FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Škálovatelnost algoritmu nejbližších sousedů
Student:	Antonín Dvořák
Vedoucí:	doc. Ing. Pavel Kordík, Ph.D.
Studijní program:	Informatika
Studijní obor:	Znalostní inženýrství
Katedra:	Katedra aplikované matematiky
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Proveďte rešerši algoritmů a jejich implementací založených na hledání nejbližších sousedů (KNN, K Nearest Neighbors). Vyberte několik variant a zhodnoťte škálovatelnost vzhledem k dimenzionalitě dat. Experimentujte s výpočty na CPU, GPU, případně TPU pro minimálně 3 datasety.
Pro jednotlivé varianty algoritmu KNN diskutujte poměr zhoršení přesnosti a zrychlení výpočtu zejména z hlediska použití CPU vs GPU.

Seznam odborné literatury

https://github.com/vincentfpgarcia/kNN-CUDA
https://github.com/chrischoy/knn_cuda
https://github.com/src-d/kmcuda

Ing. Karel Klouda, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 29. ledna 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Škálovatelnost algoritmu nejbližších sousedů

Antonín Dvořák

Katedra aplikované matematiky
Vedoucí práce: doc. Ing. Pavel Kordík, Ph.D.

16. května 2019

Poděkování

Chtěl bych poděkovat svému vedoucímu doc. Ing. Pavlu Kordíkovi, Ph.D za jeho ochotu a cenné rady, které mi poskytl při psaní této práce. Dále bych rád poděkoval své rodině za podporu nejen při studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 16. května 2019 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Dvořák, Antonín. *Škálovatelnost algoritmu nejbližších sousedů*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato práce se zabývá efektivními algoritmy pro hledání k nejbližších sousedů. Představili jsme tři konkrétní přístupy. Prvním z nich je použití k-means shlukování pro *kNN*, druhým tvorba aproximativního *kNN* grafu za použití *lokálně senzitivní hashov* a posledním *kNN* na grafické kartě. Implementace jsou z podstatné části psány v jazyce C++. V práci je nakonec provedeno měření skutečné efektivity jednotlivých algoritmů.

Klíčová slova kNN, škálovatelnost, paralelní implementace, efektivní hledání sousedů, graf sousedů, strojové učení, CUDA, cuBLAS, Open MPI, kMkNN, LSH

Abstract

This thesis deals with efficient algorithms used for k nearest neighbors. We introduced three different approaches. First of them is k-means clustering for *kNN*, second one is an approximate *kNN* graph construction using *locality sensitive hashing* and last one *kNN* on graphical processing unit. Most parts of implementations are writen in C++. In the end we measure real efficiency of given algorithms.

Keywords kNN, scalable, parallel implementation, efficient neighbor search, neighbour graph, machine learning, CUDA, cuBLAS, OPEN MPI, kMkNN, LSH

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Algoritmus pro hledání nejbližších sousedů (*kNN*) je v dnešní době hojně využíván v řadě odvětví, jako je například strojové učení, rozpoznávání vzorů a data mining.

Klasický *kNN* algoritmus funguje rychle pro menší datasety, ale pro robustní datasety je hledání velmi časově náročné.

Téma jsem si zvolil z toho důvodu, že tento algoritmus je jedním ze základních prvků oboru Znalostního inženýrství. Zajímá mě, jak aplikovat algoritmy na dnešní problémy, kdy je potřeba zpracovávat více a více dat.

Práce je určena pro všechny, kteří mají zájem o výše zmiňovaná odvětví. Mohou pro ně být zajímavé různé způsoby toho, jak jsou algoritmy přizpůsobené tomu, aby dobře fungovaly pro robustní datasety. Ovšem většina uživatelů internetu se s ním pravděpodobně setkává každý den. Mnoho doporučovacích systémů je založeno právě na tomto algoritmu. Může se jednat například o to, jaké jsou nám doporučovány seznamy písniček a videí, filmy, produkty a další.

V této práci si představíme algoritmy, které vycházejí z *kNN*, ale jsou nějakým způsobem upravené tak, aby byly z časového hlediska použitelné i pro zmiňované robustní datasety.

Cílem je teoreticky představit jednotlivé přístupy, ukázat jak fungují, nalézt a upravit pro své potřeby již existující implementace těchto algoritmů pro CPU i GPU. Na závěr budou porovnány jejich rychlosti a přesnosti na různých datasetech. U jednotlivých algoritmů budeme hýbat s jejich parametry a sledovat, jak se mění rychlost vůči přesnosti vyhodnocování.

Na začátku si představíme a zavedeme základní pojmy, které nás celou prací budou provázet. Poté si uděláme teoretický přehled různých algoritmů založených na rychlém hledání nejbližších sousedů. Dále si ukážeme konkrétní implementace daných algoritmů a prostředí, na kterém budeme vykonávat testování. Nakonec zhodnotíme rychlosti a přesnosti vyhodnocování pro jednotlivé algoritmy při konkrétních implementacích.

KAPITOLA 1

Základní pojmy

1.1 Strojové učení

V této sekci je čerpáno primárně z [1].

Strojové učení se zabývá extrakcí znalostí z dat. Je to věda založená na statistice, umělé inteligenci a computer science. Zabývá se vývojem systémů, které se dokáží učit z dat a přizpůsobovat se změnám, bez zásahu člověka.

V posledních letech se s ním setkáváme každý den, ať už vědomě, tak i nevědomky. Například doporučení filmů, na které se podíváme, jaké jídlo si objednáme, co za produkty zakoupíme. Dále doporučené playlisty, označování a rozpoznávání lidí na fotkách. Když se podíváme na stránky jako jsou například Facebook, Amazon, Netflix, Youtube a další, víme, že každá z nich využívá nejeden model strojového učení.

Strojové učení má využití i mimo komerční sféru, například zkoumání hvězd, objevování nových planet a nových částic, analýza DNA, diagnóza a léčba rakoviny.

V dřívějších dobách „inteligentních“ aplikací se systémy psaly za pomoci mnoha „if“ rozhodnutí pro zpracování vstupních dat. Vezměme například emailový spam filtr - můžeme rozlišovat klíčová slova, podle kterých budeme maily označovat jako spam. Toto je příklad použití expertního systému. V některých případech je tento přístup dostačující, ale má svá rizika a nedostatky. Logika za těmito pravidly, podle kterých se rozhodujeme je specifická pouze pro jeden úkol. I minimální změna účelu může znamenat přepsání celého systému. Navíc tvorba pravidel vyžaduje znalosti experta daného oboru.

Tento způsob například narazí v případě rozpoznávání obličejů. V dnešní době toto dokáže každý běžný telefon. Nejsme schopni pomocí jednoduchých podmínek napsat systém, který by z obrázku dokázal odhalit obličej.

Až v roce 2001 se povedlo tento problém vyřešit právě za pomoci strojového učení. Máme systém, který má k dispozici dostatečně velkou množinu obličejů, na základě kterých dokáže v dalších obrázcích rozpoznat charakteristické rysy obličejů.

1.1.1 Metody strojového učení

V této části zmíníme tři metody strojového učení, které jsou pro naše účely dostačující. Různé publikace [2, 3] se zmiňují i o další metodě *Reinforcement machine learning*, tou se v této práci zabývat nebudeme.

1.1.1.1 Supervizované učení (učení s učitelem)

Většina strojového učení je právě *supervizovaná*. Poskytujeme systému nejen vstupní data, ale i výstupy, které očekáváme. Analogie: „*Dáme dítěti různě velké barevné kostky a ukážeme mu, jak je třídit podle barvy a velikosti.*“ [2]

Supervizované učení je takové, kde se snažíme zjistit, jak *vysvětlovanou proměnnou* Y ovlivňují *příznaky* $X_0, X_1, \ldots, X_{p-1}$. Hledáme tedy nějaký funkční vztah, který tuto závislost co nejblíže popisuje.

$$Y \approx f(X_0, X_1, \ldots, X_{p-1})$$

Tvar hledané funkce často ovlivňuje to, jakých hodnot může nabývat *vysvětlovaná proměnná* Y.

- Pokud Y nabývá pouze pár málo hodnot (ano/ne, identifikace ručně psaného písma, ...), mluvíme o **klasifikaci**.
- Pokud Y nabývá tolika hodnot, že je rozumnější k ní přistupovat jako k spojité (cena předmětu, výška osoby, ...), mluvíme o **regresi**.

Popis *supervizovaného učení* byl převzat z [4] .

Mezi některé populární modely patří například:

- *lineární regrese* pro regresní problémy,
- *náhodné lesy* pro klasifikační i regresní problémy,
- *metoda nejbližších sousedů* pro klasifikační i regresní problémy [5].

1.1.1.2 Nesupervizované učení (učení bez učitele)

V tomto případě poskytujeme systému pouze vstupní data, ale žádné očekávané výstupy. Nemáme tedy žádnou *vysvětlovanou proměnnou* Y, kterou bychom u trénovacích dat znali a snažili se ji naučit predikovat.

Cílem *nesupervizovaného učení* je porozumět struktuře dat pouze na základě dat samotných. Proto jej nazýváme také *učení bez učitele*. Analogie: „*Dáme dítěti různě*

velké barevné kostky a čekáme, dokud nenajde způsob, jak si v nich udělat pořádek.“ [2]

„*Porozuměním zde typicky myslíme nalezení co ”nejmenších“ oblastí v prostoru příznaků, kde se data vyskytují nejčastěji. Obvykle totiž platí, že se naměřená skutečná data nevyskytují v celém prostoru stejně pravděpodobně, ale bývají lokalizována - tvoří nějaké shluky, vyskytují se v méně-dimenzionálních oblastech.*“ [6]

Nesupervizované učení lze ještě rozdělit do dvou částí [5]:

- **clusterování** - shlukování dat na základě podobnosti; například zákazníci, kteří nakupují podobné produkty,
- **asociace** - identifikace pravidel v datech; například zákazníci, kteří zakoupí produkt X, zakoupí i produkt Y.

1.1.1.3 Semi-supervizované učení

Poskytujeme systému data označená *vysvětlovanou proměnnou* Y, ale i data, která označená nemáme, těch obvykle bývá většina. Toto učení se nachází mezi *supervizovaným* a *nesupervizovaným*.

Spousta systémů se nachází právě zde. Příkladem může být analýza řeči, klasifikace obsahu webů, klasifikace proteinové sekvence a další [7].

Je to způsobené tím, že získávání neoznačených dat je jednoduché, levné. Naopak zisk označených nemusí být tak jednoduchý, ani levný. Často vyžaduje zkušenosti experta dané problematiky.

Můžeme zde použít *nesupervizovaného učení* a snažit se porozumět struktuře dat, nebo můžeme použít *supervizovaného učení* a postupně predikovat neoznačené záznamy.

1.2 Hledání nejbližších sousedů (kNN, k Nearest Neighbors)

Základní informace o algoritmu *kNN* byly získány z [8].

kNN je důležitý algoritmus pro strojové učení, lze jej použít pro *supervizované* i *nesupervizované učení*. V případě *supervizovaného učení* jej lze použít pro *klasifikaci* i *regresi*.

Tento algoritmus má mnoho použití v data miningu (např. rozpoznávání vzorů), bioinformatice (např. diagnostika rakoviny prsu, předpověď počasí) a doporučovacích systémech.

Celočíselná hodnota $k > 0$ určuje, kolik nejbližších bodů z referenční množiny algoritmus *kNN* hledá k námi dotazovanému bodu q (query).

1.2.1 Popis metody kNN

kNN lze popsat pomocí následujících kroků:

1. zvolme hodnotu k (volba záleží na konkrétních datech, neexistuje obecná metoda, jak tuto hodnotu zvolit),
2. spočtěme vzdálenost1.3 nového bodu q a všech referenčních bodů,
3. seřaďme spočítané vzdálenosti a vyberme k nejbližších bodů,
4. podíváme se na jejich hodnoty *vysvětlované proměnné* Y,
   a) v případě *klasifikace*, vezměme nějčastější hodnotu Y a prohlašme ji jako predikovanou hodnotu Y našeho bodu,
   b) v případě *regrese*, vezměme průměrnou hodnotu Y a prohlašme ji jako predikovanou hodnotu Y našeho bodu.

1.2.2 Časová složitost

V krátkosti shrneme časovou složitost tohoto algoritmu.

Buď n počet referenčních bodů, m počet query bodů a d dimenze dat.

1. Složitost výpočtu vzdáleností mezi jedním query bodem q a n referenčními body je rovna $\mathcal{O}(nd)$. Pro m bodů je tedy složitost $\mathcal{O}(mnd)$.
2. Složitost řazení pole vzdáleností, které obsahuje n vzdáleností, kde řadíme prvních k hodnot je rovna $\mathcal{O}(k \log n)$. Seřazení vzdáleností všech referenčních bodů má složitost $\mathcal{O}(m \cdot k \log n)$.

Výsledná složitost je tedy $\mathcal{O}(mnd)$.

1.3 Metrika (vzdálenost)

Informace o konkrétních metrikách byly čerpány z [6, 9].

Metrika, nebo také vzdálenost, na množině $\mathcal{X}$ je funkce $d\colon \mathcal{X} \times \mathcal{X} \to [0, \infty)$ taková, že pro každě $x, y, z \in \mathcal{X}$ platí [10]:

1. $d(x, y) \geq 0, d(x, y) = 0 \iff x = y$ (pozitivní definitnost)
2. $d(x, y) = d(y, x)$ (symetrie)
3. $d(x, y) \leq d(x, z) + d(z, y)$ (trojúhelníková nerovnost)

1.3.1 Eukleidovská vzdálenost

Také nazývána jako L_2 vzdálenost. Jedná se o nejčastěji používanou metriku.

Mějme dva body $x = (x_0, x_1, \dots, x_{p-1})$ a $y = (y_0, y_1, \dots, y_{p-1})$ z $\mathbb{R}^p$. Vzdálenost těchto bodů vyjádříme jako:

$$||x - y||_2 = d_2(x, y) = \sqrt{\sum_{i=0}^{p-1} (x_i - y_i)^2}$$

1.3.2 Manhattanská vzdálenost

Také nazývána jako L_1 vzdálenost.

Mějme dva body $x = (x_0, x_1, \dots, x_{p-1})$ a $y = (y_0, y_1, \dots, y_{p-1})$ z $\mathbb{R}^p$. Vzdálenost těchto bodů vyjádříme jako:

$$||x - y||_1 = d_1(x, y) = \sum_{i=0}^{p-1} |x_i - y_i|$$

1.3.3 Minkowského vzdálenost

Také nazývána jako L_k vzdálenost.

