Raspodijeljena obrada velikih geoprostornih podataka na grafičkim procesorima

Radelić, Nikola

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:225560

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-19**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 630

RASPODIJELJENA OBRADA VELIKIH GEOPROSTORNIH PODATAKA NA GRAFIČKIM PROCESORIMA

Nikola Radelić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 630

RASPODIJELJENA OBRADA VELIKIH GEOPROSTORNIH PODATAKA NA GRAFIČKIM PROCESORIMA

Nikola Radelić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 630

Pristupnik: **Nikola Radelić (0036509643)**
Studij: Računarstvo
Profil: Računarska znanost
Mentor: prof. dr. sc. Krešimir Pripužić

Zadatak: **Raspodijeljena obrada velikih geoprostornih podataka na grafičkim procesorima**

Opis zadatka:

Akcelerator Rapids za platformu Apache Spark omogućava raspodijeljenu obradu velikih podataka na grafičkim procesorima koja je znatno učinkovitija u odnosu na središnje procesore zbog masivnog paralelizma operacija na grafičkim procesorima kojima se podaci mogu obrađivati nekoliko redova veličine brže nego na središnjim procesorima. U teoretskom dijelu ovog rada treba proučiti i predstaviti akcelerator Rapids za platformu Apache Spark te opisati način na koji se pokretanjem Sparkove aplikacije pisane u programskom jeziku Java mogu obrađivati veliki podaci raspodijeljeno na grafičkim procesorima korištenjem ovog akceleratora. Dodatno, u teoretskom dijelu ovog rada treba se fokusirati na obradu velikih geoprostornih podataka te proučiti i predstaviti mogućnosti programske knjižnice libcuspatial za programski jezik C++ jer ju Sparkove aplikacije za obradu velikih geoprostornih podataka pisane u programskom jeziku Java moraju pozivati korištenjem sučelja Java Native Interface (JNI). U praktičnom dijelu ovog rada treba osmisliti i izvesti u programskom jeziku Java programsko rješenje za raspodijeljenu obradu velikih geoprostornih podataka na grafičkim procesorima korištenjem akceleratora Rapids i knjižnice libcuspatial. Ostvareno rješenje treba ispitati u računalnom grozdu na odabranom studijskom slučaju velikih geoprostornih podataka.

Rok za predaju rada: 28. lipnja 2024.

Zahvaljujem se mentoru prof. dr. sc. Krešimiru Pripužiću na pomoći prilikom izrade ovog rada i tijekom cijelog mentorstva. Također zahvaljujem se obitelji i prijateljima za podršku, a pogotovo curi Tei koja je bila izuzetna podrška.

Sadržaj

1. Uvod **3**

2. Teoretski okvir **5**

2.1. Obrada velikih podataka 5

2.1.1. Definicija i značaj velikih podataka 5

2.1.2. Izazovi u obradi velikih podataka 6

2.2. Grafički procesori (GPU) 7

2.2.1. Osnovne karakteristike GPU-a 7

2.2.2. Prednosti GPU-a u odnosu na CPU 8

2.3. Apache Spark 10

2.3.1. Arhitektura Spark-a 11

2.3.2. Resilient Distributed Dataset (RDD) 15

2.3.3. SparkSQL 16

3. Akcelerator Rapids za Apache Spark **17**

3.1. Uvod u Rapids 17

3.1.1. Komponente Rapidsa 18

3.1.2. Prednosti korištenja Rapidsa 19

3.2. Integracija Rapidsa s Apache Sparkom 19

3.2.1. Tehnički detalji integracije 19

3.2.2. Performanse i efikasnost 20

3.2.3. Konfiguriranje RAPIDS-a za Spark 21

4. Obrada velikih geoprostornih podataka **23**

4.1. Geoprostorni podaci 23

4.2. Knjižnica Libcuspatial 24

4.2.1. Osnovne karakteristike knjižnice cuSpatial 24
4.2.2. Funkcionalnosti i primjene 25
4.2.3. Java Native Interface (JNI) 26
4.3. Integracija knjižnice cuSpatial s Java kodom 27

5. Praktični dio 28
5.1. Opis studijskog slučaja 28
5.1.1. Geoprostorni podatci i očekivani rezultati 28
5.1.2. Specifikacije računala za izvođenje programskog primjera 29
5.2. Razvoj programskog rješenja 30
5.3. Instalacija i pokretanje primjera 35
5.3.1. Instalacija potrebnih alata 35
5.3.2. Postavljanje i pokretanje primjera 38
5.4. Evaluacija rješenja 40

6. Zaključak 45

Literatura 47

Sažetak 49

Abstract 50

A: The Code 51

1. Uvod

U današnjem svijetu koji ovisi o digitalizaciji i potrebi za obradu sve većih količina podataka potrebno je razvijati sustave koji mogu odrađivati te zadatke efikasno i brzo. Kako ti zadatci postaju sve veći i teži, potrebno je skalirati računalne sustave kako bi se mogli nositi s tom količinom rada. Postoje dvije opcije, vertikalno i horizontalno skaliranje računalnih resursa.

