Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matevž Pavlič

# Izračun triangulacije točk v ravnini z uporabo grafičnega procesorja

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: pred. dr. Boštjan Slivnik

Ljubljana 2012

*Besedilo je oblikovano z urejevalnikom besedil LaTeX.*

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: 00260/2012
Datum: 10.04.2012

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

Kandidat: **MATEVŽ PAVLIČ**

Naslov: **IZRAČUN TRIANGULACIJE TOČK V RAVNINI Z UPORABO GRAFIČNEGA PROCESORJA**

**COMPUTING THE MINIMAL WEIGTH TRIANGULATION USING GRAPHICS PROCESSING UNIT**

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Napišite program, ki izračuna triangulacijo izbranih točk v realni ravnini. Izračunana triangulacija naj ima najmanjšo možno skupno dolžino vseh daljic, ki sestavljajo triangulacijo. Ker je ta problem NP težak, za izračun uporabite grafični procesor in knjižnico CUDA. Ugotovite, ali je problem triangulacije primeren za izračun z uporabo grafičnega procesorja ter primerjajte čas izračuna z grafičnim procesorjem in brez njega.

Mentor:
B. Slivnik
pred. dr. Boštjan Slivnik

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

# Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matevž Pavlič, z vpisno številko **63080304**, sem avtor diplomskega dela z naslovom:

*Izračun triangulacije točk v ravnini z uporabo grafičnega procesorja*

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom pred. dr. Boštjana Slivnika,
- 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 24. septembra 2012 Podpis avtorja:

# Zahvala

Zahvalil bi se mentorju pred. dr. Boštjanu Slivniku za pomoč pri izdelavi diplomske naloge.

# Kazalo

# Povzetek

V diplomskem delu obravnavamo problem izračuna triangulacije točk v realni ravnini z uporabo grafičnega procesorja, natančneje okolja CUDA. Pri tem iščemo triangulacijo z najmanjšo možno skupno dolžino vseh povezav. Tako definiran problem je NP težak. Obstaja več različnih metod za njegovo reševanje, ni pa znano, katere od njih so primerne za izvedbo na grafičnem procesorju.

V tem delu obravnamo tri metode. Za dve metodi smo realizirali zaporedni in vzporedni algoritem, zadnji je namenjen le izvajanju na grafičnem procesorju. Izvedli smo vrsto testiranj z namenom optimizacije vzporednih algoritmov in primerjave njihove učinkovitosti.

Vzporedni algoritem je bil hitrejši samo pri eni metodi. Ker pa je ravno ta algoritem na zaporednem rač. sam po sebi najmanj učinkovit, je tudi čas izvajanja vzporedne različice še vedno daljši od časa izvajanja zaporednega algoritma druge metode. Za drugi dve metodi nam ni uspelo najti hitrega vzporednega algoritma. Razlog je v tem, da sta rekurzivnega tipa. Zato ju ni mogoče implementirati na tak način, da bi imeli majhno število vejitev, malo število vejitev pa je glavni pogoj za učinkovito izvajanje na grafičnem procesorju.

Delo sicer ni privedlo do učinkovitejših algoritmov, vendar je tudi negativni rezultat - torej ugotovitev, da je učinkovite zaporedne metode težko izvesti vzporedno - sam po sebi zanimiv in lahko služi kot izhodišče za prihodnje delo na tem področju.

## Ključne besede

triangulacija, iskanje MWT, CUDA

# Abstract

The thesis deals with the problem of triangulation in a real plane using a graphics processing unit, specifically the CUDA architecture. An additional requirement posed is that the calculated triangulation should have the least possible total edge length. The problem thus defined is of the NP complexity. There are a number of different methods for reaching the desired solution, but we do not know which of these are appropriate for running on a graphics processing unit.

In this paper we consider three methods. For two of these we have developed a sequential and a parallel algorithm; the latter is intended for running on the graphics processing unit. A series of tests were also carried out, with the purpose of optimising and comparing the efficiency of the parallel algorithms.

The parallel algorithm was faster only with one of the methods. As this method is least efficient on sequential computer, the time this algorithm requires for execution is still longer than the execution time of the sequential algorithm with the other method. We did not manage to find a fast enough parallel algorithm for the other two methods. This is because they are of the recursive type and cannot be implemented in a way that would give a small number of branchings, this being the main condition necessary for an efficient execution of a parallel algorithm on a graphics unit processor. The study did not produce more efficient algorithms, but this negative result – i. e. the finding that it is very difficult to make efficient sequential methods parallel – is in itself interesting and can serve as a starting point for further

work in this area.

## Keywords

triangulation, MWT search, CUDA

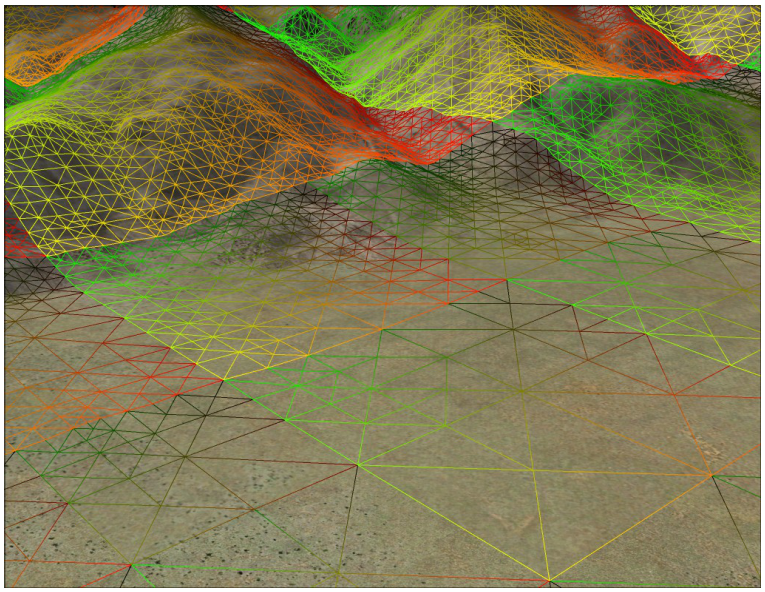
# Poglavje 1

# Uvod

V zadnjem času smo lahko spremljali velik napredek v razvoju grafičnih procesorjev (GPE) ter njihovega izkoriščanja za splošno namensko računanje. Uporabljajo se za fizikalne simulacije, pri šifrirnih/dešifrirnih algoritmih, za računanje analiz tveganj pri financah. Znano je, da GPE niso primerni za vse probleme. Zlasti se izkažejo pri problemih z visoko stopnjo podatkovne vzporednosti.

Za problem iskanja triangulacije z najmanjšo skupno dolžino povezav, znan tudi pod imenom MWT (minimum-weight triangulation), ne obstaja noben hiter algoritem, saj je problem NP težak. Vseeno pa bi radi izračunali optimalno rešitev na čimveč točkah. Iskanje takšne triangulacije se pojavlja v praksi pri stiskanju podatkov, procesiranju slik ter reševanju problemov v računalniških omrežjih. Obstajajo algoritmi, ki v primernem času izračunajo rešitev, ampak je ta približna. Rezultati optimalnih rešitev pridejo prav predvsem pri analizi uspeha takšnih hevrističnih metod. Na sliki 1.1 je predstavljen eden izmed bolj znanih primerov uporabe triangulacije, modeliranje terena.

V tem diplomskem delu smo raziskali, ali so izbrane tri metode iskanja MWT primerne za izvedbo na GPE. Na začetku dela je podrobno predstavljen problem iskanja MWT. Nato opišemo arhitekturo CUDA, kjer je tudi razloženo, kakšni tipi algoritmov so primerni za izvajanje na GPE. Četrto



Slika 1.1: Primer uporabe triangulacije pri generiranju mreže terena [10]

poglavje vsebuje kratek opis treh izbranih metod iskanja MWT, te so bile povzete po rezultatih dela Hlavaty-ja in Skale [4]. Za dve izmed teh metod tudi predstavimo našo implementacijo zaporedne izvedbe algoritma in nato postopek implementacije vzporedne izvedbe. V naslednjem poglavju so predstavljeni rezultati testiranj, ki smo jih izvedli med razvijanjem algoritmov in primerjava časov izvajanj končnih različic. Na koncu delo zaključimo s sklepom, v katerem so predstavljene glavne ugotovitve ter možno nadaljnje delo.

# Poglavje 2

# Triangulacija točk v ravnini

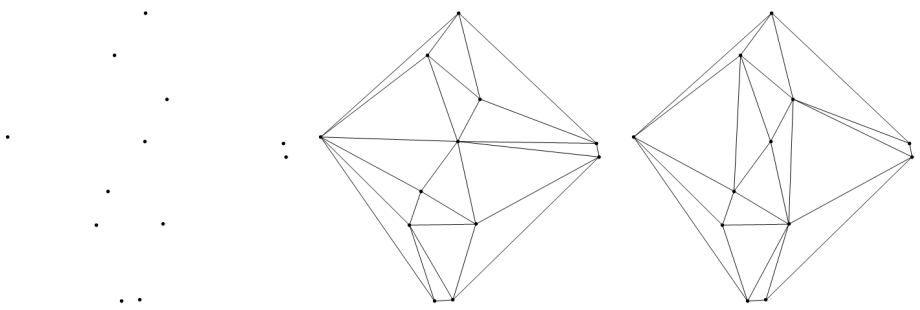
Triangulacija je razdelitev ravninskega področja na trikotnike. Malo bolj nazorna definicija pravi:

**Definicija 2.1** *Dana je končna množica $S \subset \mathbb{R} \times \mathbb{R}$. Množica $T \subseteq \mathbb{S} \times \mathbb{S}$ predstavlja triangulacijo, če veljajo naslednji pogoji:*

- *Vsaka povezava $e = < p_1, p_2 > \in T$ vključuje samo dve točki iz množice $S$, točki $p_1$ in $p_2$ pa predstavljata končni točki povezave.*
- *Dve različni povezavi v triangulaciji se ne sekata.*
- *Nemogoče je vnesti dodatno povezavo v triangulacijo.*

Na sliki 2.1 lahko vidimo dve različni triangulaciji na isti množici točk. Obstaja jih seveda še mnogo. Ti dve triangulaciji na sliki predstavljata dva posebna primera. Prva je t. i. delaunayeva triangulacija; za njo velja, da maksimizira minimalne kote v triangulaciji. Tipično se uporablja pri modeliranju terena ali drugih objektov, predstavljenih prvotno s točkami [9]. Druga triangulacija na sliki je tista, ki ima izmed vseh možnih triangulacij na podanih točkah najmanjšo skupno dolžino povezav. Kot je omenjeno že v uvodu, jo imenujemo MWT. Treba je poudariti, da je izračun MWT dosti težji problem kot pa izračun delaunayeve triangulacije.