Mějme dva body $x = (x_0, x_1, \dots, x_{p-1})$ a $y = (y_0, y_1, \dots, y_{p-1})$ z $\mathbb{R}^p$. Vzdálenost těchto bodů vyjádříme jako:

$$||x - y||_k = d_k(x, y) = \sqrt[k]{\sum_{i=0}^{p-1} |x_i - y_i|^k}$$

Eukleidovská vzdálenost odpovídá volbě $k = 2$, Manhattanská vzdálenost pak $k = 1$.

1.3.4 Čebyševova vzdálenost

Také nazývána jako L_∞ vzdálenost.

Mějme dva body $x = (x_0, x_1, \dots, x_{p-1})$ a $y = (y_0, y_1, \dots, y_{p-1})$ z $\mathbb{R}^p$. Vzdálenost těchto bodů vyjádříme jako:

$$||x - y||_\infty = d_\infty(x, y) = \max_i |x_i - y_i|$$

1.4 Dimenzionalita

Dimenzionalita je hodnota, která říká, kolik *příznaků* dataset má. Nicméně ve skutečnosti si mohou některé atributy být podobné, na sobě závislé, v nějakém smyslu duplikované, případně některé nepoužitelné, nebo i zbytečné. Je tedy obtížné určit, jaká je skutečná dimenze našich dat.

1.4.1 Prokletí dimenzionality (Curse of Dimensionality)

Prokletí dimenzionality je fenomén, který popisuje některé problémy objevující se v případě vysokého počtu příznaků (při vysoké *dimenzionalitě)*.

S *kNN* jsou spojeny zejména dva efekty způsobené vysokou dimenzí dat [10]:

- Se zvyšující dimenzí se data vzdalují a řídnou. Pro zachování stejné hustoty pro vyšší dimenzi dat je nutné exponenciálně zvyšovat počet vzorků testovacích dat, což je obvykle velmi těžké, nebo nereálné.
- S rostoucí dimenzí se pro klasické metriky (*Eukleidovská vzdálenost, Manhattanská vzdálenost* atd.) zmenšují rozdíly mezi vzdálenými a blízkými body.

Příklad řídnutí bodů [10, 11, 12]

Mějme d-dimenzionální jednotkovou hyperkrychli, tedy oblast

$$[0,1] \times [0,1] \times \ldots \times [0,1] \subset \mathbb{R}^d$$

a v ní máme dle uniformního rozdělení 1000 bodů.

Otázka: Jak velkou potřebujeme mít „podkrychli“, aby obsahovala v průměru 10 bodů (tzn. 1/100 objemu) ?

- Pro jednorozměrnou krychli (úsečku), je to úsečka o délce 0,01.
- Pro dvourozměrnou krychli (čtverec), je to čtverec o straně 0,1.
- Pro trojrozměrnou krychli, je to krychle o hraně a, kde $a = 1/100$, tedy

$$a = \sqrt[3]{\frac{1}{100}} \approx 0,215.$$

- Pro dimenzi d, je to krychle o hraně a

$$a = \sqrt[d]{\frac{1}{100}}.$$

Pro d = 10 je $a \approx 0{,}63$ a pro d = 50 je $a \approx 0{,}91$.

Tyto problémy hrají roli v mnoha metodách a proto je *redukce dimenzionality* jedním ze zásadních témat při zpracování dat.

1.4.2 Manifold hypothesis

Manifold hypothesis říká, že reálná mnohorozměrná data se vyskytují v podprostoru menší dimenze původního mnohodimenzionálního prostoru [13].

1.4.3 Redukce dimenzionality

Redukce dimenzionality je proces, kdy redukujeme dimenzi prostoru příznaků. Je důležitým krokem pro extrakci informací z mnohorozměrných dat, v případě vysoce dimenzionálních dat může být zcela nezbytný [14].

Důvody jsou:

- pomáhá kompresi dat a tím šetření místa,
- zrychlení výpočtů,
- zbavení se závislých a nepodstatných/nadbytečných příznaků,
- snížení dimenzionality dat na 2D nebo 3D nám dovoluje dobrou vizualizaci dat,
- vyhnutí se *Prokletí dimenzionality* [15].

Metody pro redukci dimenze se často dělí do dvou skupin:

- výběr příznaků (feature selection) - výběr jen těch příznaků, které jsou relevantní,
- extrakce příznaků (feature extraction) - nahrazování existujících příznaků novými, ale v prostoru menší dimenze [16].

Základním rozdílem tedy je, že *selekce proměnných* ponechává podmnožinu původních příznaků, zatímco *extrakce příznaků* vytváří nové.

Metodami jednotlivých skupin se v této práci zabývat nebudeme.

KAPITOLA 2

Rešerše algoritmů

2.1 k-Means pro k-Nearest Neighbors (kMkNN)

Tato sekce čerpá z původní práce [17].

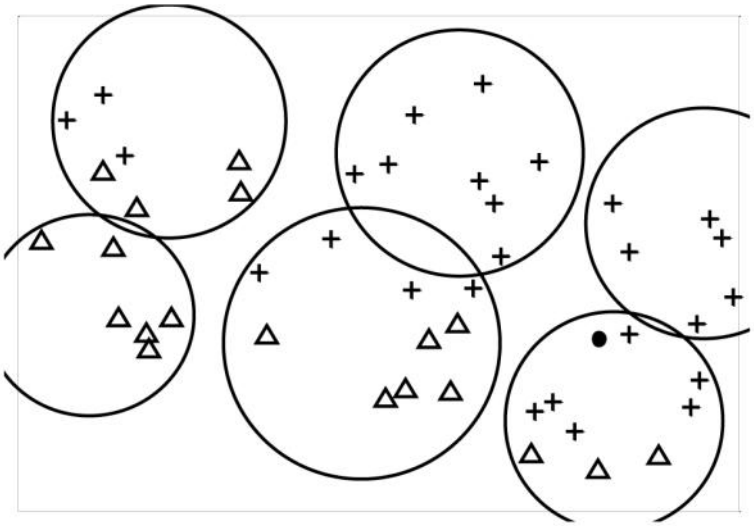
Algoritmus *kMkNN* je založený na *k*-means shlukování (clusterování) a *trojúhelníkové nerovnosti*, které využívá pro hledání k nejbližších sousedů. Je efektivní pro hledání v prostoru o vysoké dimenzi.

2.1.1 Základní myšlenka

Základní myšlenkou je rozdělení referenčních bodů do jednotlivých shluků a poté pro hledání sousedů daného bodu q (*query*) využijeme *trojúhelníkové nerovnosti*, abychom se vyhnuli počítání vzdáleností bodů, které jsou v shlucích příliš vzdálených od bodu q.

Toto je vidět na Obrázku 2.1, kde algoritmus nejdříve rozdělí body do shluků a následně hledá sousedy, počínaje body v shluku v pravém dolním rohu.

Obrázek 2.1: Ilustrace *kMkNN* [17].



2.1.2 Fáze

Předpokládejme, že máme n referenčních bodů p_i pro $(1 \leq i \leq n)$, každý s dimenzí m.

2.1.2.1 Přípravná fáze

V přípravné fázi rozdělíme referenční body do shluků, pro každý bod si uložíme vzdálenost od středu shluku, v kterém se nachází.

Algoritmus 1 Pseudokód přípravné fáze *kMkNN*

Vstup: n referenčních bodů p_i pro $(1 \leq i \leq n)$, každý s dimenzí m.
Výstup: kc shluků se středy c_j pro $(1 \leq j \leq kc)$, které mají přiřazeny body p_i.

1: $kc \leftarrow s\sqrt{n}$, kde $s \geq 0$
2: Rozděl n bodů do kc shluků za použití k-means algoritmu
3: Spočítej a ulož vzdálenosti d_{ij} každého bodu p_i od středu jeho shluku c_j
4: Pro každý shluk j seřaď vzdálenosti d_{ij} sestupně

2.1.2.2 Hledací fáze

V hledací fázi začneme shlukem, který je nejblíže bodu q. Postupně procházíme i vzdálenější. V každém shluku procházíme body p_i, počínaje nejvzdálenějším od aktuálního středu shluku c_j. Díky *trojúhelníkové nerovnosti* vyřadíme mnoho bodů, u kterých bychom jinak museli počítat vzdálenost od bodu q, viz Pseudokód 5 . Pokud $d_{max} \leq ||q - c_j||$ - $||p_i - c_j||$, můžeme se přesunout k dalšímu shluku, protože platí $||p_i - c_j|| > ||p_{ix} - c_j||$ pro všechny zbývající body p_{ix} v shluku j, neboli na základě Předpokladů 2.1.3 už v shluku nenajdeme bližší bod.

Algoritmus 2 Pseudokód hledací fáze *kMkNN*

Vstup: kc shluků se středy c_j pro $(1 \leq j \leq kc)$ s body p_i;
Výstup: k nejbližších sousedů bodu q

1: Alokuj strukturu pro k bodů, nastav vzdálenost každému z nich a d_{max} na nejvyšší možnou
2: Spočítej vzdálenosti $||q - c_j||$ pro všechny středy shluků c_j z $(1 \leq j \leq kc)$ a seřaď je vzestupně
3: **for** each shluk $j \in (1 \leq j \leq kc)$, počínaje nejbližším shlukem k bodu q **do**
4: **for** each bod $p_i \in j$, počínaje nejvzdálenějším bodem od středu c_j **do**
5: **if** $d_{max} \leq ||q - c_j||$ - $||p_i - c_j||$ **then**
6: *break*
7: **else**
8: $d_{tmp} \leftarrow ||q - p_i||$
9: **if** $d_{max} > d_{tmp}$ **then**
10: Odstraň ze struktury sousedů bod s vzdáleností d_{max}
11: Přidej do struktury sousedů bod p_i,
12: Přepočítej d_{max}
13: **end if**
14: **end if**
15: **end for**
16: **end for**

2.1.3 Předpoklady

První předpoklad: většinu k nejbližších sousedů nalezneme v „pár prvních“ shlucích. To právě díky tomu, že jsme si shluky seřadili podle vzdálenosti mezi bodem q a středem shluku c_j od nejbližšího.

Druhý předpoklad: pro každý bod p_i v shluku j (kromě toho nejbližšího bodu q), je pravděpodobné, že vzdálenost $||p_i - c_j|| \leq ||q - c_j||$. Toho si můžeme například všimnout na Obrázku 2.1.

Na základě tohoto předpokladu můžeme použít *trojúhelníkovou nerovnost*, abychom snížili počet nepotřebných výpočtů vzdáleností. Upravíme

$$||q - c_j|| \leq ||q - p_i|| + ||p_i - c_j||$$
$$||q - c_j|| - ||p_i - c_j|| \leq ||q - p_i||$$

Pokud $d_{max} \leq ||q - c_j|| - ||p_i - c_j||$, poté $d_{max} \leq ||q - p_i||$ a nemusíme vzdálenost $||q - p_i||$ počítat.

2.1.4 Časová složitost

V této části se podíváme, jak na tom algoritmus *kMkNN* je z pohledu asymptotické složitosti.

2.1.4.1 Přípravná fáze

U přípravné fáze záleží, jaký algoritmus se pro k-means zvolí. V původní práci byl použit *Lloydův algoritmus (Lloyd's algortithm)*, což je jednoduchý heuristický algoritmus, který iterativně konverguje k lokálnímu minimu. Asymptotická složitost tohoto algoritmu se dá vyjádřit jako $\mathcal{O}(nmki)$, kde n je počet m-dimenzionálních referenčních bodů, k je počet shluků, i je počet iterací.

2.1.4.2 Hledací fáze

Mějme počet shluků $kc = \mathcal{O}(\sqrt{n})$, každý s $\mathcal{O}(\sqrt{n})$ body a navíc $k < \sqrt{n}$, následně v této fázi *kMkNN* počítá vzdálenosti mezi bodem q a referenčními body p_i pouze v pár nejbližších shlucích a nachází k nejbližších sousedů.

Výpočet vzdálenosti mezi bodem q a všemi středy shluků je $\mathcal{O}(m\sqrt{n})$, seřazení těchto vzdáleností od nejbližšího zabere $\mathcal{O}(\sqrt{n}\log n)$, kontrola *trojúhelníkové nerovnosti* pro (v nejhorším případě) všechny referenční body má složitost $\mathcal{O}(n)$, výpočet vzdálenosti mezi q a referenčními body v nejbližších shlucích zabere $\mathcal{O}(m\sqrt{n})$.

2.1.5 Poznámky

Pro efektivní získávání vzdálenosti d_{max} ze struktury pro k nejbližších sousedů je vhodné použít *maximovou binární haldu (max-heap)*, kde kořenem je bod s nejvyšší vzdáleností (d_{max}). Pokaždé, když najdeme bod p_i, který je blíže našemu bodu q, můžeme efektivně odstranit kořen a vložit nový bod v čase $\mathcal{O}(\log k)$. Při inicializaci haldy nastavíme všem uzlům maximální možnou vzdálenost. Složitost inicializace haldy se dá zapsat jako $\mathcal{O}(k)$.

2.2 Konstrukce kNN grafu za použití lokálně senzitivního hashování (kNNg LSH)

Tato sekce čerpá z původní práce [18].

Učení na základě grafů *(graph-based learning)* je významná kategorie metod strojového učení, která se hojně využívá v oblastech zpracování obrazu, strojového vidění a data miningu. Tyto metody v prvním kroku reprezentují dataset jako graf podobnosti, na kterém dělají tradiční operace (shlukování, redukce dimenzionality, klasifikace atd.).

Tvorba grafu *hrubou silou* je časově velmi náročná, protože hledání pro každý bod s dimenzí d obnáší porovnání s dalšími n-1 d dimenzionálními body, to znamená časovou složitost $\mathcal{O}(n^2 d)$, což je příliš náročné pro rozsáhlé datasety.

V poslední době se vyvinuly rychlé metody založené na grafech a právě pomalá konstrukce grafu je jejich slabé místo.

V této kapitole si tedy představíme metodu pro tvorbu *aproximativního kNN* grafu za použití *lokálně senzitivní hashování (LSH)*, která je rychlá a přináší vysokou přesnost.