Vertikalno skaliranje, poznato i kao skaliranje gore (engl. scaling up), podrazumijeva dodavanje resursa pojedinačnom poslužitelju ili računalnom sustavu. Horizontalno skaliranje, poznato i kao skaliranje van (engl. scaling out), uključuje dodavanje više poslužitelja ili čvorova u sustav. U okviru velikih podataka, fizičkih ograničenja, jedne točke neuspjeha i visokih troškova češće se koristi horizontalno skaliranje. Problem takvog pristupa je kompleksnost upravljanja, konzistentnost podataka i modifikacije aplikacija. Navedene probleme rješavamo s sustavima kao Apache Spark[1] koji rješavaju probleme raspodijeljenih sustava.

Platforma Apache Spark[1] je alat za obradu velikih podataka na raspodijeljenim sustavima, razvijen je 2009. godine kao istraživački projekt na Sveučilištu Berkeley te se brzo razvio u vodeću platformu otvorenog koda za obradu velikih podataka. Omogućuje brzu obradu podataka koristeći obradu izravno na radnoj memoriji bez pisanja i čitanja podataka s trajne memorije što značajno ubrzava obradu. Dizajniran je za rad na računalnim grozdovima i podržava visoku skalabilnost. Jednostavan je za korištenje i podržava programiranje u velikom broju jezika kao Java, Scala, Python i R. Otporan je na greške i ima ugrađene mehanizme za otkrivanje i oporavak od softverskih i hardverskih kvarova. Također podržava obradu na grafičkim karticama, koje jako ubrzavaju brzinu obrade u slučaju jako paraleliziranih programa.

NVIDIA Rapids[2] je skup knjižnica otvorenog koda koje omogućuju obradu podataka na grafičkim karticama. Primarno je dizajniran za obradu velike količine podataka koji se mogu razdijeliti na male količine za laganu i brzu obradu. Prvi put je predstavljena 2018. godine, a 2020. godine je predstavljen Spark Rapids koji omogućuje platformi Apache Spark da koristi akcelerator Rapids u obradi velike količine podataka. Rapids akcelerator ubrzava programe bez potrebe za velikim mijenjanjem postojećeg koda, samo se treba dodati odgovarajuća knjižnica. Neke od bitnih knjižnica Rapids su cuDF, cuSpatial, cuML.

U ovom radu biti će opisana platforma Apache Spark i kako je moguće ubrzati rad korištenjem akceleratora Rapids. Također će se opisati korištenje knjižnice cuSpatial[3] te kako se ona može koristiti za obradu geoprostornih podataka.

2. Teoretski okvir

2.1. Obrada velikih podataka

2.1.1. Definicija i značaj velikih podataka

Veliki podaci (engl. Big Data) odnose se na skupove podataka koji su tako opsežni i složeni da ih je teško obraditi korištenjem tradicionalnih metoda i alata za obradu podataka. Veliki podaci, u širem smislu, obuhvaćaju podatke iz različitih izvora i formata, uključujući strukturirane, polustrukturirane i nestrukturirane podatke. Definicija velikih podataka često se oslanja na konceptu "4 V"[4]:

- Volumen (Volume): Odnosi se na količinu prikupljenih podataka. Analitičar mora odrediti koje podatke i koliko njih treba prikupiti za određenu svrhu. Zamislite mogućnosti na društvenoj mreži gdje ljudi pišu ažuriranja, lajkaju fotografije, ocjenjuju poslovanja, gledaju videozapise, pretražuju nove stavke i komuniciraju s gotovo svime što vide na ekranu. Svaka od tih interakcija generira podatke o toj osobi koje se mogu koristiti u algoritmima.

- Brzina (Velocity): Brzina u velikim podacima odnosi se na to koliko brzo se podaci mogu generirati, prikupljati i analizirati. Veliki podaci ne moraju uvijek biti odmah upotrebljivi, ali u nekim područjima velika je prednost dobivati informacije u stvarnom vremenu i djelovati u skladu s tim. U drugim poslovima, važniji je trend podataka kroz vrijeme za pomoć u predviđanjima ili rješavanju trajnih problema.

- Raznolikost (Variety): Raznolikost se odnosi na broj referentnih točaka koje se koriste za prikupljanje podataka. Ako se podaci prikupljaju iz jednog izvora, informacije mogu biti pristrane. Neće predstavljati široku populaciju ili opći trend. U nekim slučajevima, poput brzine, to je u redu. Na primjer, usluga mikročipiranja

kućnih ljubimaca može željeti ciljati podatke iz lokalne društvene mreže. Filmska kompanija, s druge strane, može željeti ciljati nekoliko društvenih mreža i ljude različitih dobnih skupina. Tako bi im trebalo više referentnih točaka za donošenje poslovnih odluka.

- Veracity (Istinitost): Odnosi se na pouzdanost podataka. Analitičar želi osigurati da su podaci valjani i dolaze iz pouzdanog izvora. To se određuje prema izvoru podataka i načinu prikupljanja. Podaci prikupljeni iz izvornih stranica, a ne od trećih strana, nužni su za pouzdane rezultate. Također, mjere testiranja moraju biti pravilno dizajnirane kako bi se osiguralo da podaci donose željene informacije, a ne suvišne.