V tem diplomskem delu se bomo ukvarjali samo z MWT. Dolgo časa je bil ta problem na seznamu odprtih NP problemov. V letu 2006 pa je prišlo



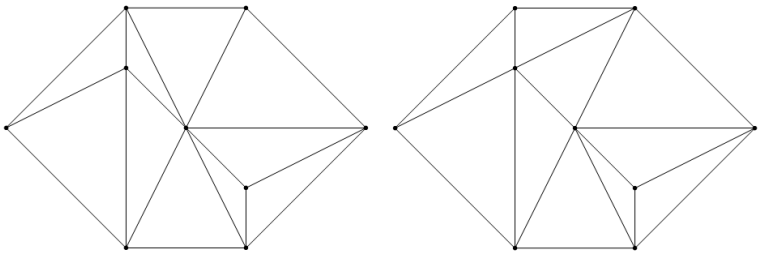
Slika 2.1: Primera triangulacije

do odkritja, da ta problem spada med NP-polne [1]. Definicija MWT se glasi takole:

**Definicija 2.2** *Triangulacija T je MWT, če za vsako drugo triangulacijo T' istih točk velja*

$$\sum_{e \in T} d(e) \leq \sum_{e \in T'} d(e).$$

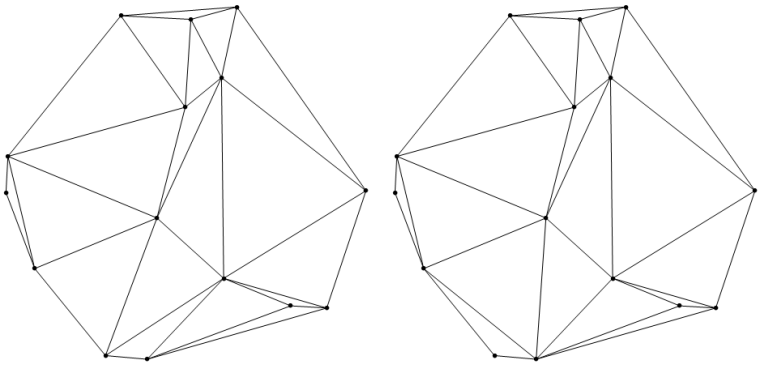
Lahko se zgodi, da definiciji 2.2 ustreza več različnih triangulacij, tak primer je prikazan na sliki 2.2. MWT se v praksi ne pojavlja toliko kot delaunayeva



Slika 2.2: Dve različni optimalni triangulaciji na isti množici točk

triangulacija. Lahko pa jo srečamo pri reševanju problemov v računalniških omrežjih, procesiranju slik in kompresiji podatkov.

Omenili smo že, da je problem izredno težak, saj čas reševanja (verjetno) narašča eksponentno s številom vhodnih točk. Zato se moramo v večini primerov zadovoljiti s približno rešitvijo. Povod za pričujoče diplomsko delo je bilo ocenjevanje teh približnih algoritmov, za kar pa je kot referenco najbolje imeti optimalno rešitev.



Slika 2.3: Levo triangulacija, pridobljena s hevristično metodo, desno optimalna triangulacija

Če pogledamo prejšnjo sliko, opazimo, da zunanje povezave v triangulaciji zmeraj predstavljajo konveksen mnogokotnik oz. konveksno ovojnico. To pomeni, da bodo robne povezave vključene v vsako možno triangulacijo, torej tudi tisto, ki jo iščemo. Na srečo se da preprosto ugotoviti, katere povezave spadajo v konveksno ovojnico, težji del za izračun optimalne triangulacije je izbira notranjih povezav.

Koliko teh povezav moramo izbrati iz celotne množice vseh povezav, nam pove naslednji izrek [4].

**Izrek 2.1** *Dana je triangulacija N točk, ki ima $N_{CH}$ točk v konveksni ovojnici. Število povezav $N_E$ in število trikotnikov $N_T$ izračunamo po naslednjih formulah:*

$$N_E = 3 \cdot (N - 1) - N_{CH} \tag{2.1}$$

$$N_T = 2 \cdot (N - 1) - N_{CH} \tag{2.2}$$

Na začetku bomo v množico izbranih povezav zmeraj dali tiste, ki se ne sekajo, saj za te velja, da so v vsaki triangulaciji, tudi v optimalni.

Zadnja stvar, ki jo je treba izpostaviti, je, da se po definiciji 2.1 v triangulaciji dve različni povezavi ne smeta sekati, kar nam znatno pomaga pri zmanjševanju iskalnega prostora.

# Poglavje 3

# CUDA

Okolje CUDA je namenjeno za vzporedno računanje na grafičnih procesorjih, ki ga je razvilo podjetje NVIDIA leta 2007. Glavni razlog, da je prišlo do razvoja tega okolja, je neprestano izboljševanje grafičnih procesorjev, ki ga je zahteval trg preko zmeraj večjih zahtev po realno časovni, visoko ločljivostni grafiki [7]. Grafični procesorji se razlikujejo od navadnih predvsem v tem, da imajo dosti več tranzistorjev, namenjenih procesiranju podatkov, dosti manj pa je predpomnenja ter krmiljenja tokov. Ta razmerja so vidna tudi na sliki 3.1. GPE ustrezajo predvsem problemi z visoko stopnjo podatkovne vzporednosti, kjer se opravi veliko zahtevnih računskih operacij. Te operacije se izvedejo za vsak podatkovni element, tukaj pa je zelo malo krmiljenja tokov.

Pri podatkovni vzporednosti več niti istočasno računa isto operacijo na


Slika 3.1: Primerjava tranzistorjev, namenjenih procesiranju

različnih podatkih. Takšen pristop pa se ne uporablja samo za grafična računanja (prepoznava vzorcev, kodiranje videa, procesiranje slik ...), ampak tudi pri simulacijah dinamike tekočin, računanju (izvedenih) finančnih inštrumentov, procesiranju signalov [6].

Računanje negrafičnih problemov na grafičnih karticah se je izvajalo že dosti pred splavitvijo okolja CUDA. Sama implementacija teh rešitev je bilo mukotrpno delo, saj je bilo brez okolja CUDA potrebno uporabljati tehnologiji OpenGL oz. DirectX ter računanja preoblikovati v grafične probleme [7]. Nvidia pa je omogočila, da lahko izkoristimo vse, kar CUDA ponuja, z uporabo uveljavljenega programskega jezika C z nekaj malega dodatnimi konstrukti.

## 3.1 Programski model

Tipično izvajanje programa, napisanega v okolju CUDA, lahko ponazorimo takole:

1. Prenos podatkov v pomnilnik GPE.
2. CPE naredi zahtevo za izračun na GPE.
3. GPE izvrši zahtevo.
4. Prenos izračunanih podatkov iz pomnilnika GPE v glavni pomnilnik.

### 3.1.1 Ščepec in organizacija niti

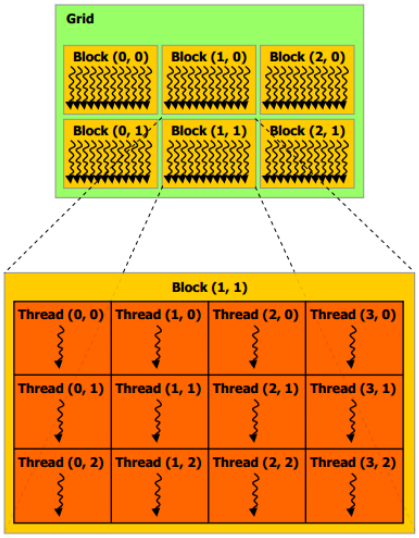
Ščepec je samostojen kos programske kode, ki ga ob klicu izvede določeno število CUDA niti. Od navadnih C funkcij se razlikuje v tem, da ima v podpisu dodano oznako `__global__`.

```
1 // Definicija scepca
2 __global__ void sestej(float* A, float* B, float* C) {
3   int i = threadIdx.x;
4   C[i] = A[i] + B[i];
5 }
```

```
6  int main() {
7    ...
8    // Klic scepca z N nitmi
9    sestej<<<1, N>>>(A, B, C);
10   ...
11 }
```

Listing 3.1: Primer ščepca, kjer vsaka nit sešteje po en istoležen element v dveh tabelah

Organizacija niti je prikazana na sliki 3.2. Vidimo, da so niti organizirane v bloke. Slika predstavlja primer dvodimenzionalnih blokov, na izbiro pa imamo še enodimenzonalne ali tridimenzionalne. Ker se vse niti bloka nahajajo na istem procesorskem jedru, lahko blok vsebuje do največ 1024 niti. Poudariti velja tudi, da te niti lahko tudi komunicirajo med seboj preko skupnega pomnilnika. Posamezne skupine blokov so nato organizirane v mrežo. Tudi mreže so lahko eno, dvo ali tridimenzionalne. V večini programov je število blokov v mreži v korelaciji s številom podatkov, namenjenih procesiranju.




Slika 3.2: Mreža blokov niti

Izvajanje posameznih blokov je neodvisno, lahko se izvajajo vzporedno ali zaporedno. Za niti znotraj blokov pa velja, da se izvajajo vzporedno. Sinhronizacija niti je možna samo znotraj posameznih blokov.