2.2.1 Zavedení pojmů

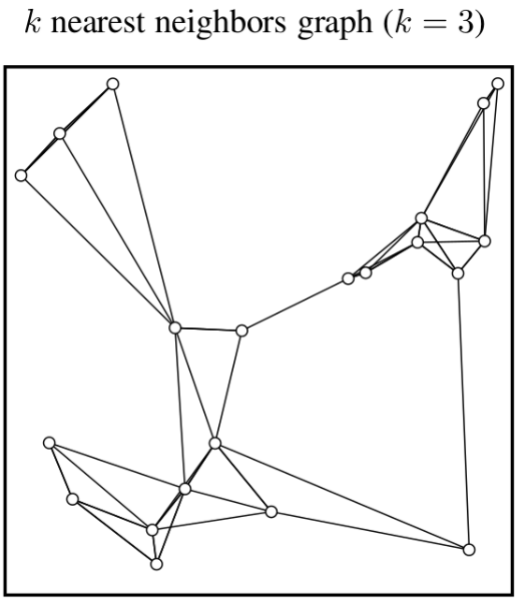
Kromě základních pojmů, které jsme si zavedli v Kapitole 1, si představíme ještě další, které se týkají této problematiky.

2.2.1.1 kNN graf

Mějme nějaký dataset s n body, body p a q jsou spojeny hranou, pokud vzdálenost mezi p a q je mezi $k$tou nejmenší vzdáleností od p k dalším bodům datasetu.

Graf není symetrický, to znamená, že pokud q je nejbližším sousedem p, nemusí platit, že p je nejbližším sousedem q. Graf může být orientovaný, ale není to pravidlo.

Obrázek 2.2: Ukázka neorientovaného kNN grafu, konkrétně pro 3 sousedy [19].



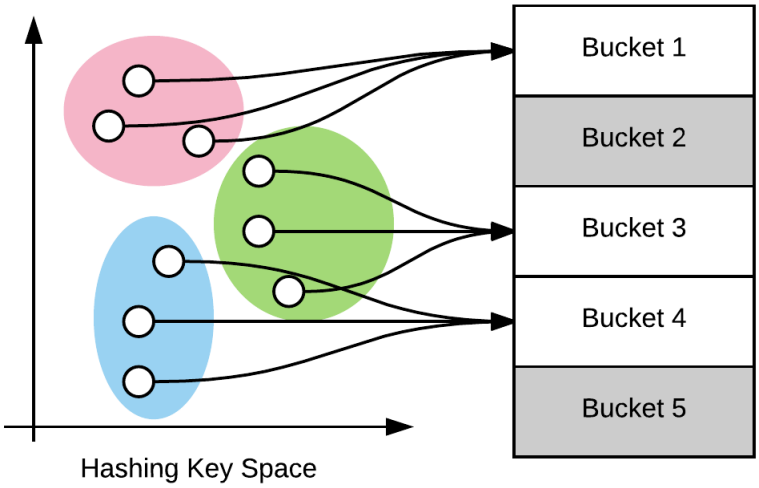
2.2.1.2 Lokálně senzitivní hashování (LSH)

Lokálně senzitivní hashování je efektivní technika pro (nejen) *aproximativní kNN* hledání.

Základním principem *LSH* je rozdělení vícerozměrného prostoru na části, nazývané *buckety*.

Rozdělení jednotlivých bodů do *bucketů* se zachováním podobností znamená ukládání podobných bodů do stejného *bucketu*, naopak odlišné v různých.

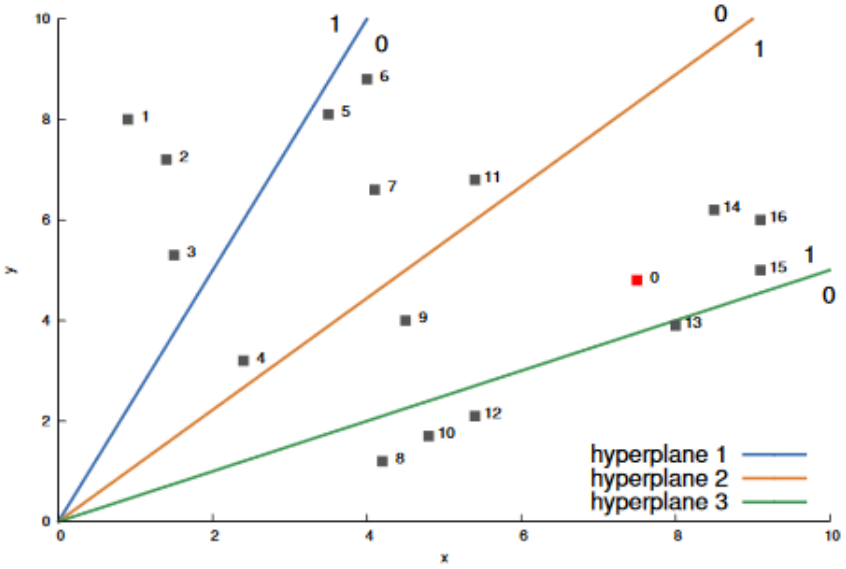
Obrázek 2.3: Ukázka LSH, ukládání podobných bodů do stejného *bucketu* [20].



Hashovací funkce *LSH* převádí d-dimenzionální body na sekvenci m bitů. Odlišností oproti klasickým hashovacím funkcím, které se snaží zabránit kolizím, je to, že *LSH* hashovací funkce se snaží maximalizovat kolize podobných položek. Definujme tuto hashovací funkci následovně:

$$h(x) = \{h_1(x), h_2(x), \ldots, h_m(x) : h_i(x) \in \{0, 1\}\}$$

Obrázek 2.4: Rozdělení 2D prostoru náhodnými nadrovinami jako ilustrace indexování prostoru do bucketů pomocí LSH v metrice kosinové podobnosti [21].



„Na Obrázku 2.4 je vidět rozdělení 2D prostoru třemi nadrovinami. Nadrovina je podprostor dimenze n-1 v prostoru dimenze n. V 2D prostoru je nadrovinou přímka. Každá nadrovina rozdělí prostor na dva podprostory. Z toho plyne, že pokud máme d nadrovin, tak celkový počet podprostorů (bucketů), které tyto nadroviny vymezí, bude #buckets = 2^d. Na Obrázku 2.4 jsou tyto podprostory představovány 0 a 1 u jednotlivých přímek. Každá nadrovina může představovat například 1 bit binárního čísla. Buckety jsou pak například binární čísla 000 nebo 010. V případě vyhodnocování bodu 0 z Obrázku 2.4 je bod umístěn v bucketu (řazeno podle čísel nadrovin) 011. Vyhodnocení a počítání podobnostní funkce pak proběhne na bodech v tomto bucketu, konkrétně pak na bodech 9, 14, 15, 16 a vrátí se k nejbližších.“ [21]

Původní práce [21], které detailně popisuje principy LSH, byla zaměřena na uživatele, nás zajímají obecné body, nicméně myšlenka je totožná.

2.2.2 Algoritmus

Mějme dataset S, který má n bodů, $S = \{x_1, x_2, \ldots, x_n\}$ a nějakou *míru podobnosti* $s(x_i, x_j)$, *kNN* graf pro S je orientovaný, hrana z bodu x_i do bodu x_j existuje právě tehdy, když bod x_j je mezi k nejpodobnějšími body k bodu x_i.

Klíčovou myšlenkou je rozdělení všech bodů do malých skupinek a následné nalezení *kNN* daného bodu ve skupince, do kterého spadá. Z pohledu grafové konstrukce to vypadá tak, že se postaví z každé skupinky malý *kNN* graf a spojením těchto malých grafů vznikne *aproximativní kNN* graf.

Vzhledem k tomu, že hledání *kNN* se provádí pouze v rámci jedné skupinky, nalezení *kNN* bodu q vyžaduje jen *block-sz* porovnání, kde *block-sz* je velikost skupinky, do které spadá. Za předpokladu, že velikost *block-sz* všech skupinek je stejná, složitost této metody při použití *hrubé síly* je $\mathcal{O}(\textit{block-sz} \cdot n \cdot d)$ oproti $\mathcal{O}(n^2 \cdot d)$. Značíme n jako počet bodů, d jako dimenzi dat.

Aby tato metoda fungovala dobře, je třeba splnit dvě podmínky:

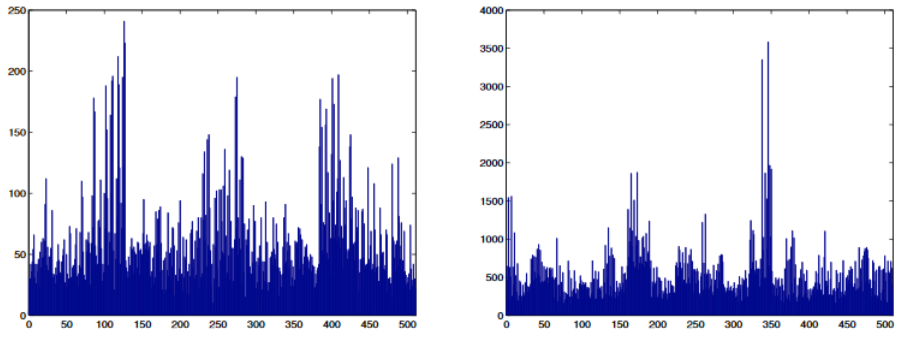
1. Podobné body by měly být pohromadě ve stejné skupince, z čehož plyne, že většina skutečných *kNN* nějakého bodu q je k nalezení v jeho skupince. Dá se tedy předpokládat, že výsledný graf je dobrou *aproximací* skutečného *kNN* grafu.

2. Vzhledem k tomu, že složitost metody je $\mathcal{O}(\textit{block-sz} \cdot n \cdot d)$, je pro rychlost žádoucí, aby velikost jednotlivých skupinek byla co nejmenší ($\textit{block-sz} << n$).

Je zřejmé, že tyto dvě podmínky jdou poměrně proti sobě. Aby dobře fungovala první podmínka, dává smysl použít velké skupinky, což je v rozporu s druhou podmínkou, a naopak. Musíme tedy udělat kompromis a zvolit „rozumnou“ velikost skupinek, abychom uspokojili obě podmínky. Z toho plyne přijatelná rychlost a přesnost.

Prozkoumáme, jak *LSH* dělí dataset tak, že podobné body budou pospolu v jedné skupince. Přímočará cesta je nechat *LSH* zahashovat body do *bucketů*, poté vytvořit *kNN* graf pro každý *bucket*. Nicméně typické *LSH* metody trpí na to, že distribuce v *bucketech* je dosti nerovnoměrná. To znamená, že některé *buckety* obsahují mnoho bodů, naopak některé velmi málo.

Obrázek 2.5: Nerovnoměrné rozdělení bodů do jednotlivých bucketů [18].



(a) Caltech256 (b) NUS-WIDE

Na Obrázku 2.5 je vidět nerovnoměrné rozdělení bodů do jednotlivých *bucketů*. Rozdělení bylo pozorováno na datasetech *Caltech256* a *NUS-WIDE*.

Použitá hashovací funkce vypadá následovně:

$$h_i(x) = \{\text{sgn}({w_i}^T x + b_i)\}, i \in \langle 1, 8\rangle$$

Tato funkce hashuje všechny body do jednotlivých *bucketů*. Hodnoty $\{w_i\}$ jsou získány z normálního (Gaussova) rozdělení $N(0, 1)$ a $\{b_i\}$ jsou mediány hodnot projekcí. Délka kódu $m = 5$, počet *bucketů* je 512.

To přináší následující problémy:

1. *kNN* graf stavěný na *bucketu* s málo prvky nesplňuje Podmínku 1 a selhává na nízkou přesnost.
2. *kNN* graf stavěný na *bucketu* s mnoho prvky nesplňuje Podmínku 2 a selhává na vyšším výpočetním čase, protože má složitost $\mathcal{O}(\textit{block-sz}^2 \cdot d)$.

Představíme si efektivní řešení, jak tento problém obejít a získat tím skupinky stejných velikostí. Mějme matici $Y \in \{0, 1\}^{n \times m}$ hashů bodů x z datasetu S, který má n prvků a m je délka kódu. i-tý řadek $y_i \in \{0, 1\}^m$ je hash bodu x_i.

Nejdříve uděláme promítnutí (projekci) hashů bodů Y na náhodný směrový vektor w délky m a dostaneme $p = Yw$. Potom seřadíme prvky na základě jejich hodnoty projekce a získáme sekvenci $\{x_{\pi_1}, x_{\pi_2}, \ldots, x_{\pi_n}\}$ s hodnotami $\{p_{\pi_1} \leq p_{\pi_2} \leq \ldots \leq p_{\pi_n}\}$.

Nakonec získáme $(n/\textit{block-sz})$ skupinek $\{S_i\}$ se stejnou velikostí *block-sz*, které si definujeme jako $S_i = \{x_{\pi_{(i-1)\cdot \text{block-sz}+1)}}, x_{\pi_{(i-1)\cdot \text{block-sz}+2)}}, \ldots, x_{\pi_{i\cdot \text{block-sz})}}\}$. To pro například $i = 1$ znamená, že $S_1 = \{x_{\pi_1}, x_{\pi_2}, \ldots, x_{\pi_{\text{block-sz}}}\}$. Protože body ve stejném bucketu budou mít i stejnou hodnotu projekce, s vysokou pravděpodobností zůstanou pospolu ve skupince S_i. Popíšeme si zmíněnou část pseudokódem:

Algoritmus 3 Základní *kNN* grafová konstrukce s *LSH*

Vstup: Dataset X, počet sousedů k, délka kódu m, velikost bloku *block-sz*
Výstup: *Aproximativní kNN* graf G

```
procedure BASIC_ANN_BY_LSH(X, k, m, block-sz)
    Y ← LSH(X, m)
    Projekce Y na náhodný směrový vektor w, p = Yw
    Seřaď položky podle p hodnot a dostaň {x_π1, x_π2, ..., x_πn}
    for i ← 1 to n/block-sz do
        S_i = {x_π(i−1)·block-sz+1), x_π(i−1)·block-sz+2), ..., x_πi·block-sz)}
        g_i = brute_force_kNN(S_i, k)
    end for
    return G = ∪{g_i}
end procedure
```

Algoritmus 3 generuje základní *aproximativní kNN* graf. Nicméně je to jen spojení $n/\textit{block-sz}$ izolovaných malých grafů a bude selhávat na nedostatečnou přesnost. Pro zlepšení opakujeme vícekrát Algoritmus 3 s použitím různých hashovacích funkcí.