2.1.2. Izazovi u obradi velikih podataka

Obrada velikih podataka donosi razne izazove. Tradicionalni sustavi skladištenja podataka često nisu dovoljno skalabilni za potrebe velikih podataka. Novi pristupi kao što su raspodijeljeni datotečni sustavi, poput Hadoop Distributed File System (HDFS) i NoSQL baze podataka poput Cassandre i MongoDB-a, omogućuju učinkovitiju pohranu velikih količina podataka. Kompleksnost analize velikih podataka zahtijeva napredne algoritme i moćne računalne resurse. Tehnike poput strojnog učenja, dubokog učenja i analitike u realnom vremenu koriste se za analizu velikih podataka. Kombiniranje podataka iz različitih izvora i formata može biti tehnički zahtjevno i složeno. Potrebne su napredne metode za ekstrakciju, transformaciju i učitavanje podataka (ETL) kako bi se osiguralo da su podaci konzistentni i integrirani. Izazovi u sigurnosti i zaštiti privatnosti postaju sve važniji, posebno zbog osjetljivosti mnogih podataka. Potrebne su napredne sigurnosne mjere za zaštitu podataka od neovlaštenog pristupa i gubitka podataka. Osiguranje točnosti, konzistentnosti i cjelovitosti podataka također je ključno za pouzdane analize jer loša kvaliteta podataka može dovesti do netočnih zaključaka i pogrešnih odluka.

2.2. Grafički procesori (GPU)

Grafički procesori (GPU) su specijalizirani hardverski uređaji prvotno dizajnirani za ubrzavanje grafičkih operacija, posebno u renderiranju 3D grafike i obradi videozapisa. Međutim, njihova arhitektura koja omogućava masivnu paralelizaciju učinila ih je izuzetno korisnima za opću obradu podataka, posebno za zadatke koji se mogu paralelizirati. GPU-ovi su postali ključni alati u raznolikim znanstvenim i industrijskim aplikacijama, uključujući strojno učenje, simulacije, obradu slika i velike podatke.

Slika 2.1. Grafička kartica [5]

2.2.1. Osnovne karakteristike GPU-a

GPU-ovi posjeduju nekoliko bitnih karakteristika koje ih čine pogodnima za paralelnu obradu podataka:

- **Visok broj jezgri**: GPU-ovi sadrže tisuće manjih jezgri koje mogu simultano obrađivati podatke. Na primjer, NVIDIA Tesla V100 GPU[6] ima 5120 CUDA jezgri, što mu omogućava izvođenje velikog broja operacija paralelno. Nasuprot tome, tipični CPU-ovi imaju između 4 i 64 jezgre koje su optimizirane za serijsku obradu zadataka.

- **Paralelna obrada**: Konstrukcija GPU-a omogućava istovremeno izvođenje mnogih operacija. Ova sposobnost masovne paralelizacije čini GPU-ove idealnim za

zadatke kao što su matriksne operacije, simulacije i obrade slika, gdje se isti skup operacija može primijeniti na mnogo podataka paralelno.

- **Velika propusnost memorije**: GPU-ovi su dizajnirani za brzo premještanje velikih količina podataka između memorije i procesorskih jezgri. NVIDIA Tesla V100, na primjer, ima memorijsku propusnost od 900 GB/s, što omogućava bržu obradu podataka u usporedbi s tipičnim CPU-ovima.

2.2.2. Prednosti GPU-a u odnosu na CPU

Grafički procesori (GPU) imaju nekoliko ključnih prednosti u usporedbi s centralnim procesorima (CPU). Prvo, GPU-ovi mogu izvršavati određene zadatke mnogo brže od CPU-ova zahvaljujući svojoj sposobnosti paralelne obrade. Naime, u strojnom učenju, treniranje dubokih neuronskih mreža može biti višestruko brže na GPU-ovima nego na CPU-ovima. To je moguće zbog činjenice da su GPU-ovi dizajnirani za obradu velikih blokova podataka istovremeno, omogućavajući osjetno ubrzanje kod operacija koje se mogu paralelizirati.

Druga važna prednost GPU-ova je njihova energetska efikasnost. Za određene aplikacije, GPU-ovi mogu biti energetski efikasniji od CPU-ova jer mogu obraditi više podataka po jedinici energije. Ova efikasnost je esencijalna u velikim podatkovnim centrima gdje se energetska efikasnost direktno pretvara u uštede troškova. Dizajn GPU-ova omogućava im da budu izuzetno efikasni u izvršavanju zadataka koji zahtijevaju masivnu paralelizaciju.

GPU-ovi nude i skalabilnost kao značajnu prednost. Mogućnost obrade podataka na više uređaja čini ih idealnima za efikasnu manipulaciju velikim podacima u raspodijeljenim sustavima. Kombiniranjem više GPU-ova u grozdovima, moguće je ostvariti još veće performanse i kapacitet obrade. Ova karakteristika omogućuje raspodijeljenim sustavima da se razvijaju u skladu s rastom podataka i potreba za njihovom obradom, osiguravajući prilagodljivost i skalabilnost potrebnu za suvremene aplikacije.

GPU-ovi su specijalizirani za zadatke koji uključuju masivnu paralelizaciju, dok su CPU-ovi dizajnirani za opću obradu i sposobni su efikasno upravljati nizom različitih zadataka. Zahvaljujući ovoj specijalizaciji, GPU-ovi su izuzetno efikasni u specifičnim

primjenama kao što su strojno učenje, simulacije i obrada slika. S druge strane, CPU-ovi nude fleksibilnost za širok spektar upotreba, ali često ne mogu doseći iste performanse kao GPU-ovi u spomenutim zadacima.