## 3.2 Arhitektura GPE

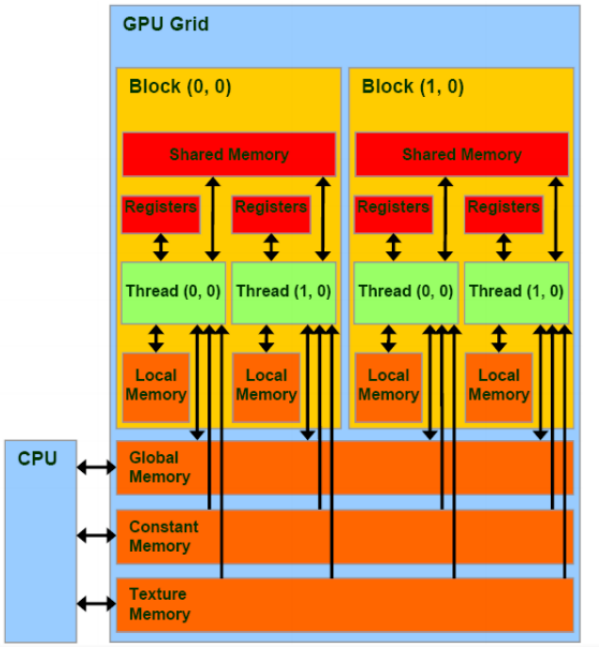
Vse CUDA grafične kartice vsebujejo več multiprocesorjev SM (Streaming multiprocessors). Tja se ob vsakem zagonu ščepca porazdelijo bloki v mreži. Niti posameznega bloka se izvajajo vzporedno na enem multiprocesorju. Ko se te končajo, multiprocesorji dobijo nove bloke za izvajanje.

Multiprocesorji so zgrajeni z namenom izvajanja več sto niti naenkrat. Da je to sploh mogoče, delujejo po principu SIMT (Single-Instruction, Multiple Thread), kar pomeni, da mora biti računanje na vseh nitih čimbolj podobno (uporabljati se morajo iste izvajalne enote za maksimalno vzporednost). Programerskim očem je skrito, da se niti na multiprocesorjih izvajajo v skupinah po 32, z drugimi besedami v snopih (warps). Celoten snop izvaja isto kodo tako, da pri nitih, ki imajo divergentne izvajalne poti, postane izvajanje neučinkovito.

## 3.3 Pomnilniška hierarhija

Na sliki 3.3 lahko vidimo pomnilniške prostore, ki se nahajajo na GPE:

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




Slika 3.3: Prikaz pomnilniških prostorov

Najkrajši dostopni čas imajo registri, ti so privatni za vsako nit. Naslednji po hitrosti dostopa je skupni pomnilnik. Ta je skupen za vse niti v bloku. Počasnejši lokalni pomnilnik se uporablja, ko zmanjka registrov, za razliko od skupnega pomnilnika je ta privaten za vsako nit. Dostop do globalnega pomnilnika je najpočasnejši. Iz njega lahko beremo in pišemo z gostiteljske CPE ter z GPE. Do njega lahko dostopajo vse niti.

Pomnilnik konstant ter pomnilnik tekstur sta še dva dodatna bralno predpomnenja pomnilna prostora, ki sta dostopna vsem nitim.

# Poglavje 4

# Iskanje optimalne triangulacije

Iščemo triangulacijo, ki ima najmanjšo možno skupno dolžino vseh povezav. Če upoštevamo ta pogoj, se izkaže, da je treba uporabiti metodo grobe sile. Ta izraz je sinonim za način reševanja problemov, pri katerem se sistematično preverja vse možnosti, dokler se ne najde rešitve. Torej generirati je treba vse možne triangulacije, jih preveriti in izbrati tisto, ki najbolje ustreza danim pogojem. Zaradi obsega dela so takšni algoritmi uporabni samo na manjših problemih. V nadaljevanju so na kratko predstavljene tri metode iskanja MWT:

1. metoda odstranjevanja povezav,

2. metoda vnašanja povezav,

3. generiranje vseh kombinacij povezav.

Te metode so bolj obširno opisane v [4]. Sledi razlaga naše zaporedne implementacije metode odstranjevanja povezav ter generiranja vseh kombinacij povezav. Metode vnašanje povezav nismo implementirali zaradi podobnosti z metodo odstranjevanja povezav. Za obe izbrani metodi opišemo tudi postopek izvedbe vzporednega algoritma.

## 4.1 Priprava na iskanje

Pri vsaki metodi pred začetkom iskanja

(a) poiščemo vse povezave, ki se ne sekajo in jih odstranimo iz nadaljnjega iskanja,

(b) izračunamo konveksno ovojnico.

Odstranitev nesekajočih povezav je potrebna, saj s tem zmanjšamo število kombinacij, ki jih je treba preveriti. Kot že omenjeno v prejšnjih poglavjih, se vse nesekajoče povezave zagotovo nahajajo tudi v triangulaciji. Med te spadajo tudi tiste v konveksni ovojnici.

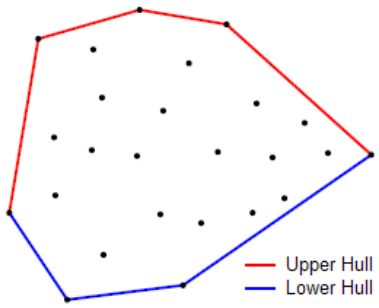
Izračun konveksne ovojnice je potreben zaradi podatka o številu povezav v konveksni ovojnici. Brez te informacije ne znamo izračunati, koliko notranjih povezav mora imeti triangulacija. Postopek iskanja konveksne ovojnice se zaradi praktičnosti izvede zmeraj na CPE.

Po izvedbi obeh postopkov sledi priprava kontrolnih struktur, ki bodo uporabljene pri iskanju triangulacije. Če teče program na CPE, lahko potem kar direktno začnemo z iskanjem, pri iskanju z GPE pa moramo te podatke prej prenesti v pomnilnik grafične kartice.

### 4.1.1 Iskanje konveksne ovojnice

Računanje konveksne ovojnice na končnem številu točk je geometrijski problem, zato spada v področje računske geometrije. Znanih je kar nekaj algoritmov, najboljši izmed njih imajo časovno zahtevnost $\mathcal{O}(nlogn)$, kjer n predstavlja število vhodnih točk. Uporabili smo algoritem z imenom "Andrew's monotone chain convex hull algorithm", ki najprej razvrsti točke leksikografsko (najprej po koordinati x, če pride tam do ujemanja, pa še po koordinati y) in zatem zgradi zgornjo ter spodnjo ovojnico [8].

Rezultat so vse točke, ki so del te ovojnice. Nas pa zanima število povezav v konveksni ovojnici, to pa je za ena večje od števila točk.



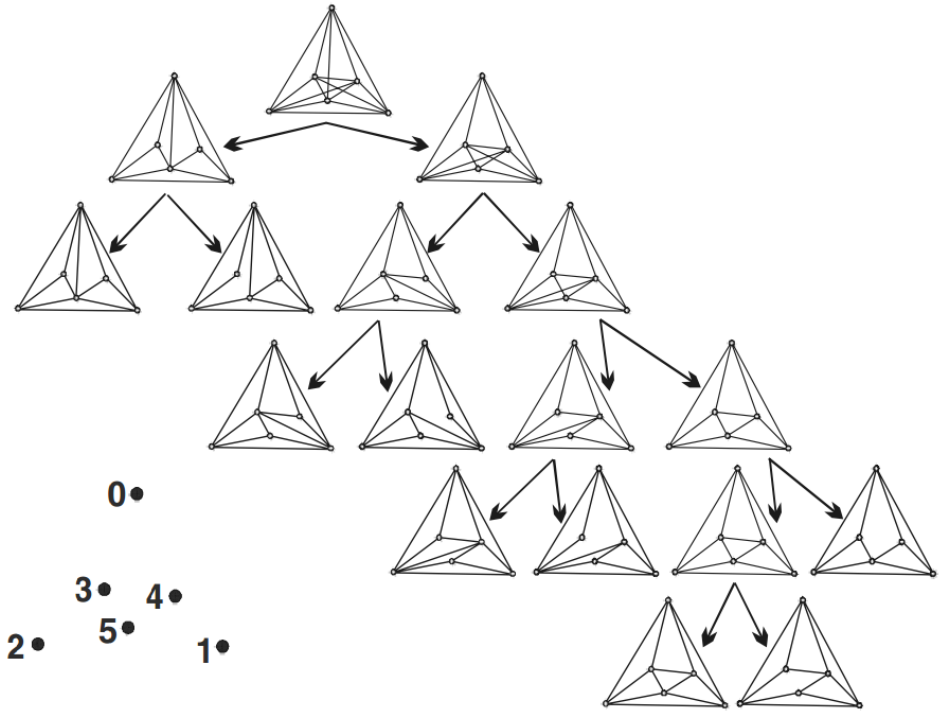

Slika 4.1: Zgornja in spodnja ovojnica množice točk.

## 4.2 Izračun MWT z odstranjevanjem povezav

Vse možne povezave med danimi točkami predstavljajo poln neusmerjen graf. Začnemo s polnim grafom, kar ponazarja, da so vse povezave v triangulaciji, nato pa postopoma izbiramo, katere povezave bodo ostale v triangulaciji in katere ne. Za vsako povezavo, dodeljeno triangulaciji, moramo tudi odstraniti vse njene sekajoče povezave. Od tu tudi prihaja ime metode.

Povezave izbiramo, dokler nimamo v grafu samo takšnih povezav, ki predstavljajo veljavno triangulacijo. Do vseh triangulacij pridemo tako, da ob izbiri povezave zmeraj nadaljujemo po dveh poteh. Pri prvi damo povezavo v triangulacijo, pri drugi pa ne.