Algoritmus 4 Konstrukce *aproximativního kNN* grafu s *LSH* - hlavní část

Vstup: Dataset X, počet sousedů k, délka kódu m, velikost bloku *block-sz*, počet iterací l

Výstup: Výsledný *aproximativní kNN* graf G

```
for i ← 1 to l do
    G_i = Basic_ann_by_lsh(X, k, m, block-sz)
end for
Zkombinuj grafy {G_1, G_2, ..., G_l} do grafu G
Zdokonal G jednokrokovou propagací sousedů
return G
```

Označme si množinu *aproximativních kNN* bodu x nalezených v i-té iteraci jako $N_i(x)$, můžeme získat maximálně $k \cdot l$ různých kandidátů v $\{N_i(x)\}$ pro $(1 \leq i \leq l)$. S rostoucím počtem iterací l se více přibližujeme skutečným *kNN*.

Výsledný graf vznikne spojením bodů x s k body z množiny $\{N_i(x)\}$ pro $(1 \leq i \leq l)$, které jsou ke k nejblíže.

Pro další zlepšení této metody se používá *jednokroková propagace sousedů (one step neighbor propagation)*, jak je vidět v Algoritmu 4 . Je založena na následujícím pozorování: pokud bod x je podobný y a y je podobný z, poté s vysokou pravděpodobností je x podobné i z. To znamená, že můžeme vylepšit *aproximované kNN* bodu x tím, že nevybíráme jen z jeho sousedů, ale i ze sousedů sousedů. Neboli vybíráme *kNN* z množiny $N(x) \cup \{\cup_{v \in N_{(x)}} N(v)\}$, což se na základě experimentů v původní přineslo značné vylepšení výsledků.

2.2.3 Časová složitost

V krátkosti shrneme časovou složitost tohoto algoritmu. Značení je následovné: S dataset, n počet bodů v datasetu, d dimenzioalita dat, m délka hash kódu, k počet hledaných sousedů, l počet iterací.

1. Složitost výpočtu $LSH(S, k)$ je $\mathcal{O}(nmd)$, projekce $\mathcal{O}(nm)$ a řezení $\mathcal{O}(n \log n)$. Složitost tvorby všech grafů $g_i(1 \leq i \leq n/block\text{-}sz)$ je $\mathcal{O}(ndblock\text{-}sz)$. Výsledná složitost Algoritmu 3 je tedy $\mathcal{O}(n(md + m + dblock\text{-}sz + n \log n))$.

2. Kombinace l grafů zabere $\mathcal{O}(lnk)$ operací.

3. Každý bod má maximálně k^2+k potenciálních sousedů (k sousedů a k^2 sousedů sousedů) a potřebuje $\mathcal{O}(dk^2)$ operací pro nalezení k nejbližších. To znamená, že krok propagace sousedství zabere $\mathcal{O}(ndk^2)$.

Výsledná složitost je tedy $\mathcal{O}(ln(md + dblock\text{-}sz + \log n + k) + ndk^2)$. Vzhledem k tomu, že k, m a $block\text{-}sz$ jsou v praxi malá čísla, můžeme za výslednou složitost považovat $\mathcal{O}(ln(d + \log n))$.

2.3 Hledání kNN za použití maticových operací (kNN matrix)

Tato sekce čerpá z původní práce [23].

Klasické výpočty *kNN* se skládají ze dvou kroků:

1. Výpočet vzdálenostní matice - pro každý „hledaný bod“ q_i spočítáme vzdálenost od každého referenčního bodu r_j.

2. Seřazení a získání k nejbližších sousedů pro každý bod q_i.

Nicméně my si ukážeme algoritmus, který používá k výpočtům vzdáleností vektorové a maticové operace.

2.3.1 Algoritmus

Základním rozdílem je výpočet vzdálenostní matice. Místo klasického výpočtu Eukleidovské vzdálenosti (popsané v Sekci 1.3.1) využijeme přepisu, který zahrnuje maticové násobení a sčítání.

Mějme dva body x, y v d-dimenzionálním prostoru.

$$x = (x_1, x_2, \ldots, x_d)^T, y = (y_1, y_2, \ldots, y_d)^T$$

Přepíšeme do maticového tvaru:

$$||x - y||_2^2 = d_2^2(x, y) = (x - y)^T(x - y) = ||x||^2 + ||y||^2 - 2x^T y$$

kde $||.||$ je Eukleidovská norma. Například pro bod x:

$$||x||_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_d^2}$$

Tento přístup se dá aplikovat i na množiny bodů.

Mějme matici R o rozměru $d \times m$, která obsahuje m referenčních bodů r_i, a matici Q o rozměru $d \times n$, která obsahuje n query bodů q_i, pro které hledáme sousedy. Dimenzi dat značíme d. Výsledná vzdálenostní matice má rozměry $m \times n$, obsahuje čtverce vzdáleností bodů q_j od všech bodů r_i.

Vzdálenost vypočítáme jako:

$$d_2^2(R, Q) = N_R + N_Q - 2R^T Q$$

Matice N_R, N_Q mají rozměry $m \times n$. Řádek i v N_R obsahuje hodnoty, které jsou všechny rovny Eukleidovské normě $||r_i||^2$. Sloupeček j v matici N_Q obsahuje hodnoty rovny Eukleidovské normě $||q_j||^2$. To sebou samozřejmě přináší poměrně velkou paměťovou náročnost. Vzhledem k tomu, že víme, jaký tvar má výsledná matice mít, stačí nám uložit N_R jako vektor čísel o délce m a N_Q jako vektor délky n. Na i-té pozici vektoru N_R je hodnota $||r_i||^2$, obdobně pro vektor N_Q.

Algoritmus se skládá z osmi kroků, lze vyjádřit pseudokódem:

Algoritmus 5 Pseudokód fází algoritmu *kNN* za použití maticových a vektorových operací

Vstup: Množina referenčních bodů R, množina „hledaných" bodů Q, počet sousedů k

Výstup: Matice k nejbližších sousedů

1: Spočti vektor N_R
2: Spočti vektor N_Q
3: Spočti matici $A = -2R^T Q$ o rozměru $m \times n$
4: Přičti i-tou hodnotu vektoru N_R ke každému prvku v i-tém řádku matice A. Označ matici jako B
5: Seřaď jednotlivé sloupečky B, za použití partial sortu, a označ matici jako C
6: Přičti j-tou hodnotu vektoru N_Q k prvním k prvkům v j-tém sloupečku matice C a označ matici jako D
7: Spočti odmocninu pro každý prvek v prvních k řádcích matice D a označ matici jako E
8: Vyjmi z matice E horní část o rozměru $k \times n$. Výsledná matice odpovídá vzdálenostem k nejbližších sousedů pro každý bod q_j

Výpočetně nejnáročnějším je krok 3. Provedením kroků 6 a 7 až na konci ušetříme čas, jelikož je aplikujeme jen na k řádků, které nás výsledně zajímají. Poslední krok je pouze kopírování.

Pojmenování matic je jen v rámci pseudokódu pro lepší orientaci. Po spočítání matice A jsou všechny zbývající operace dělány *in place*. To znamená, že matice A až E je tatáž matice zabírající stejné místo v paměti.

2.3.2 Časová složitost

V krátkosti shrneme časovou složitost tohoto algoritmu. Značíme m jako počet referenčních bodů, n je počet query bodů, d je dimenze dat.

1. Výpočet vektoru N_R - Eukleidovská norma: $\mathcal{O}(m \cdot d)$
2. Výpočet vektoru N_Q - Eukleidovská norma: $\mathcal{O}(n \cdot d)$
3. Výpočet matice $A = -2R^T Q$: $\mathcal{O}(m \cdot d \cdot n)$
4. Výpočet matice B - přičítání N_R k A: $\mathcal{O}(n \cdot m)$
5. Výpočet matice C - seřazení prvních k prvků ve sloupcích B: $\mathcal{O}(n \cdot k \cdot \log m)$.
6. Výpočet matice D - přičítání N_Q ke k řádkům C: $\mathcal{O}(m \cdot k)$
7. Výpočet matice E - odmocnění k řádků D: $\mathcal{O}(m \cdot k)$

KAPITOLA 3

Implementace algoritmů

V této kapitole si představíme implementace algoritmů z Kapitoly 2 a dalších algoritmů, které potřebujeme pro měření. Každá implementace načítá data ze souboru a zapisuje do souboru matici indexů nejbližších sousedů.

3.1 k-nearest neighbors (kNN)

V této sekci představíme sekvenční a paralelní implementace algoritmu *kNN* pro CPU i GPU.

3.1.1 Sekvenční

Ukážeme si vlastní jednoduchou CPU sekvenční implementaci algoritmu *kNN*, která je psána v jazyce C++.

Jádrem programu je funkce *knn*, která pro každý query bod spočítá Eukleidovskou vzdálenost vůči všem referenčním bodům. Tyto vzdálenosti poté seřadí *partial sortem*, který řadí pouze prvních *k* položek.

Výpočet vzdálenosti vykonává funkce *calculateDistance*.
Implementaci použijeme prvotně jako referenci pro testování paralelní implementace.

3.1.2 Paralelní CPU

Ukážeme si vlastní paralelní CPU implementaci algoritmu *kNN*, která je psána v C++ s použitím knihovny *Open MPI*.

V první řadě se podíváme na základní informace o knihovně Open MPI.

3.1.2.1 Open MPI

Open MPI je open-source implemetace Message Passing Interface, což je komunikační protokol pro paralelní a distribuované výpočty.

Umožňuje paralelizovat celé programy přes množinu (ne)homogenních systémů (například superpočítače), které spolu komunikují po jedné síti.

3.1.2.2 Implementace

Program se skládá z jednoho master procesu, který přijímá výsledky od zbytku procesů, které označíme jako pracovní.

Na začátku si master proces naalokuje pole pro k nejbližších indexů a vzdáleností každého query bodu. Pracovní procesy si rovnoměrně rozdělí query body, pro které počítá vzdálenost od všech referenčních bodů.

Po výpočtu vzdáleností, o který se stará funkce *calculateDistance*, pro jeden query bod pracovní proces seřadí hodnoty *partial sortem*, který řadí prvních k hodnot. Z tohoto seřazeného pole vyjme prvních k hodnot, které předá master procesu. Navíc předá i reálný index tohoto query bodu, aby jej mohl správně umístit do výsledného pole.

Každý pracovní proces si ukládá část query bodů a všechny referenční body. Z tohoto pohledu má poměrně velké paměťové nároky. V tomto případě, kdy výpočty provádíme v rámci jen jednoho serveru, by bylo vhodnější použít OpenMP, které umožňuje sdílení paměti napříč vlákny.

Master proces přijímá výsledky od prvního k poslednímu pracovnímu procesu a vždy v pořadí od prvního query bodu daného procesu. To je opět prostor pro zlepšení, kdy bude přijímat vzdálenosti v pořadí, ve kterém se vyhodnotí.

Po otestování spolehlivosti bude tato implementace použita jako reference z důvodu časové úspory při experimentech.

3.1.3 Paralelní GPU

Ukážeme si paralelní GPU implementaci, která vychází z existujícího řešení [24], je psána v C++ za použití architektury CUDA.

V první řadě se podíváme na základní informace o architektuře CUDA.

3.1.3.1 CUDA

Informace o architektuře CUDA pochází z [22].

Základní ideou obecných výpočtů (general purpose computing) na grafických kartách (GPGPU) je poskytnutí levnějšího řešení s lepšími paralelními vlastnostmi, než je CPU.

NVIDIA vyvinula architekturu s názvem CUDA a kompilátory, které umožňují programátorům psát v běžných jazycích a spouštět programy na NVIDIA grafických kartách.

Grafické karty jsou určeny pro grafické zpracování. Až s příchodem architektury TESLA si společnost NVIDIA uvědomila, že programátoři mohou používat GPU

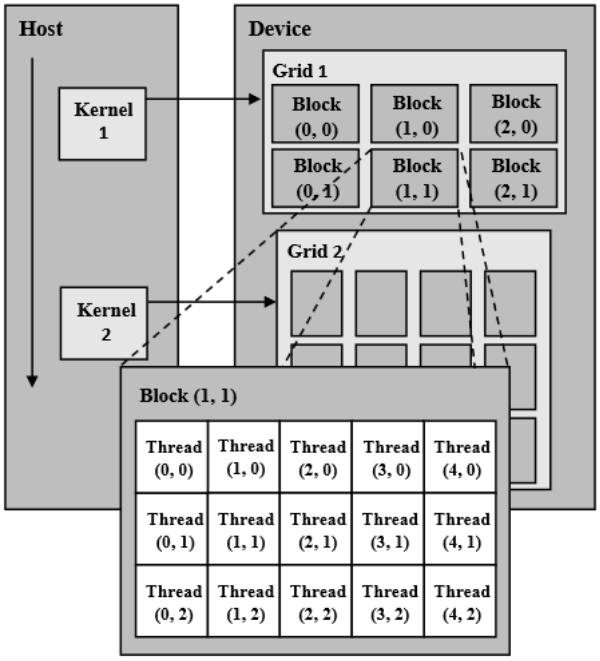
jako procesor a jejich programy na nich paralelizovat. V listopadu 2006 představili CUDA (compute undefined device architecture) s C/C++ kompilátorem a knihovnami pro vývoj paralelních aplikací na grafických kartách. Umožňuje implementovat paralelismus v programu bez znalosti nízkoúrovňové architektury GPU.

Kernel
Kernel je rozšíření jazyka umožňující programátorům definovat funkce, které jsou paralelně Nkrát spuštěny. N udává počet CUDA vláken [25].

Existují tři úrovně vláknové abstrakce:

1. Vlákno (thread): jeden proces, který spouští instanci *kernelu*.
2. Blok: 1-3 dimenzionální kolekce vláken, které sdílí společnou paměť.
3. Grid: 1-2 dimenzionální kolekce bloků.

Obrázek 3.1: CUDA organizace vláken [22].



3.1.3.2 Implementace

Jádrem programu je funkce *knn_cuda_global*. Na začátku se přenesou potřebná data do paměti GPU. O výpočet druhé mocniny Eukleidovské vzdálenosti se stará kernel *compute_distances*. Každé vlákno počítá vzdálenost mezi jedním query bodem a jedním referenčním.

Po dokončení výpočtu vzdáleností se volá kernel *modified_insertion_sort*, kde každé vlákno řadí prvních k hodnot (indexy a vzdálenosti) pro jeden query bod.

Poslední kernel *compute_sqrt* odmocní prvních k vzdáleností jednotlivých query bodů. Každé vlákno odmocňuje jednu hodnotu. Tento výpočet provádíme až nakonec z důvodu časové úspory.