Razvoj softvera za GPU-ove dodatno unapređuje njihove performanse. Alati poput NVIDIA CUDA i OpenCL omogućavaju programerima da maksimalno iskoriste snagu GPU-ova za paralelnu obradu podataka. Ovi alati pružaju API-je i razvojna okruženja koja olakšavaju pisanje i optimizaciju koda za GPU-ove, čime se povećava njihova efikasnost i primjenjivost u različitim industrijama.

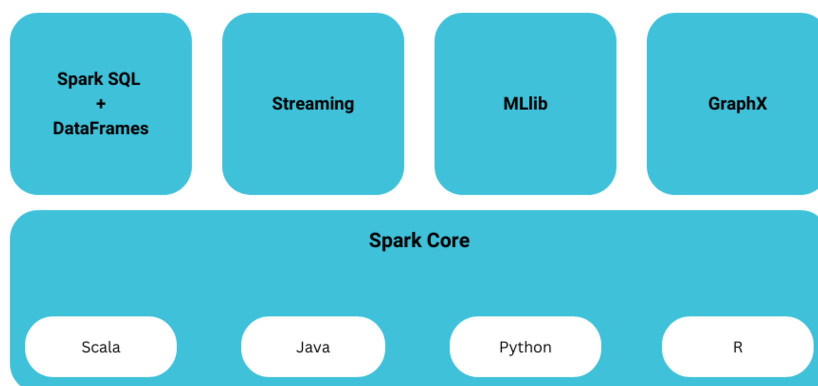
Integracija GPU-ova s postojećim računalnim sustavima također je važna prednost. GPU-ovi se mogu dodati postojećim sustavima, omogućavajući organizacijama da povećaju svoje kapacitete bez potrebe za potpunom zamjenom infrastrukture. Ova prilagodljivost može biti ključna za postizanje visokih performansi u kombinaciji s postojećim CPU resursima, omogućavajući organizacijama da iskoriste najbolje od oba svijeta.

GPU-ovi se koriste u raznim aplikacijama koje zahtijevaju visoku računalnu snagu. U području strojnog učenja, GPU-ovi se koriste za treniranje dubokih neuronskih mreža, što uključuje obradu milijuna podataka u vrlo kratkom vremenu. U znanstvenim istraživanjima, GPU-ovi se koriste za simulacije fizičkih procesa, kao što su dinamika fluida, molekularno modeliranje i klimatske simulacije. U industriji zabave, GPU-ovi neophodni su za renderiranje 3D grafike u realnom vremenu, omogućavajući stvaranje visokokvalitetnih vizualnih efekata u filmovima i video igrama. U medicini, GPU-ovi se koriste za analizu medicinskih slika, uključujući računalnu tomografiju (CT), magnetsku rezonancu (MRI) i ultrazvuk, omogućavajući brzu i preciznu dijagnostiku.

U zaključku, GPU-ovi predstavljaju ključnu tehnologiju za obradu velikih podataka, omogućavajući brže, efikasnije i skalabilnije rješenje u usporedbi s tradicionalnim CPU-ovima. Njihova sposobnost paralelne obrade, visoka propusnost memorije i specijalizirana arhitektura čine ih idealnim za zadatke koji zahtijevaju visoke performanse i obradu velikih količina podataka.

2.3. Apache Spark

Apache Spark je platforma otvorenog koda napravljena za raspodijeljene sustave je fokusirana na obradu velike količine podataka. Baziran je na platformi Hadoop MapReduce koja je napravljena za istu funkciju kao Spark. Omogućila je pisanje programa koji se mogu paralelno izvoditi na grozdu računala. Podatci na MapReduce se obrađuju u koracima, svaki korak čita podatke s trajne memorije, izvodi operaciju i rezultate zapisuje natrag na trajnu memoriju. Te ulazno/izlazne operacije u svakom koraku značajno usporavaju rad platforme MapReduce. Spark uvodi inovaciju u MapReduce, a to je zadržavanje podataka u radnoj memoriji za međukorake. Na početku programa čitaju se podaci, a na kraju se zapisuju rezultati, što značajno ubrzava rad platforme Spark i dramatično povećava njezinu učinkovitost. U nekim slučajevima, to ubrzanje doseže čak i do 100 puta.

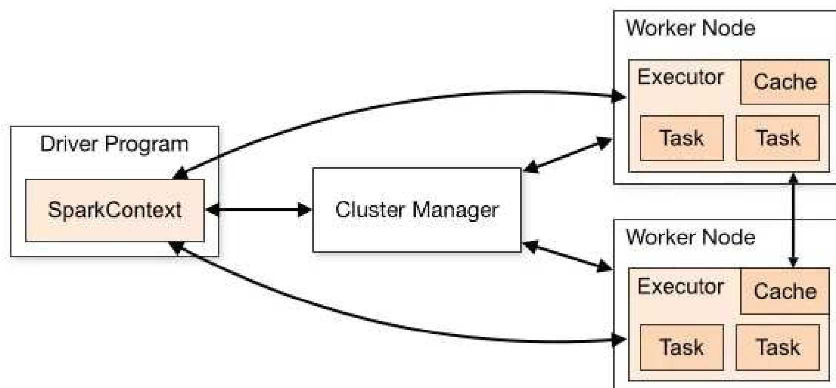


Slika 2.2. Spark Core

Na slici 2.2. vidljiva je organizacija Spark komponenti. Spark Core pruža osnovne funkcionalnosti potrebne za rad s raspodijeljenim podacima, zadužen je za raspodjelu podataka i poslova po svim procesima u grozdu. Baziran je na programskom jeziku Scala, ali podržani jezici s rad s Spark-om su također i Java, Python i R.