Rezultat je dvojiško drevo stanj. V korenu drevesa imamo poln neusmerjen graf, listi drevesa pa so razdeljeni v dve skupini. Prva skupina predstavlja vse veljavne triangulacije, v drugi pa imamo vse liste, kjer se nahaja premalo povezav za veljavno triangulacijo. Izbiranje povezav, ali bodo v triangulaciji ali ne, poteka po vrsti glede na indeks povezave. Vsaka izbira predstavlja en nivo drevesa. Maksimalno število nivojev je enako številu povezav v polnem grafu, v praksi je pa to dejansko manjše zaradi lastnosti, omenjenih v poglavju 2. Na sliki 4.2 lahko vidimo primer delovanje te metode.




Slika 4.2: Prikaz metode odstranjevanja povezav

### 4.2.1 Implementacija zaporedne metode odstranjevanja povezav

Imamo globalno tabelo z dolžino enako številu vseh sekajočih povezav. Vrednosti v tabeli povedo, katere povezave so v triangulaciji ter katere niso. Na začetku nastavimo vrednosti tako, da so vse povezave v triangulaciji, torej imamo same enice.

Metodo smo implementirali z rekurzivno funkcijo, ki ima en vhodni argument. To je celo število, ki pove, od katerega indeksa v tabeli naj se prisotnost povezav v triangulaciji spreminja. Pri vsakem klicu funkcije (razen pri robnem pogoju) bomo izvedli dva nova klica te funkcije. Pri prvem bo izbrana povezava del triangulacije, pri drugem ne. Pred izvedbo klica, kjer povezavo dodelimo triangulaciji, moramo seveda odstraniti vse tiste povezave, ki sekajo izbrano povezavo. Ob vrnitvi tega klica ponastavimo vrednosti, kot so

bile, odstranimo izbrano povezavo in izvedemo še drugi klic.

Imamo dva robna pogoja rekurzije, do prvega pridemo ob prevelikem številu odstranjenih povezav (vemo, da to ne bo triangulacija), drugi, uspešnejši, je vsaka zaključena kombinacija (novi klici rekurzije bi imeli indeks začetka večji, kot pa je dolžina tabele). Pri vsaki zaključeni kombinaciji preverimo še njeno dolžino in če je ta najmanjša, si to dolžino in tabelo s povezavami shranimo kot najboljšo rešitev. Za boljšo razumljivost je priložena še psevdokoda algoritma pod imenom Algoritem 1.

**Algoritem 1** Iskanje z odstranjevanjem povezav

```
1: function FIND(i)                          ▷ i pove, od kje naprej izbiramo
2:     if edgesInTriangulation < edgesNeeded then
3:         return
4:     end if
5:     while ¬isInTriangulation[i] do        ▷ preskočimo odstranjene pov.
6:         i ← i + 1
7:     end while
8:     if i ≥ allEdges then    ▷ če smo šli čez vse povezave, preglej rešitev
9:         CheckSolution()
10:        return
11:    end if
12:
13:    RemoveSegmentsThatCross(i)
14:    Find(i + 1)
15:    RollbackSegmentsThatCross(i)
16:    isInTriangulation[i] ← false
17:    Find(i + 1)
18:    isInTriangulation[i] ← true
19: end function
```

### 4.2.2 Vzporedna izvedba metode odstranjevanja povezav

Na prvi pogled ni čisto jasno, kako bi implementirali metodo odstranjevanja povezav za vzporedno izvajanje na GPE. Odločili smo se za pristop, kjer na CPE izvajamo rekurzijo do določenega nivoja, od tam naprej pa prepustimo izvajanje GPE. Torej CPE poskrbi za veliko število vej rekurzije, ki jih potem na GPE izvajajo posamezne niti. Da se lahko veje rekurzije izvajajo na GPE, si je treba pred začetkom prenesti vrednosti spremenljivk ter kontrolnih struktur. To seveda pomeni, da velikost podatkov, ki jih je treba prenesti na GPE, hitro narašča s številom vej rekurzije.

Pojavi se še en problem, CUDA nima prav zelo dobre podpore za rekurzijo. Ta je podprta za vse GPE, ki imajo računsko zmogljivost verzije 2.x ali več [6]. Večinoma novejše GPE temu ustrezajo, ampak je sklad pri vsaki niti premajhen za uspešno izvedbo programa. Zato smo se odločili implementirati rekurzijo s skladom in `while` zanko.

### 4.2.3 Implementacija zaporedne metode odstranjevanja povezav s skladom

Glavni del tega algoritma je ponavljanje `while` zanke, dokler sklad vsebuje kakšen element. Pred začetkom izvajanja zanke se najprej naložijo začetni elementi. Elementi vsebujejo začetni indeks, od kje naprej naj se s tabelo manipulira, in oznako, kateri del kode naj se izvede v zanki. Imamo 3 možnosti:

(a) postavi povezavo v triangulacijo in odstrani sekajoče povezave;

(b) odstrani povezavo iz triangulacije in ponastavi vrednosti, ki so bile spremenjene v prejšnjem koraku;

(c) ponastavi vrednosti, kot so bile pred izvajanjem prvega koraka.

Ko se zanka konča, imamo v ustreznih spremenljivkah shranjeno kombinacijo najboljše rešitve ter najmanjšo dolžino. Psevdokoda algoritma se nahaja na

naslednji strani. Omenimo še, da bo na GPE vsaka nit izvajala svojo instanco te metode s podatki, ki bodo prenešeni z gostitelja.

### 4.2.4 Pregled vzporedne metode odstranjevanja povezav

Zdaj ko smo opisali posamezne dele, lahko naredimo še pregled čez celotno vzporedno metodo. Na začetku izračunamo konveksno ovojnico. Nato odstranimo vse sekajoče povezave iz nadaljnjega iskanja. Z zaporedno implementacijo metode odstranjevanja povezav se spustimo do želenega nivoja rekurzije, shranimo trenutne vrednosti ter prekinemo nadaljnje spuščanje. Do zdaj se je vse izvajalo na CPE. Zbrane vrednosti prenesemo z gostitelja na GPE ter zaženemo takšno število niti, kot je bilo kombinacij na izbranem nivoju rekurzije. Vse te niti izvajajo metodo odstranjevanja povezav s skladom. Tukaj pride tudi do zelo različnih časov izvajanja niti, saj nekatere hitro zaključijo z delom zaradi velikega števila odstranjenih povezav. Ko se zadnja nit zaključi, prenesemo najboljše vrednosti za vsak blok na CPE ter tam poiščemo najboljšo kombinacijo.

### 4.2.5 Mogoča izboljšava vzporedne metode odstranjevanja povezav

Poskusili smo izboljšati del, kjer odstranjujemo sekajoče povezave ter del kjer ponastavljamo te vrednosti na prvotno stanje.

Trenutno odstranjevanje sekajočih povezav poteka tako, da se sprehodimo čez tabelo povezav, ki sekajo izbrano povezavo, ter za vsako vrednost v tabeli (oznaka povezave) odstranimo to povezavo iz triangulacije. Vse to je možno nadomestiti z eno samo logično operacijo nad dvema celima številoma. Namreč vsak bit celoštevilske vrednosti lahko predstavlja eno povezavo. Pri tem prva celoštevilska vrednost hrani trenutne povezave v triangulaciji (bit, postavljen na mestu i, pomeni, da je povezava z oznako i v triangulaciji), druga vrednost pa pove, katere povezave so sekajoče z izbrano povezavo. Za vsako

**Algoritem 2** Iskanje z odstranjevanjem povezav s skladom

```
1: function FindWithStack
2:     PushOnStack(0, c)                ▷ na sklad postavimo začetne elemente
3:     PushOnStack(0, b)                ▷ argumenta sta indeks ter ime operacije
4:     PushOnStack(0, a)
5:     while ¬StackIsEmpty() do
6:         popEl ← PopFromStack()
7:         i ← popEl.i
8:         if popEl.op = a then
9:             RemoveSegmentsThatCross(i)
10:        else if popEl.op = b then
11:            isInTriangulation[i] ← false
12:            RollbackSegmentsThatCross(i)
13:        else if popEl.op = c then
14:            isInTriangulation[i] ← true
15:            continue
16:        end if
17:        if edgesInTriangulation < edgesNeeded then
18:            continue
19:        end if
20:
21:        i ← i + 1
22:        while ¬isInTriangulation[i] do
23:            i ← i + 1                ▷ preskočimo odstranjene pov.
24:        end while
25:        if i ≥ allEdges then
26:            CheckSolution()
27:            continue
28:        end if
29:        PushOnStack(i, c)
30:        PushOnStack(i, b)
31:        PushOnStack(i, a)
32:    end while
33: end function
```

dodajanje povezave k triangulaciji izvedemo ustrezno logično operacijo. Ta spremeni bite v prvi vrednosti tako, da nimamo več sekajočih povezav dodane povezave.

Pri tem nastane manjši problem, saj ni možno na enak način tudi povrniti vrednosti, zato moramo število, ki hrani triangulacijo pred spremembo, shraniti na sklad, kar nam prej ni bilo treba. Potrebno se je tudi zavedati, da zdaj ne hranimo več števila odstranjenih povezav kot v prejšnjem algoritmu. Tam smo to število uporabljali v pogoju, ki je preverjal, ali smo odstranili že preveč povezav in v tem primeru tudi zaključili s spuščanjem v globino. Tega tu ni več in zato tudi preiskujemo več prostora kot v starem algoritmu.