Nakonec překopíruje prvních k indexů a vzdáleností každého query bodu do serverové RAM.

Velikost gridu a bloku záleží na počtu referenčních bodů, query bodů a hodnotě *BLOCK_DIM*, kterou ponecháme na hodnotě 16.

Tuto implementaci budeme používat pro porovnání s paralelní CPU verzí *kNN*, popsanou v Sekci 3.1.2, a GPU verzí s použitím maticových operací, popsanou v Sekci 3.4.

3.2 k-Means pro k-Nearest Neighbors (kMkNN)

Ukážeme si sekvenční implementaci algoritmu *kMkNN*, který jsme představili v Sekci 2.1. Vychází z BiocNeighbors [26], což je balíček pro hledání exaktních a aproximativních sousedů, který je psán v R s C++.

V první řadě se podíváme na základní informace o jazyce R.

3.2.1 R

„R je jazykem a prostředím pro statistické výpočty a grafiku.

Poskytuje širokou škálu statistických (lineární a nelineární modely, klasické testy, analýza časových řad, klasifikace, klastrování, ...) a grafických technik.

Jednou z nejsilnějších částí R je snadnost, s kterou lze vytvářet dobře navrhnuté obrázky a grafy v profesionální kvalitě. Do grafů lze snadno v případě potřeby vkládat matematické symboly a vzorce.

R je dostupné jako volně šiřitelný software (Free Software) při dodržení podmínek GNU General Public License nadace Free Software Foundation (nevylučuje komerční využití programu).“ [27]

3.2.2 Implementace

V přípravné fázi, kterou jsme popsali v Sekci 2.1.2.1, voláme funkci *buildKmknn*. Jejím základem je R implementace algoritmu k-means. Počet clusterů je nastaven na hodnotu $\lceil \sqrt{n} \rceil$, kde n je počet referenčních bodů. Po dokončení shlukování projde jednotlivé shluky a seřadí vzdálenosti bodů od středu daného shluku sestupně.

Pro hledání, které jsme popsali v Sekci 2.1.2.2, voláme funkci *queryKNN*. Ta na základě parametru *BNPARAM=KmknnParam()* volá C++ funkci *query_knn*, která hledá k nejbližších sousedů pro každý query bod. Pravidla pro hledání jsou definovány ve třídě *kmknn*, konkrétně v metodě *search_nn*. Pro každý query bod spočítá

vzdálenost od středů všech shluků, tyto vzdálenosti seřadí od nejmenší. Na základě trojúhelníkové nerovnosti se rozhodne, jaké shluky má smysl procházet, stejně jako jsme popsali v Sekci 2.1.2.2.

V obou fázích používáme Eukleidovskou vzdálenost.

3.3 Konstrukce kNN grafu (kNNg)

V této sekci si představíme implementace algoritmů pro tvorbu *kNN* grafu.

3.3.1 Konstrukce hrubou silou

Ukážeme si vlastní sekvenční implementaci základního algoritmu pro tvorbu *kNN* grafu hrubou silou, která je psána v C++.

Pro každý bod spočítá Eukleidovskou vzdálenost od ostatních bodů, výpočet vzdálenosti mezi dvěma body obstarává funkce *distance*. Po výpočtu vzdálenosti mezi jedním bodem a všemi ostatními seřadí prvních k vzdáleností za použití *partial sortu*. Po seřazení přidá prvních k hodnot k výsledkům.

Tuto implementaci použijeme pro měření přesnosti aproximativního řešení.

3.3.2 Konstrukce za použití LSH

Ukážeme si vlastní sekvenční implementaci algoritmu, který jsme představili v Sekci 2.2. Je psána v C++.

Jádrem programu je funkce *basic_ann_by_lsh*. V této funkci inicializujeme třídu *Lsh*, která na základě normální rozdělení N(0, 1) vytvoří hashovací funkce a náhodný vektor. Zahashujeme referenční body pomocí metody *getHash* ze třídy *Knn*, kde binarizaci (převod do dvojkové soustavy) provádíme tak, že pokud $h_i(x) \geq 0$, nastavíme bit na 1, jinak 0. Uděláme projekci získaných hashů na náhodný vektor pomocí metody *projection* ze třídy *Lsh*. Získané hodnoty seřadíme vzestupně a rozdělíme body do skupinek velikosti *block-sz* podle pořadí hodnot projekce. Pro každou skupinku zavoláme statickou metodu *isolatedGraph* ze třídy *Knn*, která vytváří malé izolované *kNN* grafy. Tedy pro každý bod spočítá Eukleidovskou vzdálenost, kterou počítá statická metoda *distance* také ze třídy *Knn*.

S každým volání funkce *basic_ann_by_lsh* vznikne *aproximativní* graf, který je spojením izolovaných grafů. Toto provedeme celkem *i*krát, kde *i* je počet iterací.

Po *i* iteracích funkce *basic_ann_by_lsh* získáme *i aproximativních* grafů. Ve všech získaných grafech se podíváme na k nejbližších sousedů každého bodu. Tyto sousedy ukládáme do setu, abychom nemuseli řešit duplicity. Poté se podíváme na potenciální získané sousedy každého bodu a vypočítáme Eukleidovskou vzdálenost. Opět pomocí *partial sortu* seřadíme prvních k hodnot. Tím získáme už téměř finální *aproximativní* graf.

Nakonec graf vylepšíme *jednokrokovou propagací sousedů*, kterou jsme si představili v Sekci 2.2.2. To znamená, že pro každý bod projdeme i sousedy sousedů, které si uložíme do setu k původním sousedům daného bodu. Spočítáme Eukleidovskou vzdálenost vůči všem získaným potenciálním sousedům. Poté seřadíme prvních k hodnot *partial sortem*. Těchto prvních k sousedů prohlásíme za výsledné sousedy daného bodu.

Výchozí *block-sz* je nastavena na 50, délka hash kódu $\lceil \log_2 (n/\text{block} - \text{sz}) \rceil + 1$.

Prostor pro zlepšení je rozumné ukládání již spočítaných vzdáleností, protože reálně některé vzdálenosti počítáme mnohokrát. Dále zrychlení hashování bodů a projekce pomocí efektivní knihovny pro vektorové/maticové operace (například CBLAS).

3.4 Hledání kNN za použití maticových operací (kNN matrix)

Ukážeme si paralelní GPU implementaci, kterou jsme si představili v Sekci 2.3. Vychází z existujícího řešení [24]. Je psána v C++ za použití architektury CUDA, kterou jsme si představili v Sekci 3.1.3.1, s cuBLAS [28]. Knihovna cuBLAS je CUDA implementace BLAS (Basic Linear Algebra Subprograms), což je knihovna optimalizovaná pro vektorové a maticové operace.

Jádrem programu je funkce *knn_cublas*. Na začátku se přenesou potřebná data do paměti GPU.

Prvním kernelem je *compute_squared_norm*. Počítá druhou mocninu Eukleidovské normy, kterou jsme definovali v Sekci 2.3.1, referenčních bodů. Každé vlákno zpracovává jeden referenční bod.

Druhým kernelem je *compute_squared_norm*, nicméně tentokrát počítá Eukleidovskou normu pro query body. Každé vlákno zpracovává opět jeden query bod.

Třetí kernel, který je výpočetně nejnáročnější, je *cublasSgemm*. Využívá knihovnu cuBLAS. Stará o výpočet matice $A = -2R^T Q$, neboli násobí transponovanou matici referencí s maticí query.

Čtvrtý kernel *add_reference_points_norm* přičítá k matici A z předešlého kroku vypočítané Eukleidovské normy referenčních bodů. Tedy ke každé hodnotě ve sloupečku i matice A přičte druhou mocninu Eukleidovské normy referenčního bodu i. Každé vlákno se stará o jedno sčítání.

Pátým kernelem je *modified_insertion_sort*, kde každé vlákno řadí prvních k hodnot (indexy a vzdálenosti) pro jeden query bod.

Posledním kernelem je *add_query_points_norm_and_sqrt*. Přičítá k matici z předešlého kroku vypočítané Eukleidovské normy query bodů. Tedy k prvním k hodnotám ve sloupečku j matice přičte druhou mocninu Eukleidovské normy query bodu j. Každé vlákno se stará o jedno sčítání a následné odmocnění.

Nakonec překopíruje prvních k indexů a vzdáleností každého query bodu do serverové RAM.

Velikost gridu a bloku záleží na počtu referenčních bodů, query bodů a hodnotě *BLOCK_DIM*, kterou ponecháme na hodnotě 16.

KAPITOLA 4

Testování a diskuse

V této kapitole budeme experimentovat s algoritmy, které jsme si teoreticky představili v Kapitole 2 a implementovali v Kapitole 3.

4.1 Použité prostředky

Server pro měření používá systémem Ubuntu 18.04. Je vybaven 16GB DDR3 RAM, dvěma procesory Intel© Xeon© CPU E5-2670, kde každý z nich má osm fyzických jader s hyper-threadingem, a grafickou kartou ASUS Turbo GeForce RTX™ 2070 8GB GDDR6 s 2304 CUDA jádry.

Verze CUDA ovladačů: 10.0.130
Verze g++: 7.4.0
Verze Open MPI: 3.3a2
Verze R: 3.6.0

4.2 Datasety

Pro naše testování budeme používat synteticky generované datasety, které mají číselné hodnoty podle normálního rozdělení $N(0, 1)$.

Zvolil jsem umělé datasety z toho důvodu, že budeme zkoumat, jak nám čas a přesnost ovlivňují změny jednotlivých parametů, jako jsou počet referenčních bodů, počet query bodů, dimenze a další.

S reálnými daty by to znamenalo velké množství úprav získaných dat, které by reálně nepřispěly k přínosu této práce.

Výchozí rozměry datasetu jsou:

1. počet referenčních bodů: 50000,
2. počet query bodů: 1000,
3. dimenze dat: 128.

Jednotlivé rozměry budeme v rámci měření exponenciálně zvyšovat. Velikost je přizpůsobena hardwarovým prostředkům, na kterých probíha měření a samozřejmě času.

4.3 Testování implementovaných algoritmů

V této části budeme zkoumat reálný čas běhu algoritmů, vzhledem k zadaným parametrům. Počet sousedů necháváme fixně pro všechny algoritmy na 16.

4.3.1 kNN

V této sekci provedeme měření paralelníchch implementací algoritmu *kNN* na CPU a GPU.

4.3.1.1 Palalelní CPU

Měření paralelní implementace na CPU, kterou jsme si představili v Sekci 3.1.2.

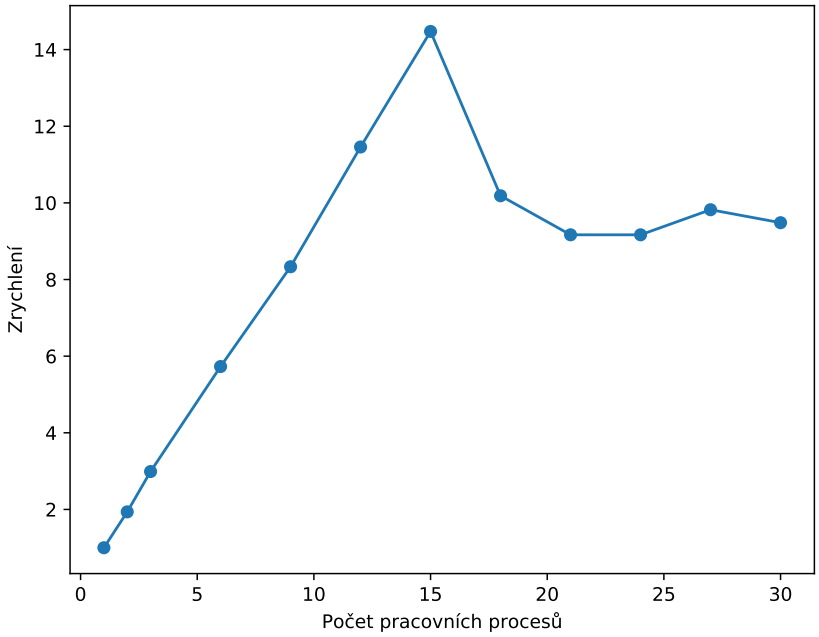
Na základě měření bylo ověřeno, že implementace skutečně poskytuje správné výsledky.

Nyní budeme zkoumat, jaké je zrychlení výpočtu vzhledem k počtu procesů vůči sekvenčnímu řešení. Na základě Obrázku 4.1 je vidět, že nejlepšího zrychlení dosahuje při 15 pracovních procesech (celkově 16 procesů). S rostoucím počtem procesů už nedosahuje zlepšení, ba naopak. Je to způsobené tím, že server má reálně 16 fyzických jader. Hyper-threading zde selhává, protože každý proces je vytížen celou dobu na 100%. Dochází pak k vzájemnému brzdění mezi procesy [29]. Do zmíněných 16 procesů je zrychlování lineární, čehož jsme chtěli dosáhnout.

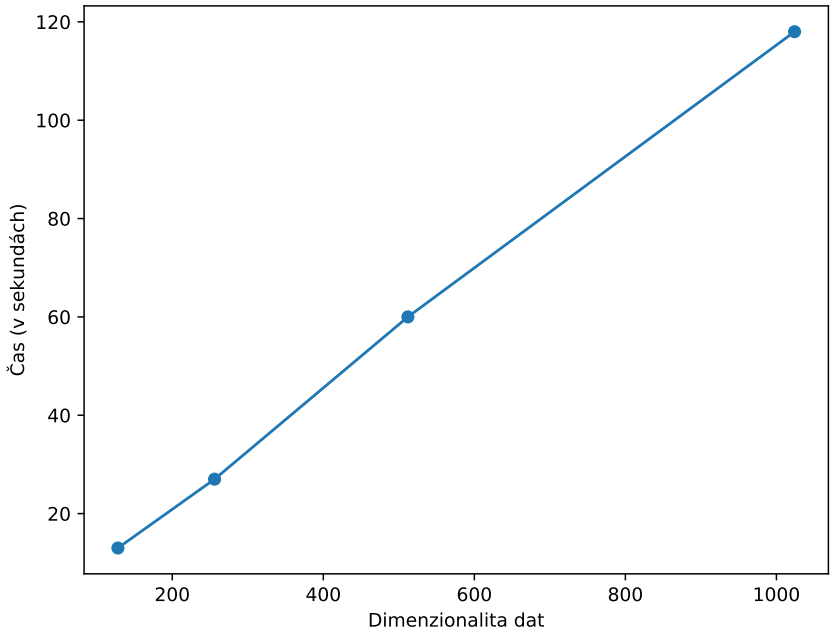
Ověřili jsme, že čas lineárně roste vůči počtu referenčních a query bodů. Totéž platí i pro dimenzionalitu dat, která je znázorněna na Obrázku 4.2.

