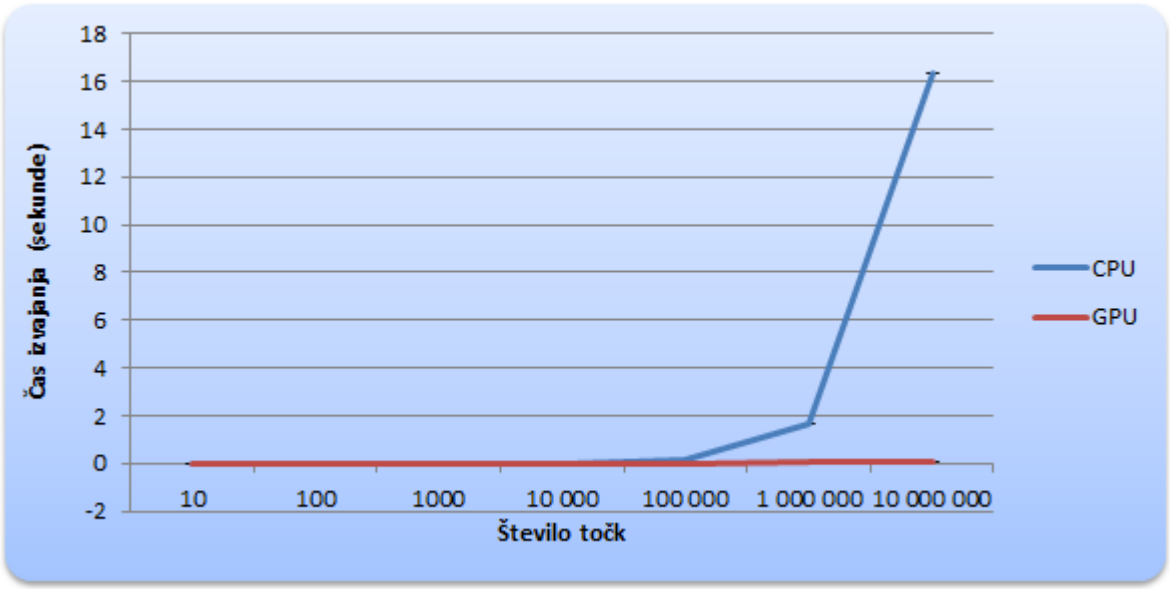
4.4.3 Testiranje najslabšega možnega primera

Pri tem testu smo točke generirali na najslabši možni primer, ki smo ga opisali pri časovni analizi hitre konveksne ovojnice.

Rezultati testiranja so prikazani v spodnji tabeli.

Naprava	Centralna procesna enota		Grafična kartica	
n	povprečje [s]	standardni odklon [s]	povprečje [s]	standardni odklon [s]
10	0,0002	0,0004	0,0099	0,00123
100	0,000467	0,0005	0,015	0,00411
1 000	0,00226	0,00044	0,015	0,0042
10 000	0,0172	0,001	0,016	0,0031
100 000	0,16	0,0012	0,015	0,0052
1 000 000	1,64	0,007	0,024	0,005
10 000 000	16,36	0,02	0,088	0,004

Pripravili pa smo tudi graf, ki nam pokaže čas izvajanja algoritma na centralni procesni enoti in grafični kartici, za različne velikosti testnih množic.



Slika 26: Graf izvajanja algoritma v sekundah za algoritem hitra konveksna ovojnica za najslabši primer

Kot je razvidno iz grafa, je rezultat procesorja veliko slabši kot pri prejšnjem testu. To je bilo tudi za pričakovati, saj se je algoritem izjalovil. Vidi se tudi, da je veliko manjši standardni odklon. Ta je manjši zato, ker vedno delamo z enakimi podatki in je oblika rekurzivnega drevesa vedno enaka.

Tudi pri tem testu smo imeli probleme pri večkratnem ponavljanju testa, saj se je čas izvajanja krajšal. Tudi tukaj smo ta problem rešili tako, da smo hkrati zaporedoma izvajali le pet meritev.

Presenetili so nas rezultati na arhitekturi CUDA. Rezultati so bili pri velikih številkah boljši kot pa pri testu z naključnimi točkami. Sklepamo, da do tega pride zaradi zasnove našega algoritma, saj en naš ščepec ne računa celotnega rekurzivnega nivoja. V teoriji najslabši primer se je izkazal kot za pisanega na kožo grafični kartici.

Poglavje 5

Zaključek

V diplomski nalogi smo predstavili dva algoritma s teoretičnega in praktičnega stališča. Algoritma sta bila implementirana za izvajanje na centralni procesni enoti in arhitekturi CUDA, ki teče na grafični kartici. Algoritma rešujeta problema s področja računske geometrije. Njuna pričakovana časovna zahtevnost je $O(n \log n)$.

Prvi algoritem se imenuje iskanje najbližjega para. Algoritem nam v množici točk poišče točki, ki imata najmanjšo evklidsko razdaljo. Pri tem algoritmu smo ugotovili, da je izvajanje na grafični kartici učinkovitejše le pri veliki množici točk v prostoru. Za manjše probleme je učinkovitejša implementacija algoritma na centralni procesni enoti. Pri velikih problemih je bolje uporabiti implementacijo na arhitekturi CUDA.