## 4.3 Izračun MWT z vnašanjem povezav

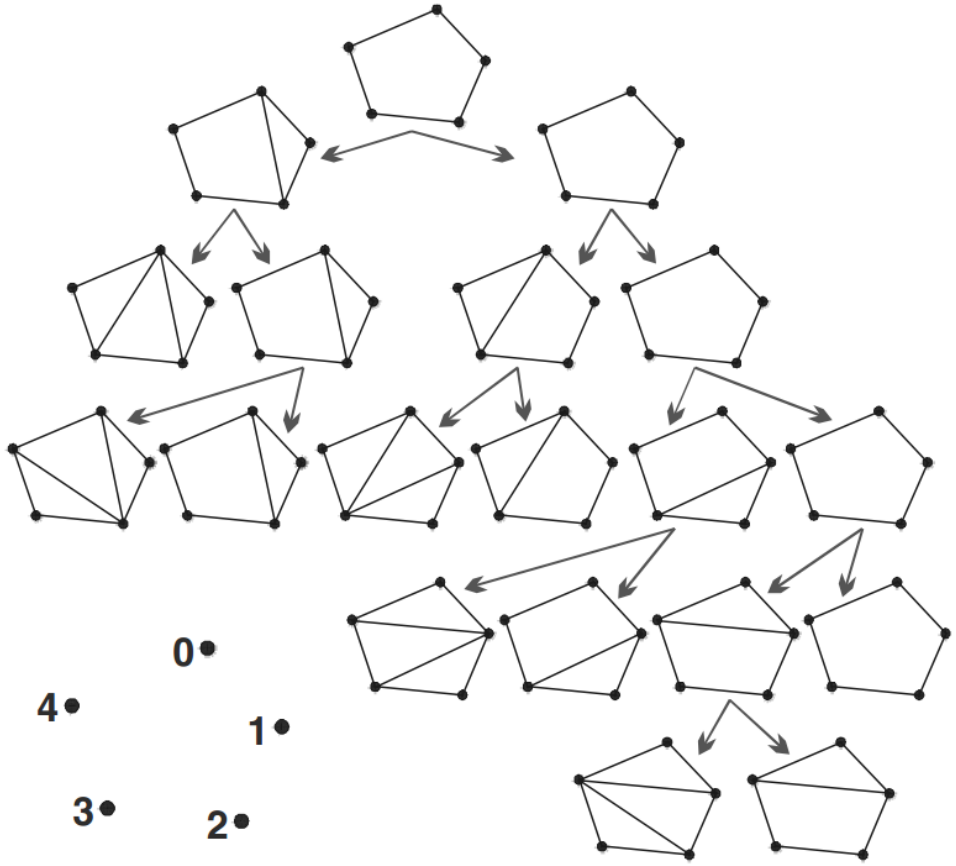
Ta metoda je zelo podobna prejšnji. Algoritem je enak, razlika je le v začetnem stanju. Začnemo s praznim grafom (graf brez povezav) in tako tudi dobimo drugačno drevo stanj. Slika 4.3 prikazuje delovanje te metode.

Rezultati, predstavljeni v [4] povedo, da je ta metoda počasnejša od metode odstranjevanja povezav. Zaradi podobnostih obeh metod tukaj nismo naredili implementacije. V primeru, da bi to storili, bi pri implementaciji vzporedne verzije spet naleteli na probleme zaradi vejitev.

## 4.4 Izračun MWT z generiranjem kombinacij povezav

Postopek generiranja kombinacij povezav je preprostejši od prejšnjih dveh metod. Kombinacije povezav predstavljajo, katere povezave so v triangulaciji. Večina kombinacij bo odvečnih, saj ne bodo predstavljale triangulacij. Vemo, da bo med njimi zagotovo tudi MWT.

S pomočjo enačb v poglavju 2 si poglejmo, koliko bo teh kombinacij. V enačbi (1) je zapisano, da imajo vse triangulacije enako število povezav $N_E$. Vemo tudi, da je število vseh različnih možnih povezav med točkami, ki




Slika 4.3: Drevo stanj metode vnašanja povezav

obenem tvorijo poln graf, enako

$$n = \binom{N}{2} = \frac{N \cdot (N-1)}{2}, \tag{4.1}$$

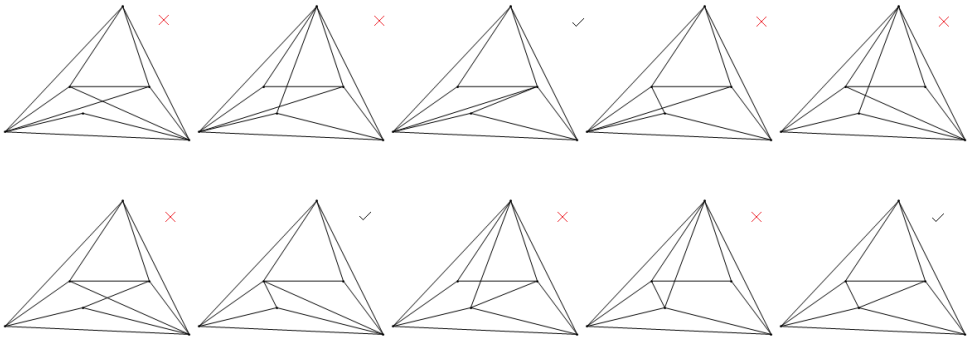
kjer N predstavlja število točk.

Če vsako povezavo označimo s svojo oznako, potem lahko problem generiranja kombinacij povezav prevedemo na problem izbire $N_E$ različnih oznak iz začetnega kupa n oznak. Tukaj vsaka kombinacija predstavlja možno veljavno triangulacijo, število teh kombinacij je enako binomskemu koeficientu n nad k, definiranem kot

$$\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}, k = N_E - N_{CE}, \tag{4.2}$$

kjer je n število povezav v polnem grafu, $N_E$ število povezav v triangulaciji ter $N_{CE}$ število skupnih povezav vsem triangulacijam na teh točkah, to so tiste, ki sestavljajo konveksno ovojnico, ter vse povezave, ki se ne sekajo.

Na sliki 4.4 lahko vidimo primer generiranja kombinacij, kjer izbiramo dve povezavi izmed sedmih. Iz enačbe 4.2 je razvidno, da število kombinacij



Slika 4.4: Primer generiranja kombinacij dveh povezav izmed sedmih

narašča eksponentno s številom povezav. To pomeni, da bo ta metoda manj učinkovita od metode odstranjevanja povezav, saj le-ta deluje po principu "razveji in omeji", kjer lahko občutno zmanjšamo iskalni prostor.

### 4.4.1 Implementacija zaporednega generiranja kombinacij povezav

Dejansko imamo pred sabo čisto kombinatoričen problem. Poiskati moramo vse različne k-podmnožice množice $S = \{1, \ldots, n\}$. Pri tem je k število povezav, ki jih zahteva triangulacija, ter n število vseh možnih povezav. Med vsemi temi kombinacijami so zajete tudi vse veljavne triangulacije. Te predstavljajo majhen delež vseh kombinacij. Tabela 4.1 prikazuje primer vseh k-podmnožic na n elementih, kjer je k enak 3 ter n enak 5. K-podmnožice so razvrščene leksikografsko.

V implementaciji iskanje začnemo s kombinacijo reda 0. Ta zmeraj predstavlja k-podmnožico, ki vsebuje števila od 1 do k, razvrščena po velikosti.

| k – podmnozice | red |
|---|---|
| {1, 2, 3} | 0 |
| {1, 2, 4} | 1 |
| {1, 2, 5} | 2 |
| {1, 3, 4} | 3 |
| {1, 3, 5} | 4 |
| {1, 4, 5} | 5 |
| {2, 3, 4} | 6 |
| {2, 3, 5} | 7 |
| {2, 4, 5} | 8 |
| {3, 4, 5} | 9 |

Tabela 4.1: Prikaz vseh 3-podmnožic množice s 5 elementi

V samem programu hranimo k-podmnožico v tabeli števil s k + 1 elementi (element z indeksom 0 se ne uporablja). Nato preverimo, če imamo veljavno triangulacijo; če je to res, si zapomnimo trenutno kombinacijo in dolžino vseh povezav. Nadaljujemo s klicanjem funkcije, ki v tabelo zapiše leksikografskega naslednika prejšnje podmnožice, spet preverimo, če je to veljavna triangulacija in v pozitivnem primeru preverimo še, če je dolžina povezav v triangulaciji manjša od trenutno najboljše. Nadaljujemo v tem stilu, dokler nismo generirali vseh kombinacij, teh pa je $\binom{n}{k}$. Postopek je prikazan tudi v psevdokodi z imenom Algoritem 3.

### 4.4.2 Implementacija vzporednega generiranja kombinacij

Vemo, da algoritmi z vejitvami niso primerni za izvajanje na arhitekturi CUDA, zato tudi ni veliko upanja v pohitritev prejšnjih dveh metod. Eden izmed primernih algoritmov bi lahko bil generator vseh kombinacij. Ideja je, da bi zagnali toliko niti, kot je kombinacij, kjer bi potem vsaka preverila,

**Algoritem 3** Iskanje z generiranjem kombinacij

```
1: function FIND
2:     numCombinations ← BinCoef(n, k)
3:     for i ← 1, k do
4:         T[i] ← i
5:     end for
6:     CheckSolution(T)
7:     for i ← 1, numCombinations do
8:         KSubsetLexSuccessor(T, k, n)
9:         CheckSolution(T)
10:    end for
11: end function
```

če je ta kombinacija res triangulacija in v tem primeru shranila rezultat. CUDA je dobro prilagojena obvladovanju velikega števila niti, medtem ko se pri CPE zadeva lahko zelo upočasni, saj je veliko dela s preklapljanjem med posameznimi nitmi. Vseeno pa število kombinacij narašča eksponentno glede na število točk, zato bomo omejili število niti na maksimalno velikost reda 10-20 tisoč. Vsaka bo potem obdelala svoj delež kombinacij. Iz tega sledi, da ima vsaka nit različno začetno kombinacijo. Tukaj pridemo do problema, kako izračunati začetno kombinacijo na podlagi indeksa te niti. V naši implementaciji smo uporabili algoritem, ki sta ga opisala Kreher in Stinson [5]. Algoritmi, ki kot vhod sprejmejo številko reda (rank) ter vrnejo kombinatorični objekt s tem redom, se imenujejo "unranking algorithms".

V izseku kode 4.1 je prikazana funkcija, ki v dano tabelo zapiše kombinacijo z redom r. Ker funkcija vsebuje vejitve, ne vemo ali bo izračun začetnih kombinacij hitrejši na CPE, ali na GPE. V primeru, da se začetne kombinacije izračunajo na CPE, jih je treba še prenesti iz gostitelja na GPE. Do odgovora katera verzija je hitrejša, bomo najlažje prišli s testiranjem obeh.