Implementace se tedy chová správně vzhledem k předpokladu časové složitosti algoritmu ze Sekce 1.2.2.

Obrázek 4.1: Zrychlení kNN výpočtu vůči počtu procesů.



Obrázek 4.2: Škálovatelnost kNN vůči dimenzionalitě dat na CPU.



4.3.1.2 Palalelní GPU

Měření paralelní implementace na GPU, kterou jsme si představili v Sekci 3.1.3.

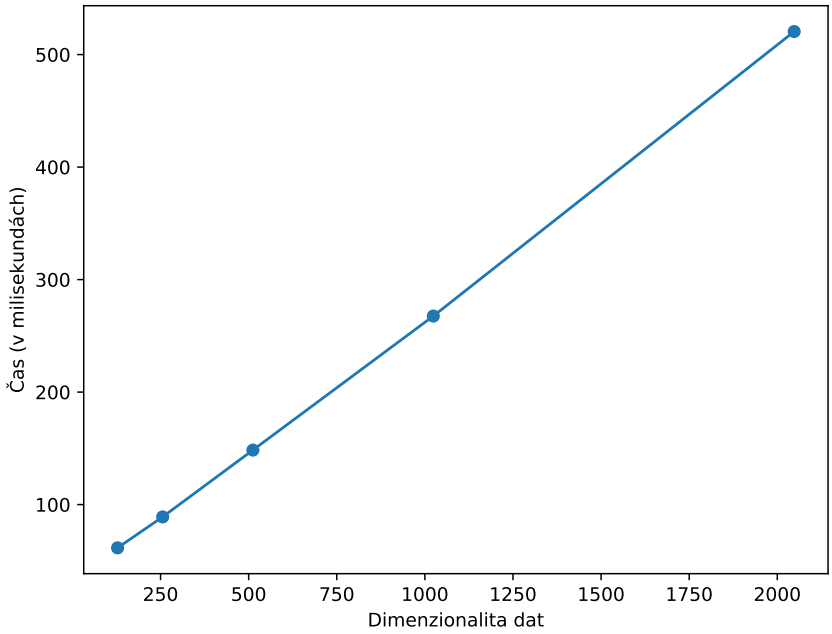
Vzhledem k tomu, že se pohybujeme časově v řádech milisekund, budeme každý výpočet pouštět tisíckrát a získané časy průměrovat, aby výsledky byly co nejméně zkresleny výkyvy.

Konečné výsledky každého měření jsme porovnávali vůči referenčním výsledkům, které najdeme v Sekci 4.3.1.1, a ověřili jsme, že opravdu vrací správné exaktní sousedy.

Stejně jako u paralelní implementace na CPU jsme ověřili, že čas lineárně roste vůči počtu referenčních a query bodů. Nicméně vizuálně ukážeme jen škálovatelnost oproti dimenzionalitě dat, která je vidět na Obrázku 4.3, kde čas také lineárně roste s dimenzionalitou dat.

Implementace se tedy chová správně vzhledem k předpokladu časové složitosti algoritmu ze Sekce 1.2.2.

Obrázek 4.3: Škálovatelnost kNN vůči dimenzionalitě dat na GPU.



4.3.2 kMkNN

Měření sekvenční implementace, kterou jsme uvedli v Sekci 3.2.

Reálný čas přípravné fáze, kde se používá algoritmus *k*-means velmi záleží na počátečním výběru bodů, které bereme jako středy shluků. U měření na stejném datasetu pozorujeme i několikanásobné změny v čase. Budeme tedy průměrovat časy na základě deseti měření, abychom pozorovali o něco méně zkreslené výsledky.

Tento jev má podstatně menší dopad na čas fáze hledání, nicméně opět budeme pozorovat čas na základě průměru deseti měření.

V původní práci [17] nebyl kladen téměř žádný důraz na zkoumání přípravné fáze, zabývali se primárně časem hledání. V této práci budeme pozorovat i celkový čas běhu algoritmu *kMkNN*.

Naměřené hodnoty tentokrát reprezentujeme tabulkou. Důvodem je úspora místa. Zaměřujeme se na přípravnou i hledací fázi. Zkoumáme zrychlení vůči paralelní implementaci algortitmu *kNN* ze Sekce 4.3.1.1, kde jsme hodnoty naměřeného času převedli na CPU čas, kvůli férovému porovnání.

V první řadě se zaměříme na pozorování hledací fáze. Naměřili jsme lineární přírůstky v čase vzhledem k rostoucímu počtu referenčních bodů, jak znázorňuje Tabulka 4.1. Čas lineárně roste také s počtem query bodů, viz Tabulka 4.2, a také s rostoucí dimenzionalitou dat, Tabulka 4.3. Implementace hledací fáze se tedy chová správně vzhledem k předpokladu časové složitosti části tohoto algoritmu ze Sekce 2.1.4.2.

V druhé řadě se podíváme na přípravnou fázi. Zde už nemůžeme porovnávat s předpokládanou časovou složitostí ze Sekce 2.1.4.1, protože nedokážeme reálně pozorovat počet iterací. Ukážeme si tedy pouze, jak reálně škáluje.

Vůči rostoucímu počtu referenčních bodů pozorujeme lineární časový přírůstek, jak je vidět v Tabulce 4.1. Počet referenčních bodů v této fázi nemá žádný význam, lze pozorovat v Tabulce 4.2. Naměřili jsme obrovské časové skoky vzhledem k rostoucí dimenzionalitě dat, které jsou k vidění v Tabulce 4.3. Pozorujeme velký dopad *Prokletí dimenzionality*, které jsme rozebírali v Sekci 1.4.1.

Tabulka 4.1: Škálovatelnost kMkNN vůči počtu referenčních bodů, zrychlení.

Parametr	kMkNN (s)			kNN (s)	Zrychlení	
# referencí (tis.)	Příprava	Hledání	Celkem	Hledání	Hledání	Celkově
50	38,41	10,87	49,28	208	19,13	4,22
100	68,80	19,96	88,76	432	21,64	4,86
200	118,50	50,08	168,58	944	18,84	5,59
400	496,50	102,07	598,57	2096	20,53	3,50

Tabulka 4.2: Škálovatelnost kMkNN vůči počtu query bodů, zrychlení.

Parametr	kMkNN (s)			kNN (s)	Zrychlení	
# query (tis.)	Příprava	Hledání	Celkem	Hledání	Hledání	Celkově
1	38,41	10,87	49,28	208	19,13	4,22
2	39,97	24,22	64,19	432	17,83	6,73
4	40,20	48,85	89,05	688	14,08	7,72
8	38,97	97,36	136,33	1312	13,47	9,62

Tabulka 4.3: Škálovatelnost kMkNN vůči dimenzionalitě dat, zrychlení.

Parametr	kMkNN (s)			kNN (s)	Zrychlení	
Dimenze	Příprava	Hledání	Celkem	Hledání	Hledání	Celkově
128	38,41	10,87	49,28	208	19,13	4,22
256	94,07	21,57	115,64	432	20,03	3,73
512	2266,34	44,58	2310,92	960	21,53	-2,40
1024	6420,62	91,96	6512,58	1888	20,53	-3,44

4.3.3 Konstrukce kNN grafu za použití lokálně senzitivního hashování (kNNg LSH)

Měření sekvenční implementace, kterou jsme představili v Sekci 3.3.2.

Pro toto měření budeme používat menší výchozí dataset a to s rozměry:

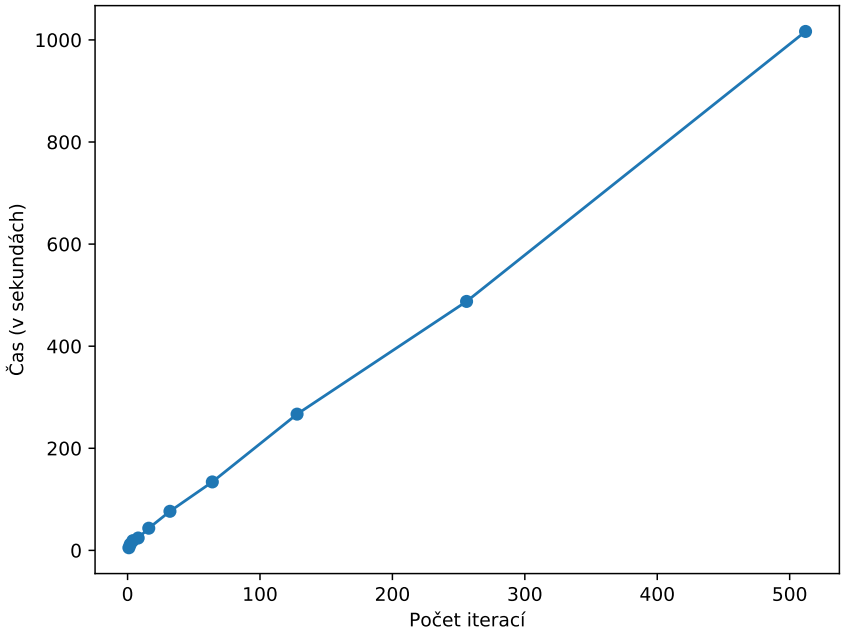
1. Počet bodů: 10000
2. Dimenze dat: 128

Důvodem omezení je složitost tvorby grafu hrubou silou ze Sekce 3.3.1, která se pohybuje s druhou mocninou počtu bodů. Proti výsledkům zmíněné implementace ověřujeme přesnost vyhodnocování. Veškeré časy průměrujeme na základě deseti měření.

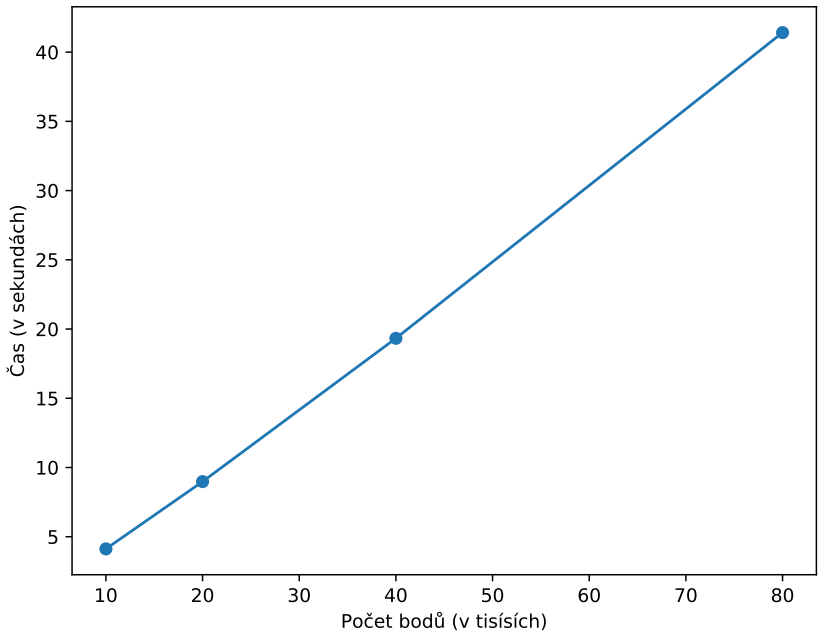
Naměřili jsme lineání příbytky v čase s rostoucím počtem iterací, jak znázorňuje Obrázek 4.4. Čas také lineárně roste s počtem bodů, jak je vidět na Obrázku 4.5, a dimenzionalitou dat, kterou vidíme na Obrázku 4.6.

Tím jsme ověřili, že implementace se chová správně vzhledem k předpokladu časové složitosti algoritmu ze Sekce 2.2.3.

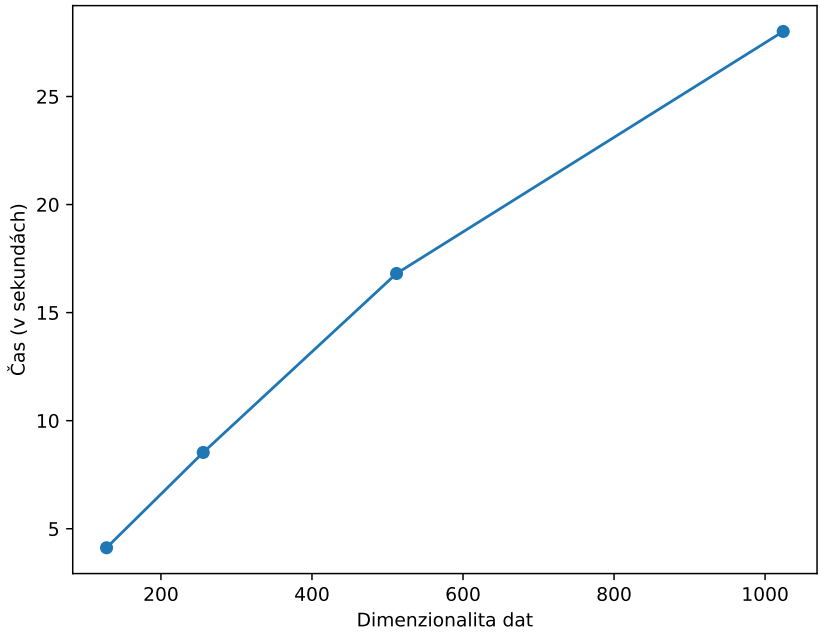
Obrázek 4.4: Škálovatelnost kNNg LSH vůči počtu iterací.



Obrázek 4.5: Škálovatelnost kNNg LSH vůči počtu bodů.



Obrázek 4.6: Škálovatelnost kNNg LSH vůči dimenzionalitě dat.