2.3.1. Arhitektura Spark-a

Sparkova arhitektura[7] temelji se na konceptu master-slave modela, gdje master upravlja radom cijelog sustava, dok sluga izvršava zadane zadatke.



Slika 2.3. Spark Context [1]

Organizacija Sparka vidljiva je na slici 2.3. i zadrži sljedeće komponente:

- Driver Program: Pokreće glavnu funkciju aplikacije i definira RDD-ove te akcije i transformacije nad njima. Driver program također upravlja rasporedom zadataka na različite izvršitelje. On sve operacije transformira u plan izvedbe zvan DAG (engl. directed acyclic graph).
- Cluster Manager: Upravlja resursima unutar grozda. Spark može raditi s različitim vrstama upravitelja klastera, uključujući Standalone, Apache Mesos, Hadoop YARN i Kubernetes.
- Workers: Izvršitelji koji rade na radnim čvorovima. Svaki worker sadrži više izvršitelja (engl. executors) koji izvode zadatke.
- Executors: Procesi koji rade na radnim čvorovima i izvršavaju zadatke. Svaki executor čuva podatke i vrši operacije nad njima.
- Task: Temeljna jedinica rada koju executor izvršava. Taskovi se stvaraju na temelju operacija definiranih u driver programu.

Za pokretanje Spark aplikacije koristimo naredbu `spark-submit` [1]. Ta metoda nam omogućuje pokretanje Spark aplikacija lokalno ili na grozdu. Sljedeći primjer pokazuje primjer pokretanja Sparkove aplikacije.

```
./bin/spark-submit \
    --class <main-class> \
     --master <master-url> \
     --deploy-mode <deploy-mode> \
    --conf <key>=<value> \
    ... # other options
    <application-jar> \
    [application-arguments]
```

Naredba `spark-submit` osigurava da se pokrene platforma Spark lokalno ili na grozdu i da su sve ovisnosti i postavke postavljene. Neke od postavki su postavljanje glavne klase Java ili Scala aplikacije, postavljanje URL-a za lokalno ili pokretanje na grozdu, razne konfiguracijske postavke, postavljanje ovisnosti i knjižnica itd. Moramo koristiti `spark-submit` kada pokrećemo aplikaciju na grozdu ili kada koristimo razne ovisnosti i treba nam naprednije postavke nad pokretanjem.

U Sparku 3 koristimo `SparkSession`[8] kao glavnu točku ulaska u Spark Aplikaciju.

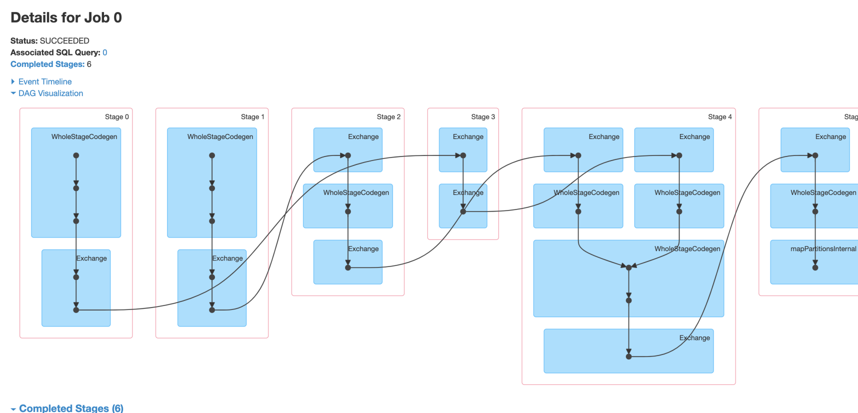
```python
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

Ovdje vidimo primjer inicijalizacije Sparka u kodu Python [1]. Moguće je još postaviti puno parametara kao broj jezgri, url grozda gdje će se program izvoditi, broj particija, količina memorije i razno. Može se koristiti u kombinaciji s `spark-submit` kada naš Python kod ovisi o nekim vanjskim datotekama i mora se izvoditi na grozdu, ali se može koristiti i samostalno ako pokrećemo aplikaciju samostalno i lokalno.

Izvođenje Spark aplikacije:

- Pokretanje Spark aplikacije iz naredbenog retka
- Spark stvara logički plan izvođenja u obliku DAG (engl. Directed Acyclic Graph)
- Pretvaranje logičkog plana u fizički plan koji ovisi o grozdu
- Generiranje koda koji će se izvoditi na svakom grozdu
- Izvođenje zadataka na svakom grozdu

DAG[9] se koristi kao logički plan za optimizaciju izvršavanja operacija na grozdu. Bitno je napomenuti da je DAG acikličan i ne sadrži petlje, što doprinosi optimizaciji izvođenja analizom akcija i transformacija radi određivanja najefikasnijeg redoslijeda operacija. Također može neke operacije spojiti u jednu kako bi daljnje optimirao izvođenje [9]. On ne ovisi o tipu grozda i tipu podataka, samo o korisničkom kodu. Na slici 2.4. možemo vidjeti složeniji primjer DAG-a, gdje se ističe optimizacija u prvoj i drugoj fazi te kako se one zapravo mogu izvoditi u paraleli.



Slika 2.4. Primjer DAG-a [10]

Fizički plan izvođenja predstavlja prave korake koji će se izvoditi na svakom računalu u grozdu, on ovisi o tipu grozda, tipu podataka, dostupnim sredstvima itd. U sljedećem isječku možemo vidjeti primjer dijela fizičkog plana.