V nadaljevanju je opisana verzija ščepca, kjer se začetne kombinacije izračunajo na GPE. Najprej se izračuna začetna kombinacija. Nato se preveri ali je ta kombinacija tudi triangulacija. Če je to res, se ta kombinacija

```
1  void kSubsetLexUnrank(long long r, short *T) {
2    int x, i, y;
3    x = 1;
4    for (i = 1; i <= k; i = i + 1) {
5      y = BinCoef(n - x, k - i);
6      while (y <= r) {
7        r = r - y;  x++;
8        y = BinCoef(n - x, k - i);
9      }
10     T[i] = x; x++;
11   }
12 }
```

Listing 4.1: Funkcija ki v tabelo T zapiše k-podmnožico množice n z redom r

ter njena skupna dolžina shrani v skupni pomnilnik za nadaljnje primerjave. Potem pride na vrsto `while` zanka, kjer ščepec prebiva večino časa. V tej `while` zanki vsakič generiramo naslednika ter preverimo ali predstavlja triangulacijo. Zanka se izvede $\frac{st.kombinacij}{st.niti}$-krat oz. pri zadnji niti ponavadi malo manjkrat.

Trenutno še ne vemo, katera je najboljša kombinacija, vemo le najboljše kombinacije za vsako nit. Namesto da se sprehodimo čez vse te rezultate, lahko poiščemo najboljši rezultat za vsak blok v logaritemskem času, saj se da ta postopek izvesti vzporedno. Vsaka nit bo primerjala dva rezultata ter shranila boljšega. S tem se v vsakem koraku znebimo polovico vrednosti. Torej bomo rabili `log2(št. niti v bloku)` korakov, medtem ko bi nam navadna iteracija vzela čas, proporcionalen številom niti v bloku.

Na koncu imamo v skupnem pomnilniku shranjene najboljše vrednosti za vsak blok. Tega se nam ne splača pregledovati na GPE, zato te vrednosti prenesemo na gostitelja ter tam z navadno iteracijo poiščemo najboljšo triangulacijo.

```
1  // funkcija, ki izracuna naslednika in ga zapise v tabelo T
2  // U je pomozna tabela, mem_off pove odmik v tabelah T in U
3  __device__ void kSubsetLexSuccessor(short* U, int mem_off) {
4    short i, j;
5    int mem_off_U = mem_off + (k + 1);
```

```
6     // prepisemo staro kombinacijo iz T v U
7     for(i=1; i<=k; i++)
8       U[mem_off_U + i]=T[mem_off + i];
9
10    i = k;
11    // izracun
12    while ((i >= 1) && (T[mem_off + i] == (n - k + i)))
13      i = i - 1;
14    for (j = i; j <= k; j++)
15      U[mem_off_U + j] = T[mem_off + i] + 1 + j - i;
16    // zapisemo novo kombinacijo
17    for (j = 1; j <= k; j++)
18      T[mem_off + j] = U[mem_off_U + j];
19  }
20
21
22  __global__ void kernelGenCombinations(char *dev_crosses,
23    double *dev_seg_weight,  double *dev_best_weight,
24    short *dev_best_solution, int sh_mem_cur_sol_size) {
25    // skupni pomnilnik za hranjenje najkrajse dolzine povezav
26    __shared__ double bestWeightCache[THREADS_PER_BLOCK];
27    // skupni pomnilnik, ki se uporabi ob zbiranju najboljsih rezultatov
28    __shared__ int bestSolIdxCache[THREADS_PER_BLOCK];
29
30    // inicializacija vrednosti
31    bestWeightCache[threadIdx.x] = DBL_MAX;
32    bestSolIdxCache[threadIdx.x] = threadIdx.x;
33
34    // tid hrani indeks niti
35    long long int tid = threadIdx.x + blockIdx.x * blockDim.x;
36
37    // T je dinamicno rezerviran skupni pomnilnik za hranjenje kombinacij
38    // prostor se rezervira ob klicu scepca
39    // BEST_T bo kazal na konec vseh kombinacij, tam bo prostor za najboljse
          resitve
40    short *BEST_T = &T[(sh_mem_cur_sol_size / sizeof(short))];
41
42    int it = 0;       // izracun stevila iteracij ki jih opravi posamezna nit
43    int threadIts = (num_combinations + blockDim.x * gridDim.x - 1) / (
          blockDim.x * gridDim.x);
44    tid *= threadIts;
45
46    // izracun indeksa, kateri del tabele naj uporablja vsaka nit
47    int mem_off = threadIdx.x * (k + 1) * 2;
48
49    if(tid < num_combinations) {
50      kSubsetLexUnrank(tid, &T[mem_off]);     // izracuna zacetno kombinacijo
```

```
51      checkSolution(mem_off, dev_crosses, dev_seg_weight, bestWeightCache,
            BEST_T);
52    }
53    it++;
54
55    while(it < (threadIts) && (tid + it) < num_combinations) {
56      kSubsetLexSuccessor(T, mem_off);        // zapisi naslednika
57      checkSolution(mem_off, dev_crosses, dev_seg_weight, bestWeightCache,
            BEST_T);
58      it++;
59    }
60
61    // sinhroniziraj niti v bloku
62    __syncthreads();
63
64    // zberi najkrajso dolzino
65    int cacheIndex = threadIdx.x;
66    int i = blockDim.x / 2;
67    while(i != 0) {
68      if(cacheIndex < i) {
69        if(bestWeightCache[cacheIndex + i] < bestWeightCache[cacheIndex]) {
70          bestWeightCache[cacheIndex] = bestWeightCache[cacheIndex + i];
71          bestSolIdxCache[cacheIndex] = bestSolIdxCache[cacheIndex + i];
72        }
73      }
74
75      __syncthreads();
76
77      i /= 2;
78    }
79
80    if(cacheIndex == 0) {                 // izvede samo prva nit v bloku
81      dev_best_weight[blockIdx.x] = bestWeightCache[0];
82      int bestSolIdx = bestSolIdxCache[0];
83      // prenesemo najboljso resitev bloka v glavni pomnilnik
84      for(i = 0; i < k; i++)
85        dev_best_solution[(blockIdx.x * k) + i] = BEST_T[bestSolIdx * k + i];
86    }
87  }
```

Listing 4.2: Ščepec, ki generira določeno število kombinacij ter zbere najboljše rešitve

# Poglavje 5

# Rezultati

V tem poglavju so predstavljeni rezultati, ki so bili izmerjeni med razvijanjem algoritmov ter časi izvajanj implementiranih algoritmov. Testiranje je potekalo na računalniku srednjega cenovnega razreda, ki vsebuje Intelov procesor Intel Core i5-3550 s 4 jedri in frekvenco 3.3 GHz ter grafično kartico NVIDIA GeForce GTX 560 Ti s sledečimi lastnostmi:

- 8 multiprocesorjev, ki delujejo s frekvenco 1.8 Ghz,
- vsak multiprocesor ima 48 izvajalnih enot (SP) ali CUDA jeder,
- velikost glavnega pomnilnika je 1024 MB,
- velikost skupnega pomnilnika za vsak blok je 49152 B,
- maksimalno število niti v bloke je 1024,
- v vsakem snopu je po 32 niti.

## 5.1 Naraščanje števila možnih kombinacij

Kot že velikokrat omenjeno, število možnih kombinacij, ki predstavljajo, ali so določene povezave v triangulaciji ali ne, narašča eksponentno z večanjem števila vhodnih točk. Metoda generiranja kombinacij povezav generira in preveri vse kombinacije. Metodi odstranjevanja in vnašanja povezav pa omejita

iskalni prostor, s čimer se izognemo preiskovanju velike večine kombinacij, ki ne predstavljajo triangulacije.

V spodnji tabeli so prikazane povprečne vrednosti števila možnih kombinacij za konfiguracije z določenim številom točk. $N_{CE}$ predstavlja število nesekajočih povezav, n število vseh povezav, ki so na voljo za izbiranje in k število povezav, ki jih moramo izbrati.

| št. točk | $\lvert N_{CE}\rvert$ | n | k | $\binom{n}{k}$ |
|---|---|---|---|---|
| 3 | 3 | 0 | 0 | 1 |
| 4 | 4 | 2 | 1 | 2 |
| 5 | 6 | 4 | 1 | 4 |
| 6 | 7 | 8 | 3 | 56 |
| 7 | 8 | 13 | 5 | 1287 |
| 8 | 9 | 19 | 7 | 50388 |
| 9 | 10 | 26 | 9 | $3 \cdot 10^6$ |
| 10 | 10 | 35 | 11 | $417 \cdot 10^6$ |
| 11 | 11 | 44 | 13 | $51 \cdot 10^9$ |
| 12 | 11 | 55 | 15 | $11 \cdot 10^{12}$ |
| 13 | 12 | 66 | 18 | $6 \cdot 10^{15}$ |
| 14 | 12 | 79 | 20 | $2 \cdot 10^{18}$ |
| 15 | 12 | 93 | 23 | $3 \cdot 10^{21}$ |
| 16 | 13 | 107 | 25 | $1 \cdot 10^{24}$ |

Tabela 5.1: Prikaz naraščanja števila kombinacij glede na število točk

## 5.2 Primerjava zaporedne ter vzporedne metode odstranjevanja robov

Naša verzija vzporedne metode odstranjevanja robov se je izkazala za neučinkovito. Glavni razlog za to je divergiranje niti v snopih. Ko niti diver-

girajo, te potrebujejo različne izvajalne enote, to pa zelo upočasni izvajanje zaradi medsebojnega čakanja.

Časi izvajanja so bili zelo odvisni od tega, koliko vej rekurzije smo pridelali na CPE. Najboljši rezultati vzporedne verzije so pokazali 7-kratno počasnejše izvajanje. Te rezultate smo dobili z izvajanjem metode na 14 točkah in 180 tisoč vej rekurzije, poslanih na GPE. Tipični časi so bili: spučanje v globino (840 ms), prenos kombinacij (140 ms) in samo izvajanje na GPE (7820 ms). Skupaj to znese 8800 ms, kar je približno 7-krat počasneje od implementacije zaporednega algoritma s časom 1250 ms.