Přesnost aproximativního grafu G' vyhodnocujeme následovně:

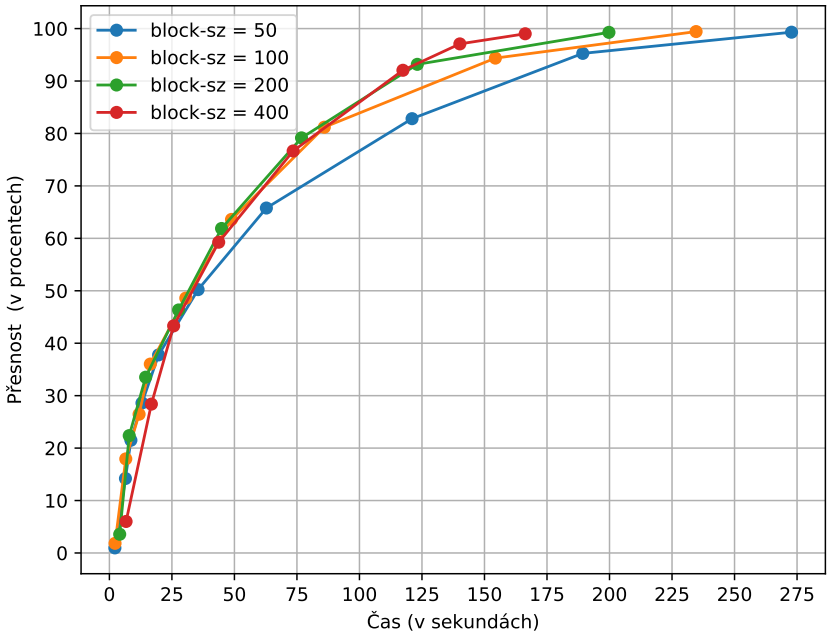
$$\text{acc}(G') = \frac{|E(G') \cap E(G)|}{|E(G)|}$$

kde G je exaktní graf získaný metodou hrubé síly, představena v Sekci 3.3.1, $E(.)$ označuje množinu orientovaných hran grafu a $|.|$ počet hran.

Měření přesnosti vyhodnocování budeme pozorovat na datasetu výchozí velikosti, kterou jsme stanovili v Sekci 4.3.3. Zaměříme se na vliv velikosti *block-sz*.

Čas běhu metodou hrubé síly ze Sekce 3.3.1 je 432 vteřin.

Obrázek 4.7: Přesnost vyhodnocování použitím LSH.



4.3.4 kNN za použití maticových operací

Měření paralelní implementace na GPU s knihovnou cuBLAS, kterou jsme představili v Sekci 3.4.

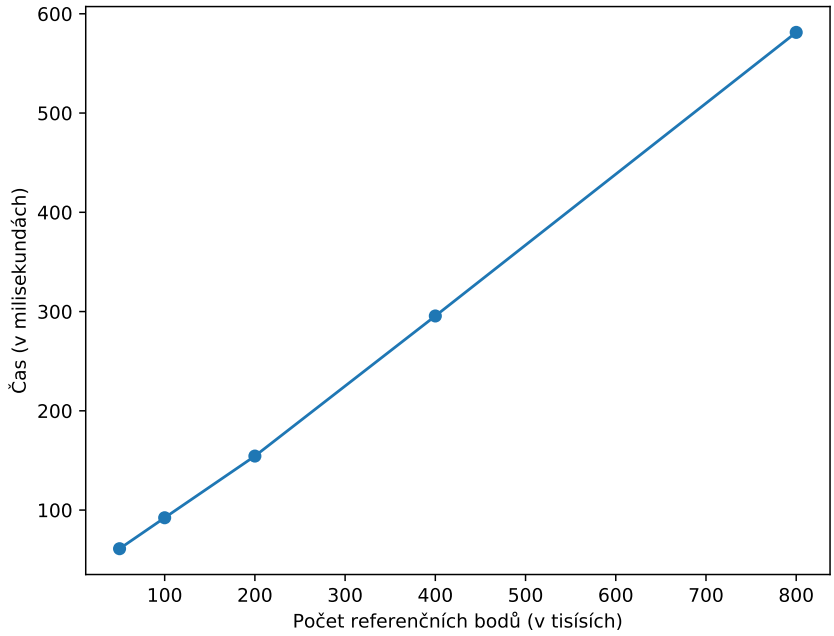
Stejně jako u měření ze Sekce 4.3.1.2 budeme každý výpočet pouštět tisíckrát a získané časy průměrovat, aby výsledky byly co nejméně zkresleny výkyvy.

Konečné výsledky každého měření jsme porovnávali vůči referenčním výsledkům, které jsme získali v Sekci 4.3.1.1, a ověřili jsme, že výsledkem jsou exaktní sousedi.

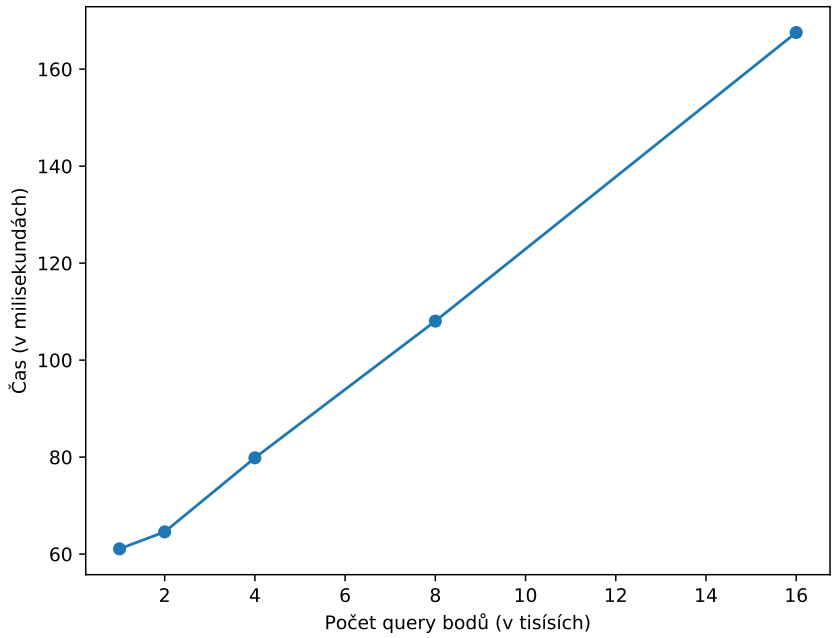
Naměřili jsme lineání příbytky v čase s rostoucím počtem referenčních bodů, jak znázorňuje Obrázek 4.8. Čas roste také lineárně s počtem query bodů, jak je vidět na Obrázku 4.9, a dimenzionalitou dat, kterou vidíme na Obrázku 4.10.

Tím jsme ověřili, že implementace se chová správně vzhledem k předpokladu časové složitosti algoritmu ze Sekce 2.3.2.

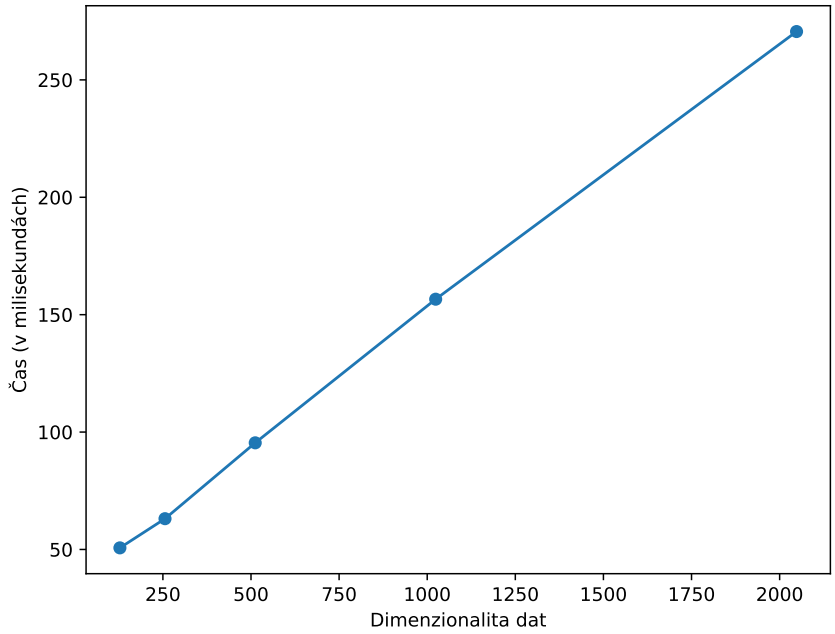
Obrázek 4.8: Škálovatelnost kNN vůči počtu referenčních bodů na GPU s cuBLAS.



Obrázek 4.9: Škálovatelnost kNN vůči počtu query bodů na GPU s cuBLAS.

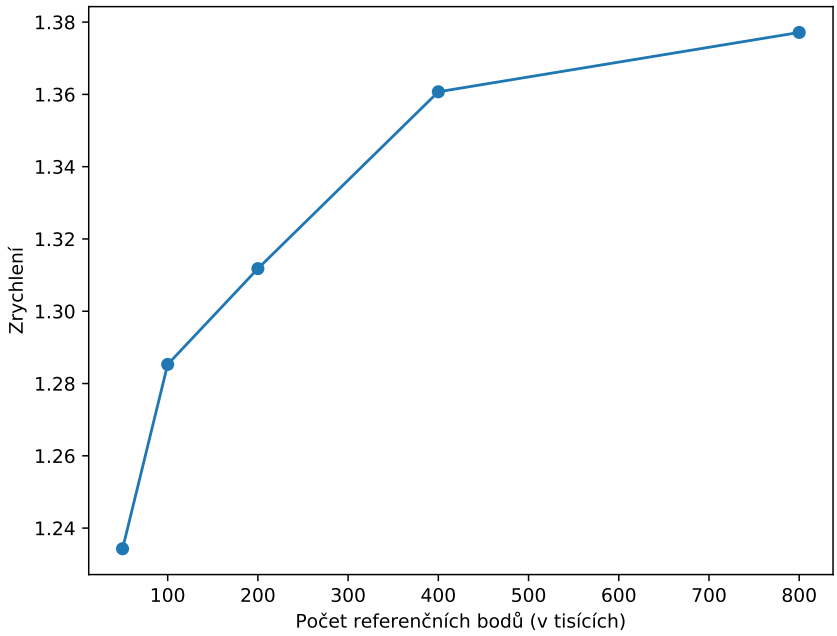


Obrázek 4.10: Škálovatelnost kNN vůči dimenzionalitě dat na GPU s cuBLAS.

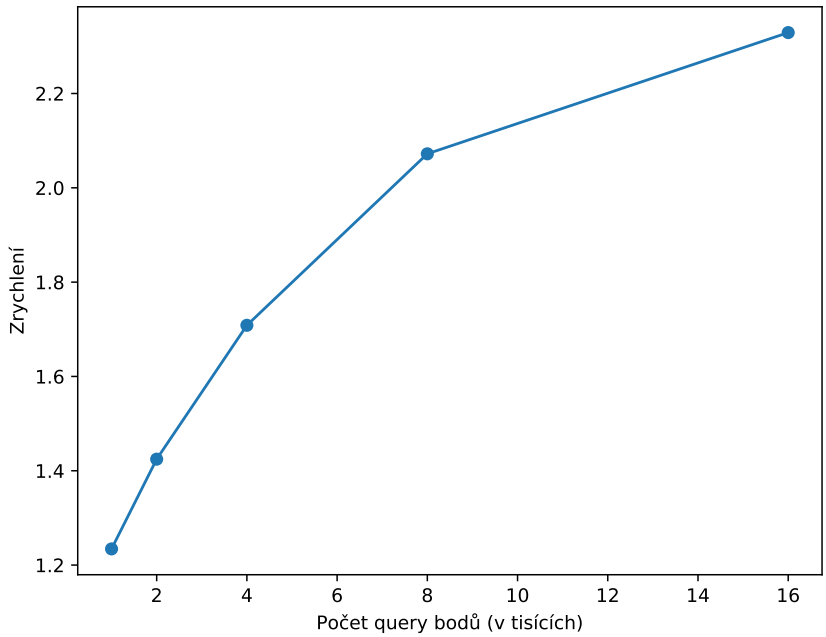


Nicméně na těchto grafech není patrné zrychlení oproti výsledkům předchozí GPU implementace, které jsme naměřili v Sekci 4.3.1.2. To si ukážeme na následujících grafech. Implementace cuBLAS zvyšuje časový náskok s rostoucím počtem referenčních bodů, jak je vidět na Obrázku 4.11. Zrychluje (relativně) také s počtem query bodů, což znázorňuje Obrázek 4.12, a rostoucí dimenzionalitou dat, Obrázek 4.13.

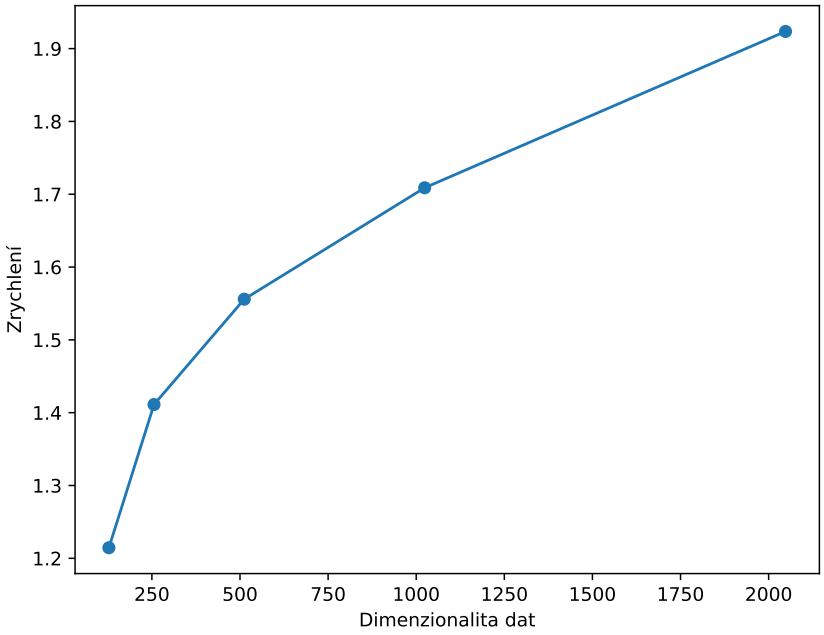
Obrázek 4.11: Zrychlení výpočtu s použitím cuBLAS, rostoucí počet referenční bodů.



Obrázek 4.12: Zrychlení výpočtu s použitím cuBLAS, rostoucí počet query bodů.



Obrázek 4.13: Zrychlení výpočtu s použitím cuBLAS, rostoucí dimenzionalita dat.



4.4 Diskuse a shrnutí

V první řadě se podíváme na výsledky algoritmu *kMkNN* ze Sekce 4.3.2. Pokud nás zajímají výsledky pouze hledací fáze, které vidíme v Tabulce 4.1, dosáhli jsme v průměru dvacetinásobného zrychlení oproti algoritmu *kNN*, který jsme rozebírali v Sekci 3.1.2.