Drugi problem, ki smo ga reševali, je bil iskanje konveksne ovojnice. Algoritem nam poišče najmanjši konveksni poligon. Za ta poligon velja, da so vse točke znotraj oziroma del ovojnice. Tudi pri tem algoritmu se je izkazalo, da je rešitev na arhitekturi CUDA učinkovita le pri veliki množici točk.

Iskanje konveksne ovojnice smo implementirali z algoritmom hitre konveksne ovojnice. Algoritem ima enak problem kot hitro urejanje. Ob izbiri napačnega pivota se nam rekurzivno drevo izjalovi. V takem primeru imamo časovno zahtevnost $O(n^2)$. Pri hitrem urejanju lahko ta problem rešimo s pametnejšo izbiro pivota. V tem primeru to ni mogoče, saj mora biti pivot najbolj oddaljena točka v prostoru.

Na splošno se arhitektura CUDA dobro obnese pri velikih problemih, saj se pri izvajanju majhnih problemov ustvarja relativno veliko režije. K režiji štejemo prenos podatkov na napravo in iz nje, sinhronizacijo in prevajanje vmesne kode PTX v strojno kodo.

Ker obstaja veliko različnih algoritmov za iskanje konveksne ovojnice, bi bilo zanimivo implementirati nekaj najpomembnejših. Na podlagi testiranj bi nato lahko ugotovili, za kakšne množice točk se kateri dobro obnese.

Viri

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to algorithms, Cambridge (Massachusetts), London : The MIT Press, 2009.

[2] Algoritmi za reševanje konveskne ovojnice. Dostopno 2013 na:

http://en.wikipedia.org/wiki/Convex_hull_algorithms.

[3] Amdalov zakon. Dostopno 2013 na:

http://en.wikipedia.org/wiki/Amdahl's_law.

[4] Aritmetična sredina. Dostopno 2013 na:

http://sl.wikipedia.org/wiki/Aritmeti%C4%8Dna_sredina.

[5] Bitcoin spletna valuta. Dostopno 2013 na:

http://www.bitcoin.si/faq.php.

[6] CUDA izvajalni model. Dostopno 2013 na:

http://www.uni-graz.at/~liebma/CUDA/NVISION08-Getting_Started_with_CUDA.pdf.

[7] Dinamični paralelizem. Dostopno 2013 na:

http://blogs.nvidia.com/blog/2012/09/12/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code/.

[8] Grafična kartica. Dostopno 2013 na:

http://wiki.fmf.uni-lj.si/wiki/Grafi%C4%8Dna_kartica.

[9] Grafična kartica Gigabyte GV-N660OC-2GD. Dostopno 2013 na:

http://www.gigabyte.com/products/product-page.aspx?pid=4361#ov.

[10] Knjižnica Thrust. Dostopno 2013 na:

http://thrust.github.io/.

[11] Najbližji par točk. Dostopno 2013 na:

http://wiki.fmf.uni-lj.si/wiki/Par_najbli%C5%BEjih_to%C4%8Dk.

[12] O CUDI in njene verzije. Dostopno 2013 na:

http://en.wikipedia.org/wiki/CUDA.

[13] OpenCV. Dostopno 2013 na:

http://code.opencv.org/projects/opencv/wiki.

[14] Praktična uporaba konveksne ovojnice. Dostopno 2013 na:

http://www.tcs.fudan.edu.cn/rudolf/Courses/Algorithms/Alg_ss_07w/Webprojects/Chen_hull/applications.htm.

[15] Procesor. Dostopno 2013 na:

http://sl.wikipedia.org/wiki/Procesor.

[16] Procesor Intel i7-3770K. Dostopno 2013 na:

http://ark.intel.com/products/65523.

[17] Hitra konveksna ovojnica najslabši primer. Dostopno 2013 na:

http://stackoverflow.com/questions/13329345/quick-hull-worst-case-explanation.

[18] Redukcija. Dostopno 2013 na:

http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[19] Shema pomnilniške hierarhije. Dostopno 2013 na:

http://www2.engr.arizona.edu/~yangsong/gpu.htm.

[20] Standardni odklon. Dostopno 2013 na:

http://sl.wikipedia.org/wiki/Standardni_odklon.

[21] OpenCL vs CUDA. Dostopno 2013 na:

http://www.keremcaliskan.com/gpgpu-opencl-vs-cuda-vs-arbb/.

[22] mag. Igor Škraba, Interno gradivo Pomnilniška hierarhija

[23] Algoritmi računske geometrije. Dostopno 2013 na:

http://geomalgorithms.com/.

[24] Master teorem. Dostopno 2013 na:

http://www.cs.ucdavis.edu/~amenta/w10/masterHandout.pdf.

[25] Knjižnice CUDA. Dostopno 2013 na:

https://developer.nvidia.com/gpu-accelerated-libraries.