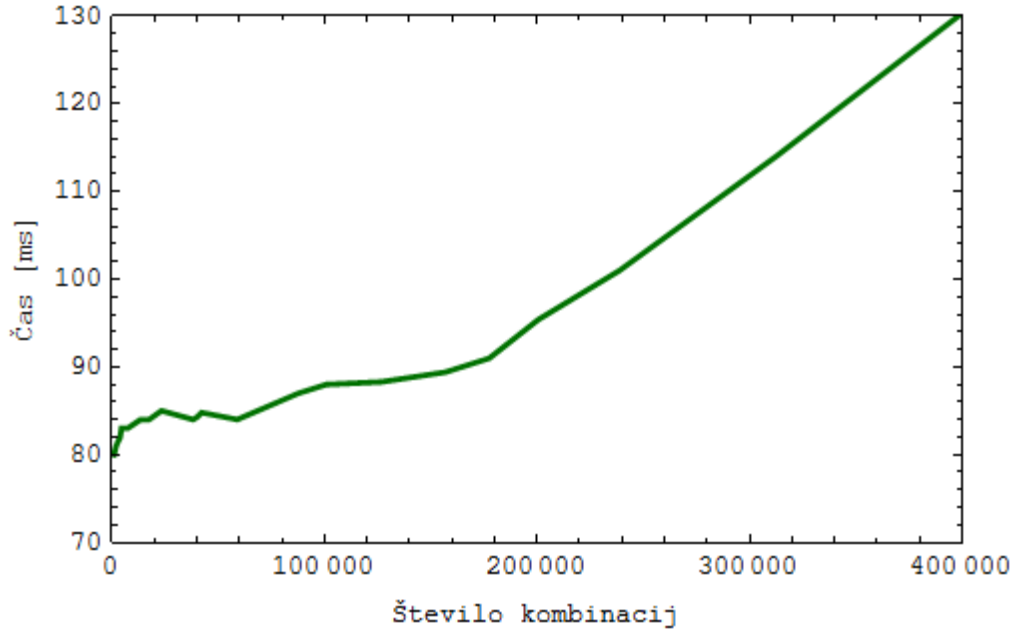
Različne variante števila niti v bloku niso prinesle bistvenih sprememb izvajalnega časa. Ugotovili smo, da algoritem deluje najhitreje z 32 nitmi v bloku.

Testirali smo tudi možno izboljšavo, opisano v sekciji 4.2.5. Rezultati pokažejo, da izvajanje ni hitrejše. Preprostejše odstranjevanje povezav ne odtehta preiskovanja večjega prostora rešitev.

Naj ponazorimo še, kako se spreminja zahtevan čas za prenos podatkov na GPE glede na število vej rekurzij. Prenos kontrolnih struktur, ki jih potrebuje GPE, da lahko nadaljuje izvajanje tam, kjer je CPE končala delo, ter rezervacija spomina za sklad se ne izvede nikoli pod 80 milisekundami. To velja tudi za primer, kjer na GPE pošiljamo podatke samo za eno vejo rekurzije. Čas porabljen za prenos podatkov je odvisen predvsem od tega, koliko različnih vej rekurzije smo pridelali do določenega nivoja z izvajanjem zaporedne verzije te metode. Iz slike 5.1 je razvidno, da ko presežemo 150 tisoč vej rekurzije, začne čas naraščati linearno s številom vej.

## 5.3 Primerjava zaporednega ter vzporednega generiranja kombinacij povezav

Pri algoritmu generiranja vseh kombinacij smo prišli do pohitritev in videli, kaj lahko zmore CUDA. Na žalost se ta metoda ne more kosati ne v zaporedni ne v vzporedni verziji z metodo odstranjevanja robov, saj preveri vse

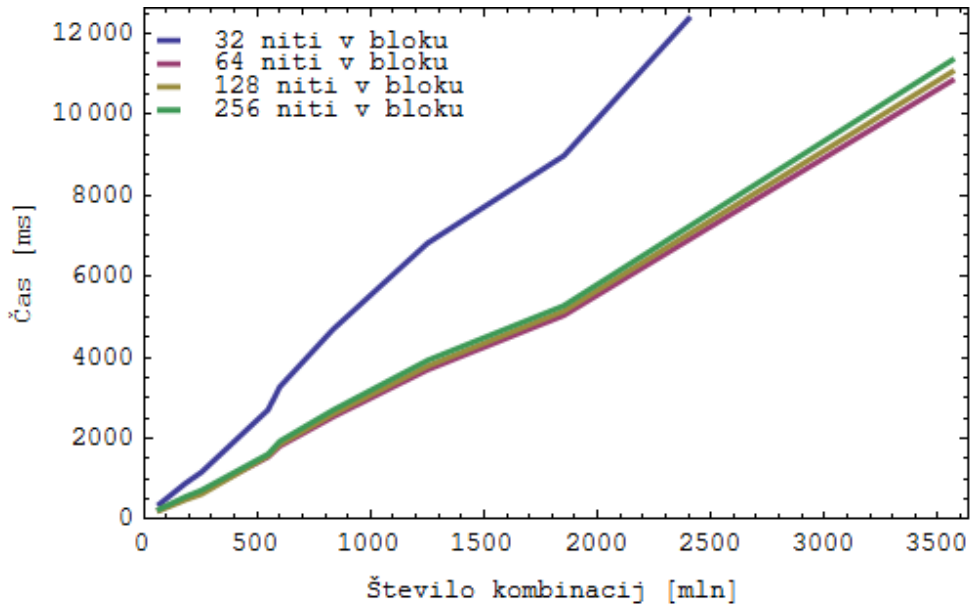



Slika 5.1: Čas prenosa podatkov ter rezerviranja prostora na GPE v odvisnosti od števila kombinacij

možnosti, teh pa je ogromno, medtem ko pri metodi odstranjevanja robov režemo stran velike dele iskalnega prostora. V nadaljevanju je predstavljenih nekaj rezultatov, ki so nam pomagali do razvoja optimalne različice algoritma. Na koncu je narejena še primerjava med zaporedno ter vzporedno različico.

Med razvijanjem vzporednega algoritma smo se spraševali, kakšno naj bo število niti v bloku ter blokov v mreži. Sodeč po viru [6] naj bi bilo število niti na blok večkratnik števila niti v snopu. Če tega ne upoštevamo, zapravljamo računske zmogljivosti GPE, saj nimamo polnih snopov. Testirali smo naslednje opcije števila niti v bloku: 32, 64, 128 ter 256. Najboljše rezultate smo dobili s 64 nitmi v bloku, kar lahko vidimo tudi na sliki 5.2.

Ostane nam še izbira blokov v mreži. Izkaže se, da je najbolje imeti teh čimveč. Tako imamo pri primerih z največ kombinacijami (več kot $500 \times 10^6$) 65535 blokov, toliko jih namreč podpira GPE. Seveda to pomeni, da je pri

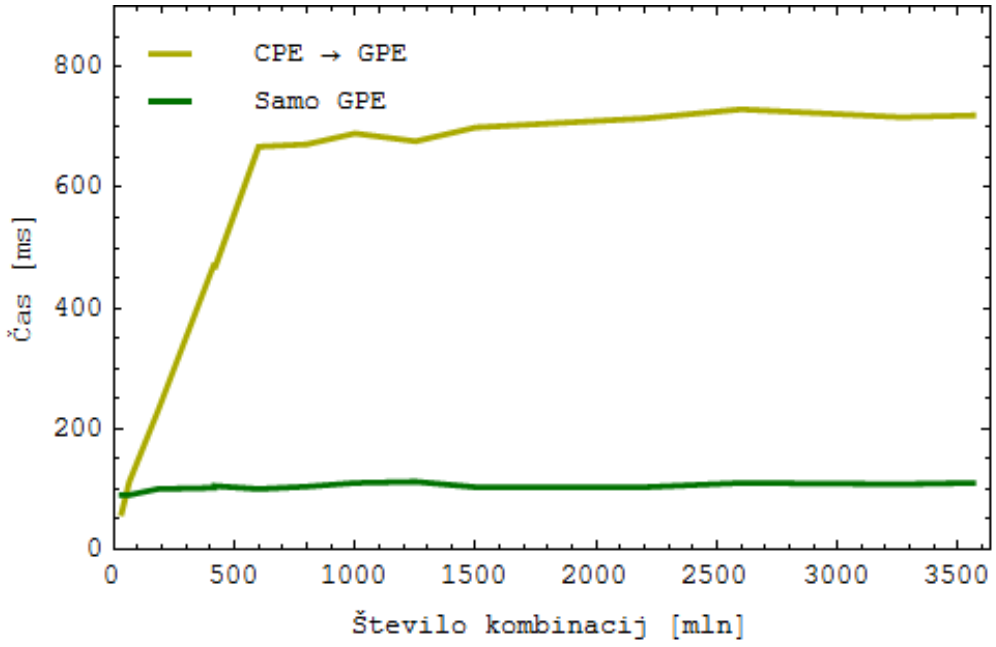



Slika 5.2: Časi izvajanj vzporednega generiranja kombinacij povezav za različna števila niti v bloku

manjših primerih pognanih občutno preveč niti, kot je to potrebno. Po analizi sodeč, je tam najbolje vsaki niti dodeliti vsaj 200 iteracij, temu ustrezno potem prilagodimo število blokov.

Naslednja stvar, kjer smo bili v dilemi glede hitrosti izvajanja, je generiranje začetnih kombinacij za vsako nit. Prva opcija je, da se začetne kombinacije generirajo za vsako nit na CPE ter potem prenesejo na GPE, pri drugi pa vsaka nit poskrbi za svojo začetno kombinacijo. Na sliki 5.3 lahko vidimo primerjavo med časi izvajanja obeh verzij.