```
   == Physical Plan ==
GpuColumnarToRow (17)
+- Execute GpuInsertIntoHadoopFsRelationCommand (16)
   +- AdaptiveSparkPlan (15)
      +- == Final Plan ==
         GpuSort (9)
         +- GpuShuffleCoalesce (8)
            +- GpuCustomShuffleReader (7)
               +- ShuffleQueryStage (6), Statistics(sizeInBytes=763.5 MiB,
                   rowCount=2.00E+7)
                  +- GpuColumnarExchange (5)
                     +- GpuProject (4)
                        +- GpuCoalesceBatches (3)
                           +- GpuFilter (2)
                              +- GpuScan parquet (1)
      +- == Initial Plan ==
         Sort (14)
         +- Exchange (13)
            +- Project (12)
               +- Filter (11)
                  +- Scan parquet (10)
```

2.3.2. Resilient Distributed Dataset (RDD)

RDD[11] je osnovna apstrakcija u Spark Core-u koja predstavlja raspodijeljeni skup podataka nepromjenjive prirode. RDD-ovi podržavaju dvije vrste operacija: transformacije i akcije. U ovom primjeru vidimo kako se RDD može stvoriti iz postojećih datoteka ili već postojeće kolekcije.

```
# Kreiranje RDD-a iz vanjskog izvora
lines = sc.textFile("data.txt")

# Kreiranje RDD-a iz postojece kolekcije
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

RDD-ovi podržavaju dva tipa operacija, to su transformacije i akcije. Transformacije su operacije koje uzimaju RDD kao ulaz i vraćaju novi RDD kao rezultat. One su lijeno evaluirane, što znači da se stvarno izvršavaju tek kada se pozove neka akcija. Akcije su operacije koje vraćaju vrijednost glavnom programu ili spremaju podatke u vanjski sustav pohrane. Akcije pokreću stvarno izvršenje svih prethodnih transformacija.

```
# Primjer transformacije
filteredLines = lines.filter(lambda s: "error" in s)

# Primjer akcije
wordCounts.saveAsTextFile("output.txt")
```

2.3.3. SparkSQL

Spark SQL[12] je komponenta Apache Sparka koja omogućuje rad s ustruktuiranim podacima koristeći SQL upite. Spark SQL koristi DataFrame API koji omogućuje manipulaciju podacima na sličan način kao SQL.

DataFrame je raspodijeljeni skup podataka organiziranih u nazvane stupce, konceptualno sličan tablici u relacijskoj bazi podataka. DataFrame-ovi se mogu kreirati iz različitih izvora podataka poput CSV datoteka, baza podataka, ili iz RDD-ova.

```
# Kreiranje DataFrame-a iz CSV datoteke
df = spark.read.format("csv").option("header", "true").load("employees.csv")

# Kreiranje DataFrame-a iz RDD-a
employeeRDD = sc.parallelize(employeeData)
employeeDF = spark.createDataFrame(employeeRDD, Employee)
```

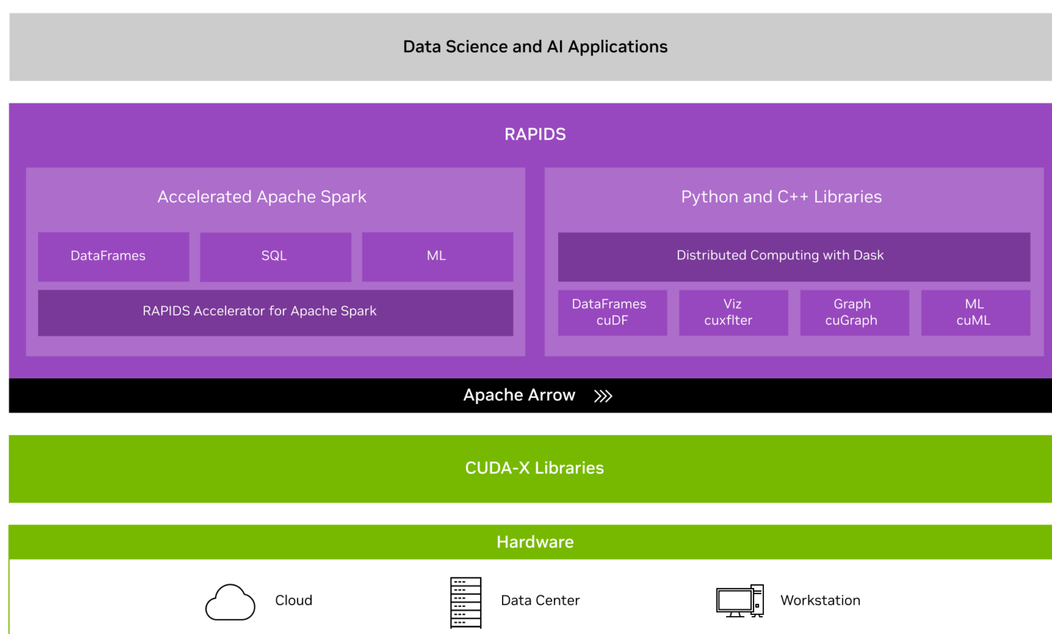
Nakon učitavanja, nad DataFrame-om se mogu izvršavati SQL upiti i druge operacije.

```
# Registracija DataFrame-a kao SQL tablice
df.createOrReplaceTempView("employees")

# Izvrsavanje SQL upita
highSalaryDF = spark.sql("SELECT * FROM employees WHERE salary > 50000")

# Primjena operacija nad DataFrame-om
departmentSalary = df.groupBy("department").agg({"salary": "avg", "salary":
    "max"})
```