Zajímá-li nás porovnání s celkovým během *kMkNN* (přípravná i hledací fáze), velmi tvrdě narážíme u vysoké dimenze (512 a 1024), jak je vidět v Tabulce 4.3, kdy je *kMkNN* dokonce pomalejší, než *kNN*. Pozorujeme dopad *Prokletí dimenzionality*, které jsme popsali v Sekci 1.4.1. Naopak vzhledem k rostoucímu počtu referenčních bodů, jak je vidět v Tabulce 4.1, a query bodů, Tabulka 4.2, pozorujeme průměrně šestinásobné zrychlení.

Druhým algoritmem je *Konstrukce kNN grafu za použití lokálně senzitivního hashování*, který jsme měřili v Sekci 4.3.3. Ten jsme porovnávali s algoritmem pro tvorbu *kNN* grafu hrubou silou ze Sekce 3.3.1. Nejlepších výsledků, které znázorňuje Obrázek 4.7, jsme dosáhli při volbě *block-sz* = 400. Zhruba ve třetinovém čase jsme získali aproximativní graf s 99,01% přesností.

Posledním algoritmem je *kNN za použití maticových operací*, který jsme měřili v Sekci 4.3.4. Porovnáváme jej s *kNN CUDA* implementací ze Sekce 3.1.3. S rostoucí velikostí datasetu pozorujeme (relativní) zrychlení. Nejvíce se projevuje rostoucí počet query bodů a dimenzionalita, znázorněno na Obrázcích 4.12 a 4.13, kde jsme dosáhli až dvojnásobného zrychlení. U rostoucího počtu referenčních bodů jsme

dosáhli zrychlení zhruba 40%, Obrázek 4.11.

Co se týče zrychlení výpočtů použitím GPU, porovnáme mezi sebou paralelní implementace *kNN* na CPU ze Sekce 3.1.2 (konkrétně 16 jader) a GPU ze Sekce 3.1.3 (konkrétně 2304 jader).

Tabulka 4.4: Zrychlení CPU/GPU vůči počtu referenčních bodů

#referencí (tis.)	GPU (s)	CPU (s)	Zrychlení
50	0,07539	13	172,43
100	0,11858	27	227,69
200	0,20254	59	291,30
400	0,40216	131	325,7
800	0,80049	344	429,73

Tabulka 4.5: Zrychlení CPU/GPU vůči počtu query bodů

#query (tis.)	GPU (s)	CPU (s)	Zrychlení
1	0,07539	13	172,43
2	0,092	27	293,47
4	0,13642	43	315,20
8	0,22387	82	366,28
16	0,39025	160	409,99

Tabulka 4.6: Zrychlení CPU/GPU vůči dimenzionalitě dat

Dimenze	GPU (s)	CPU (s)	Zrychlení
128	0,06155	13	211,21
256	0,08906	27	303,16
512	0,14846	60	404,14
1024	0,26759	118	440,97

Na základě Tabulek 4.4, 4.5 a 4.6 pozorujeme, že jsme v průměru dosáhli třistanásobného zrychlení. To znamená, že pro dosažení stejných časových hodnot bychom teoreticky museli mít k dispozici 4800 CPU jader.

Tento zásadní rozdíl je způsoben paralelní architekturou GPU. Skládá se z mnohem většího počtu jader, než CPU, které jsou samy o sobě podstatně výpočetně slabší a jednodušší, než CPU jádra. GPU vyniká na nezávislých výpočtech bez nějaké větší logiky, díky vysokému počtu jader a velmi rychlé paměti [30].

Algoritmus *kNN* má velmi dobré paralelní vlastnosti. Většina výpočtů je na sobě nezávislá, tudíž plně paralelizovatelná (například výpočet vzdálenosti) a také částečně

paralelizovatelné výpočty (například řazení) [22]. Díky těmto vlastnostem těží z paralelní architektury GPU a velmi zřetelně překonává CPU. Tudíž použití GPU pro *kNN* výpočty je velmi výhodné.

Jediný problém, který jsem při použití GPU pozoroval, je velikost paměti. I nejmodernější běžně dostupné karty (nikoli GPU pro datacentra) mají maximálně 11GB paměti, která může být pro některé výpočty nedostatečná. To jsem pocítil při testování cuBLAS implementace ze Sekce 3.4, kde jsem maximálně využil dostupnou 8GB GPU paměť. Je proto důležité s tímto faktem počítat a případně vhodně rozdělit výpočty na více částí.

Veškeré algoritmy založené na hledání nejbližších sousedů, kterými jsme se zabývali v Kapitole 2, jsou vhodné pro GPU, jelikož jejich základem je výpočet vzdáleností a řazení stejně jako u základního algoritmu *kNN*.

Závěr

Cílem práce bylo seznámit se s algoritmy založenými na hledání nejbližších sousedů a experimentovat s nimi.

V první kapitole jsme si zavedli základní pojmy, které jsou podstatné k pochopení dané problematiky.

V druhé kapitole jsme si představili tři konkrétní algoritmy založené na hledání nejbližších sousedů, včetně pseudokódů a asymptotické složitosti.

Ve třetí kapitole jsme tyto algoritmy implementovali, případně upravili existující implementace vzhledem k našim účelům. Dále byly implementovány další podpůrné algoritmy, které jsme potřebovali pro měření a vyhodnocování. Navíc byla implementovaná paralelní verze algoritmu *kNN* na CPU s použitím knihovny Open MPI. Představili jsme si i implementace na GPU s použitím architektury CUDA.

V poslední kapitole jsme uvedli prostředky pro měření a experimentovali s výpočty na celkem 17 syntetických datasetech. Ověřili jsme, že algoritmy dosahují předpokládané časové složitosti. Zkoumali jsme, jak algoritmy škálují vzhledem k rostoucímu počtu referenčních bodů, query bodů a rostoucí dimenzionalitě dat. Na závěr jsme shrnuli poznatky z měření a možnosti zrychlení výpočtů za použití grafické karet.

Literatura

[1] MÜLLER, Andreas; GUIDO, Sarah: *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media, Inc., 2016, ISBN 978-1449369415, [cit. 2019-04-18]. Dostupné z: https://www.oreilly.com/library/view/introduction-to-machine/9781449369880/ch01.html

[2] KÜRFURSTOVÁ, Jana: Strojové učení kouzla zbavené [online]. 2018, [cit. 2019-04-18]. Dostupné z: https://medium.com/edtech-kisk/strojov%C3%A9-u%C4%8Den%C3%AD-kouzla-zbaven%C3%A9-e066d79ebe51

[3] BAJAJ, Prateek: Reinforcement learning [online]. 2018, [cit. 2019-04-18]. Dostupné z: https://www.geeksforgeeks.org/what-is-reinforcement-learning/

[4] KLOUDA, Karel: Rozhodovací stromy [online]. 2018, [cit. 2019-04-18]. Dostupné z: https://courses.fit.cvut.cz/BI-VZD/lectures/files/BI-VZD-02-cs-handout.pdf

[5] BROWNLEE, Jason: Supervised and Unsupervised Machine Learning Algorithms [online]. 2016, [cit. 2019-04-18]. Dostupné z: https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/

[6] KLOUDA, Karel: Shlukování, hierarchická a algoritmus k-means [online]. 2018, [cit. 2019-04-18]. Dostupné z: https://courses.fit.cvut.cz/BI-VZD/lectures/files/BI-VZD-04-cs-handout.pdf

[7] RODRIGUEZZ, Jesus: Understanding Semi-supervised Learning [online]. 2017, [cit. 2019-05-14]. Dostupné z: https://medium.com/Žjrodthoughts/understanding-semi-supervised-learning-a6437c070c87

[8] PARSIAN, Mahmoud: *Data Algorithms: Recipes for Scaling Up with Hadoop and Spark*. O'Reilly Media, Inc., 2015, ISBN 978-1491906187,

[cit. 2019-04-22]. Dostupné z: https://www.oreilly.com/library/view/data-algorithms/9781491906170/ch13.html

[9] KARÁSEK, Jan: Citlivost metod pro měření podobnosti kvantitativních proměnných [online]. 2012, [cit. 2019-04-18]. Dostupné z: http://access.feld.cvut.cz/view.php?cisloclanku=2012090003

[10] KLOUDA, Karel: Metoda nejbližších sousedů, křížová validace [online]. 2018, [cit. 2019-04-18]. Dostupné z: https://courses.fit.cvut.cz/BI-VZD/lectures/files/BI-VZD-05-cs-handout.pdf

[11] SHEETY, Badreesh: Curse of Dimensionality [online]. 2019, [cit. 2019-04-18]. Dostupné z: https://towardsdatascience.com/curse-of-dimensionality-2092410f3d27

[12] KDNUGGETS: Must-Know: What is the curse of dimensionality? [online]. 2017, [cit. 2019-04-18]. Dostupné z: https://www.kdnuggets.com/2017/04/must-know-curse-dimensionality.html

[13] DEEPAI: Manifold Hypothesis [online]. 2017, [cit. 2019-04-18]. Dostupné z: https://deepai.org/machine-learning-glossary-and-terms/manifold-hypothesis

[14] KONEČNÝ, Jan: KMI/ZZD – Získávání znalostí z dat: Redukce dimenze dat [online]. 2015, [cit. 2019-04-18]. Dostupné z: http://phoenix.inf.upol.cz/~konecnja/vyuka/2015S/ZZDslidy/p10.pdf

[15] RAY, Sunil: Beginners Guide To Learn Dimension Reduction Techniques [online]. 2015, [cit. 2019-04-18]. Dostupné z: https://www.analyticsvidhya.com/blog/2015/07/dimension-reduction-methods/

[16] ELITEDATASCIENCE: Dimensionality Reduction Algorithms: Strengths and Weaknesses [online]. 2017, [cit. 2019-04-18]. Dostupné z: https://elitedatascience.com/dimensionality-reduction-algorithms

[17] WANG, Xueyi: A Fast Exact k-Nearest Neighbors Algorithm for High Dimensional Search Using k-Means Clustering and Triangle Inequality. *Proceedings of ... International Joint Conference on Neural Networks / co-sponsored by Japanese Neural Network Society (JNNS) ... [et al.]. International Joint Conference on Neural Networks*, ročník 43, 2012: s. 2351–2358, [cit. 2019-04-18]. Dostupné z: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3255306/

[18] ZHANG, Yan-Ming; HUANG, Kaizhu; GENG, Guanggang; LIU, Cheng-Lin: Fast kNN Graph Construction with Locality Sensitive Hashing. In *Machine*

Learning and Knowledge Discovery in Databases, editace BLOCKEEL, Hendrik; KERSTING, Kristian; NIJSSEN, Siegfried; ŽELEZNÝ, Filip, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ISBN 978-3-642-40991-2, s. 660–674, [cit. 2019-04-18]. Dostupné z: https://link.springer.com/content/pdf/10.1007%2F978-3-642-40991-2_42.pdf

[19] SIERANOJA, Sami: Fast nearest neighbor searches in high dimensions [online]. 2017, [cit. 2019-04-28]. Dostupné z: http://cs.uef.fi/pages/franti/cluster/knng_lecture_6G.pdf

[20] NI, Yun; CHU, Kelvin; BRADLEY, Joseph: Detecting Abuse at Scale: Locality Sensitive Hashing at Uber Engineering [online]. 2017, [cit. 2019-05-10]. Dostupné z: https://eng.uber.com/lsh/

[21] MARTÍNEK, Ladislav: Evaluace algoritmů lokálně senzitivního hashování (LSH) v doporučovacích systémech. 2019, [cit. 2019-04-22]. Dostupné z: https://dspace.cvut.cz/handle/10467/76825

[22] SELVALUXMIY, S; NANDA KUMARA, Titus; RAGEL, Roshan: Accelerating k-NN Classification Algorithm Using Graphics Processing Units. 12 2016, [cit. 2019-04-19]. Dostupné z: https://www.researchgate.net/publication/313299210_Accelerating_k-NN_Classification_Algorithm_Using_Graphics_Processing_Units

[23] GARCIA, Vincent; DEBREUVE, Eric; NIELSEN, Frank; BARLAUD, Michel: K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. 2010, s. 3757–3760, [cit. 2019-04-22]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.172.2896Źrep=rep1Źtype=pdf

[24] GARCIA, Vincent: kNN-CUDA [online]. [cit. 2019-04-22]. Dostupné z: https://github.com/vincentfpgarcia/kNN-CUDA

[25] NVIDIA: CUDA C Programming Guide [online]. 2019, [cit. 2019-04-22]. Dostupné z: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html}kernels

[26] LUN, Aaron: BiocNeighbors [online]. [cit. 2019-04-22]. Dostupné z: https://bioconductor.org/packages/devel/bioc/html/BiocNeighbors.html

[27] R-PROJECT: R projekt pro statistické výpočty v České republice. Co je R? [online]. [cit. 2019-05-14]. Dostupné z: http://www.r-project.cz/about.html

[28] NVIDIA: https://developer.nvidia.com/cublas [online]. 2017, [cit. 2019-04-22]. Dostupné z: https://developer.nvidia.com/cublas

[29] David: Multithreading: What is the point of more threads than cores? [online]. 2010, [cit. 2019-05-15]. Dostupné z: https://stackoverflow.com/a/3126400

[30] SCALEUP TECHNOLOGIES: GPU vs CPU Computing: What to choose? [online]. 2018, [cit. 2019-05-14]. Dostupné z: https://medium.com/altumea/gpu-vs-cpu-computing-what-to-choose-a9788a2370c4

PŘÍLOHA **A**

Seznam použitých zkratek

BLAS Basic Linear Algebra Subprograms

CPU central processing unit

cuBLAS CUDA Basic Linear Algebra Subprograms

CUDA compute undefined device architecture

GPGPU general purpose on GPU

GPU graphics processing unit

kMkNN k-means for k-nearest neighbors

kNN k-nearest neighbors

LSH locality sensitive hashing

ML machine learning

OS operating system

PŘÍLOHA **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementací algoritmů
  - `datasets` ..... zdrojové kódy pro generování datasetů
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `graphs` ..... grafy použité v práci
  - `images` ..... obrázky použité v práci
- `text` ..... text práce
  - `BP_Dvorak_Antonin_2019.pdf` ..... text práce ve formátu PDF
  - `BP_Dvorak_Antonin_2019_zadani.pdf` .... zadání práce ve formátu PDF