Pri prvi verziji čas na začetku narašča, pri meji 700 milijonov vseh kombinacij pa je konstanten, saj smo dosegli maksimalno število niti, ki so lahko zagnane na GPE ter s tem maksimalno število začetnih kombinacij. V tem primeru trajata generiranje ter prenos skupaj 700 ms, kar je zelo počasi v primerjavi z drugo verzijo. Tam vsaka nit zgenerira svojo začetno kombinacijo, s tem se podaljša izvajanje ščepca za približno 25 ms pri vsakem številu kombinacij, čas za prenos potrebnih podatkov na GPE pa je okoli 80 ms,



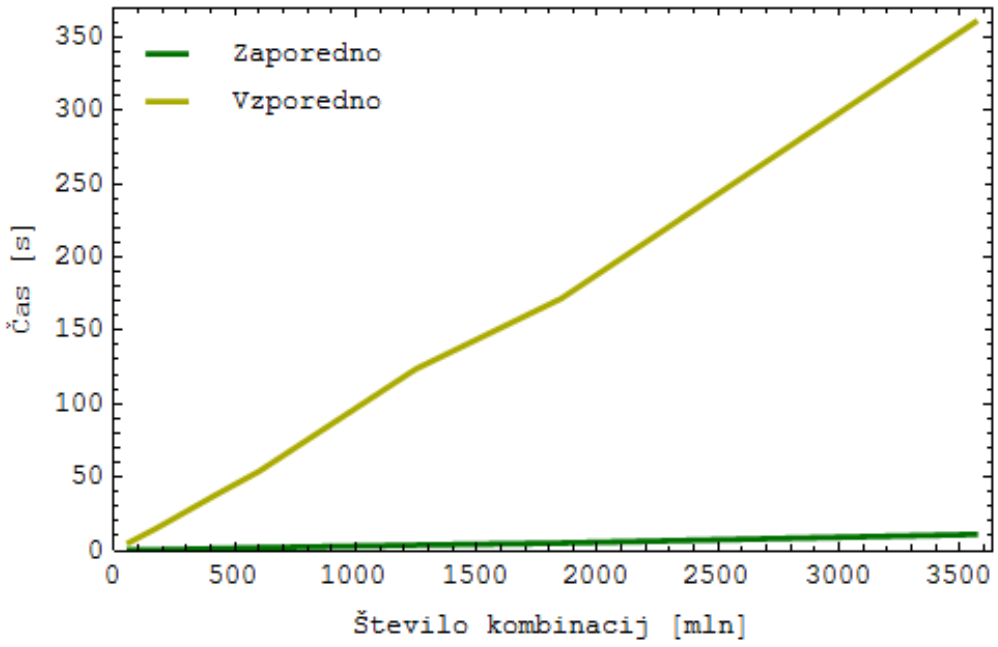

Slika 5.3: Primerjava med generiranjem začetnih kombinacij na CPE in njihov prenos ter drugo opcijo, kjer se generirajo na GPE

skupaj je to 105 ms, kar pomeni, da je ta rešitev boljša.

Zanimalo nas je tudi, kakšna je razlika, če uporabimo redukcijo pri zbiranju najboljšega rezultata v bloku. Rezultati so pokazali, da je razlika minimalna, tj. 5 ms. Razlog temu je majhno število niti v bloku, imamo jih namreč 64.

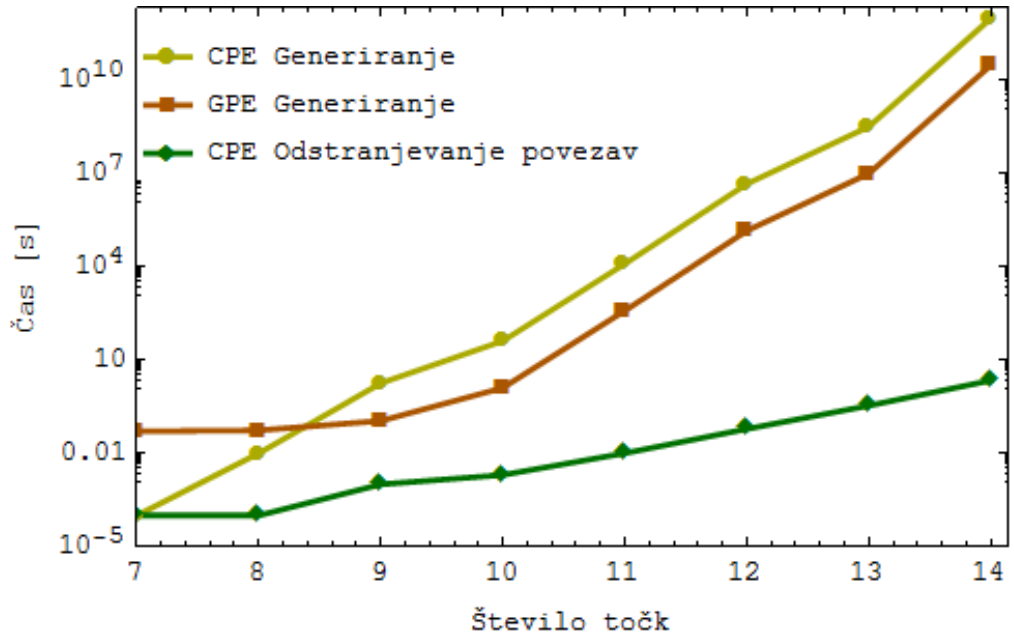
Poglejmo si, kakšne so bile dejanske pohitritve v nasprotju z zaporednim algoritmom. Na sliki 5.4 je prikazana primerjava med časom izvajanja na zaporednem algoritmu ter med vzporednim. Vidimo, da čas pri obeh narašča linearno z večanjem števila kombinacij. Rezultati pokažejo, da je vzporedni algoritem 32-krat hitrejši kot zaporedni. Torej za 3500 mln. kombinacij rabi vzporedni algoritem 11 sekund, zaporedni pa kar 352 sekund.

Na koncu smo naredili še primerjavo med metodo generiranja kombinacij (zaporedno, vzporedno) ter metodo odstranjevanja povezav. Rezultati so povprečje časov, ki so bili izmerjeni na istih množicah točk. Kot lahko




Slika 5.4: Primerjava med časom izvajanja zaporedne ter vzporedne verzije generiranja kombinacij povezav

vidimo, nam metoda odstranjevanja povezav omogoča reševanje dosti večjih problemov kot vzporedno generiranje kombinacij.




Slika 5.5: Primerjava med časi izvajanja metode z odstranjevanjem povezav in generiranjem kombinacij povezav

# Poglavje 6

# Sklep

V delu smo se seznanili s problemom iskanja triangulacije z najmanjšo skupno dolžino povezav v realni ravnini. Dodobra smo spoznali arhitekturo CUDA, njene omejitve in možnosti, ki jih ponuja za razvoj vzporednih algoritmov. Izmed mnogo metod iskanja optimalne triangulacije smo si izbrali tri, jih preučili in dve od njih tudi implementirali. Pri obeh smo se najprej lotili implementacije zaporedne verzije, nato je sledila še vzporedna za izvajanje na GPE.

Implementirani metodi sta metoda odstranjevanja povezav in metoda generiranja kombinacij povezav. Ugotovili smo, da metoda odstranjevanja povezav ni primerna za izvajanje na arhitekturi CUDA. Velik problem so predstavljale vejitve v programu. Naša vzporedna implementacija te metode je bila 7-krat počasnejša od zaporedne verzije. Dosti bolje se je odrezal vzporedni algoritem metode generiranja kombinacij povezav. Uspelo nam je doseči 32-kratno pohitritev napram zaporednem algoritmu. Čeprav nam je uspelo doseči takšno pohitritev, je ta vzporedni algoritem vseeno počasnejši od zaporednega algoritma metode odstranjevanja povezav. Razlog je v tem, da pri generiranju kombinacij povezav preiščemo občutno več kombinacij, ki se jim pa metoda odstranjevanja povezav izogne, saj je tipa "razveji in omeji". Tretje metode, metode vnašanja povezav, se nismo odločili implementirati zaradi prevelike podobnosti z metodo odstranjevanja povezav. Pri vzporedni

implementaciji bi naleteli na enake težave zaradi vejitev.

Vsekakor je CUDA primerna za razvoj vzporednih algoritmov. Morajo pa le-ti imeti visoko stopnjo podatkovne vzporednosti in malo krmiljenja tokov. Za maksimalni izkoristek je potrebno imeti tudi dobro poznavanje okolja CUDA.

Vprašanje, ali je izračun MWT primeren z uporabo GPE, ostaja. Nadaljnje delo bi se lahko nadaljevalo v smeri iskanja in preučitev že zapisanih pristopov izračuna MWT [3]. Večina metod je matematično bolj zahtevnih od teh, ki so obravnavane v tem delu. Pri implementaciji vzporednih algoritmov je mogoč tudi hibriden pristop, kjer si delo izmenjujeta CPE in GPE. Tam GPE prevzame podatkovno intenzivne naloge, CPE pa se ubada s pripravo dela za GPE [2].

# Literatura


[1] J. A. De Loera , J. Rambau , F. Santos, "Triangulations: Structures for Algorithms and Applications". Springer Publishing Company, 2010, strani 91-117.

[2] A. Boukedjar, M. E. Lalami, D. El-Baz, "Parallel Branch and Bound on a CPU-GPU System".

[3] M. Grantson, C. Borgelt, C. Levcopoulos, "Fixed Parameter Algorithms for the Minimum Weight Triangulation Problem".

[4] T. Hlavaty, V. Skala, "Combinatories and Triangulations".

[5] D. L. Kreher, D. R. Stinson, "Combinatorial algorithms: Generation, Enumeration and Search". CRC Press, 1999, strani 43-45.

[6] NVIDIA Corporation, "NVIDIA CUDA C Programming Guide Version 4.2", ZDA, 2010.

[7] J. Sanders, E. Kandrot, "CUDA by Example: An introduction to General-Purpose GPU programming", Addison-Wesley, ZDA 2010.

[8] "http://en.wikibooks.org/wiki/Algorithm_Implementation/Geometry/Convex_hull/Monotone_chain".

[9] "http://en.wikipedia.org/wiki/Delaunay_triangulation".

[10] "http://www.microsoft.com/Products/Games/FSInsider/developers/Pages/GlobalTerrain.aspx".