3. Akcelerator Rapids za Apache Spark

3.1. Uvod u Rapids

Rapids je skup softverskih knjižnica i alata dizajniranih za ubrzanje obrade podataka korištenjem grafičkih procesora (GPU-ova)[2]. Izgrađen na temeljima platforme CUDA koju razvija NVIDIA, Rapids omogućava korištenje moćnih mogućnosti paralelne obrade GPU-ova za ubrzanje različitih zadataka obrade podataka, uključujući manipulaciju podacima, strojno učenje i analizu grafova. Rapids se ističe u kontekstu velikih podataka, gdje konvencionalni CPU-ovi često nisu dovoljno brzi za obradu ogromnih količina podataka u prihvatljivim vremenskim okvirima.



Slika 3.1. Rapids dijagram [2]

Slika 3.1. pokazuje dva različita načina korištenja Rapids akceleratora. Moguće je ubrzati Spark ili Dask [13]. Dask je Python platforma za obradu velikih podataka i namijenjena je znanstvenoj zajednici za obradu podataka. Prednost Daska je jednostavnije korištenje, korištenje Numpy i Pandas API-ja i potpuno je napisan u Pythonu. Također, Dask može efikasno raditi na jednom računalu, dok Spark obično zahtijeva grozd za optimalne performanse. Spark je, s druge strane, bolji za složene analitike i obradu velikih podataka na raspodijeljenim sustavima.

3.1.1. Komponente Rapidsa

Rapids se sastoji od nekoliko ključnih komponenti, svaka sa specifičnim funkcijama koje zajedno omogućuju ubrzanje različitih aspekata obrade podataka:

- cuDF: Ova knjižnica pruža funkcionalnosti za rad s tabličnim podacima na GPU-u, slično kao knjižnica Pandas za Python. cuDF omogućava izvođenje operacija poput filtriranja, grupiranja i agregacije podataka brzinom koja je višestruko veća od tradicionalnih CPU-based rješenja.

- cuML: cuML je alatni set za strojno učenje optimiziran za korištenje GPU-ova. Pruža podršku za širok spektar algoritama poput regresije, klasifikacije i klasteriranja. Korištenje cuML-a omogućava znatno brže treniranje modela i izvođenje predikcija u usporedbi s CPU-ovima.

- cuGraph: Ova knjižnica je namijenjena analizi grafova i omogućava brzo izvođenje grafovskih algoritama na GPU-u. cuGraph uključuje funkcionalnosti među kojima su pretraživanja grafova, izračuna povezanih komponenti i centralnosti. Ova knjižnica je posebno korisna za aplikacije koje rade s mrežama i grafovima podataka, kao što su društvene mreže i analitika podataka o prometu.

- cuSpatial: Knjižnica cuSpatial specijalizirana je za obradu geoprostornih podataka na GPU-ovima. Pruža visoku učinkovitost u izvođenju operacija kao što su prostorno spajanje, izračun udaljenosti i analiza poligona. cuSpatial podržava različite formate geoprostornih podataka i omogućava njihovu brzu analizu, što je od velike koristi za aplikacije u geografskim informacijskim sustavima (GIS), urbanističkom planiranju, navigaciji i okolišnim istraživanjima.

3.1.2. Prednosti korištenja Rapidsa

Rapids nudi nekoliko ključnih prednosti, pri čemu je najznačajnija ubrzanje obrade podataka. Na primjer, testiranja su dokazala da Rapids može znatno smanjiti vrijeme obrade velikih podataka korištenjem GPU-ova, što rezultira ubrzanjem operacija do 50 puta u usporedbi s CPU-ovima. To omogućuje korisnicima brže donošenje odluka i dobivanje brzih uvida.

Druga važna prednost je smanjenje operativnih troškova. Brža obrada podataka može smanjiti troškove računalnih resursa, posebno u okruženjima s velikim količinama podataka. Ušteda vremena u obradi podataka rezultira manjim troškovima za računalne resurse, što je posebno važno za organizacije koje obrađuju velike količine podataka na dnevnoj bazi.

Jednostavna integracija Rapidsa s postojećim alatima i platformama također je znakovita prednost. Rapids je konstruiran za jednostavnu integraciju s platformom Apache Spark, što omogućava korisnicima da brzo implementiraju Rapids u svoje postojeće sustave za obradu podataka. Ovo omogućava korisnicima da iskoriste prednosti GPU-akcelerirane obrade podataka bez potrebe za značajnim promjenama u svojoj infrastrukturi.

3.2. Integracija Rapidsa s Apache Sparkom

3.2.1. Tehnički detalji integracije

Integracija Rapidsa s Apache Sparkom omogućava korištenje GPU-ova za ubrzanje obrade podataka unutar Spark aplikacija. Tehnički detalji uključuju podršku za Spark API-je koji mogu pozivati Rapids funkcionalnosti za manipulaciju i analizu podataka. Na primjer, cuDF API omogućuje izvođenje operacija na DataFrame objektima unutar Spark aplikacija, pružajući sličnu funkcionalnost kao Pandas DataFrame, ali s mnogo većim performansama zahvaljujući GPU-ovima.

Konfiguracija Spark grozda za korištenje GPU-ova je ključan aspekt u integraciji. To uključuje postavljanje CUDA alata i drivera na svim čvorovima u grozdu, kao i konfiguraciju Spark okruženja za prepoznavanje i korištenje GPU-ova. Osim toga, Sparkov

resursni menadžer mora biti konfiguriran za upravljanje GPU resursima, što omogućava dinamičku alokaciju GPU-ova za različite zadatke.

Integracija knjižnice cuSpatial: Da bi se cuSpatial koristio unutar Spark aplikacija, potrebno je osigurati da Spark može komunicirati s knjižnicom cuSpatial putem Java Native Interface (JNI)[14]. To uključuje pisanje JNI omota koji omogućava kodu Java da poziva funkcionalnosti knjižnice cuSpatial implementirane u C++. Ovaj omot omogućava prevođenje Java objekata u formate koje cuSpatial može razumjeti i obratno, osiguravajući efikasnu i brzu obradu geoprostornih podataka na GPU-ovima.

3.2.2. Performanse i efikasnost

Korištenjem Rapidsa u kombinaciji s Apache Sparkom postižu se značajna poboljšanja u performansama obrade podataka. Na primjer, izvođenje kompleksnih upita nad velikim tabličnim skupovima podataka može biti višestruko ubrzano, omogućavajući brže dobivanje rezultata i poboljšanje učinkovitosti poslovnih procesa. Testiranja su pokazala da Rapids može smanjiti vrijeme obrade podataka za nekoliko redova veličine, posebno za zadatke koji zahtijevaju intenzivnu računalnu snagu kao što su strojno učenje, analiza grafova i obrada geoprostornih podataka.

U stvarnim aplikacijama, ovo ubrzanje može značiti razliku između dobivanja rezultata u minutama umjesto satima ili danima,a posebno je vazno u industrijama kao što su financije, zdravstvo i telekomunikacije, gdje brza obrada podataka može imati značajan utjecaj na donošenje odluka i operativnu učinkovitost. U kontekstu geoprostornih podataka, brza analiza može omogućiti pravovremeno donošenje odluka u urbanističkom planiranju, upravljanju prirodnim resursima i odgovorima na hitne situacije.

3.2.3. Konfiguriranje RAPIDS-a za Spark

Za implementaciju RAPIDS-a u Sparku potrebno je imati:

- Nvidia grafičku karticu arhitekture Volta ili novije
- Drivere i CUDA Toolkit
- Spark 3.2.0 ili noviji
- Instalirane knjižnice koje se planiraju koristiti, npr. CuDF, cuSpatial (moguće poput Conde ili Docker-a)
- NVIDIA Spark-RAPIDS JAR datoteku koja je kompatibilna s verzijom Sparka

Rapids za Spark se postavlja dodavanjem konfiguracija u `spark-submit` ili `SparkSession`.

```
#primjer omogucavanja RAPIDS-a u PySparku
spark = SparkSession.builder \
    .appName("Spark RAPIDS Primjer") \
    .config("spark.rapids.sql.enabled","true") \
    .config("spark.plugins", "com.nvidia.spark.SQLPlugin") \
    .config("spark.executor.resource.gpu.amount", "1") \
    .config("spark.executor.resource.gpu.discoveryScript",
        "/put/do/getGpusResources.sh") \
    .config("spark.rapids.sql.concurrentGpuTasks", "2") \
    .config("spark.task.resource.gpu.amount", "1") \
    .config("spark.jars", "put/do/rapids-4-spark_2.12-23.02.0.jar")
    .getOrCreate()
```

```
#primjer omogucavanja RAPIDS-a u spark-submit
spark-submit \
    --master yarn \
    --deploy-mode cluster \
    --conf spark.plugins=com.nvidia.spark.SQLPlugin \
    --conf spark.executor.resource.gpu.amount=1 \
    --conf
        spark.executor.resource.gpu.discoveryScript=/put/do/getGpusResources.sh
        \
    --conf spark.rapids.sql.concurrentGpuTasks=2 \
    --conf spark.task.resource.gpu.amount=1 \
     --jars /put/do/rapids-4-spark_2.12-23.02.0.jar \
    my_spark_app.py
```

Vidimo kako je postavljanje RAPIDS-a jednostvno i ne zahtjeva velike promjene u samom kodu. Spark i RAPID-s mogu odlučiti koje se funkcije mogu odrađivati na GPU, a koje ne. Prilikom izvođenja aplikacije, moguće je provjeriti koje funkcije će se odrađivati na GPU pomoću dodatka `-conf spark.rapids.sql.explain=ALL` u `spark-submit`. Na taj način dobivamo listu funkcija koje se izvode i obavijest izvodi li se operacija na GPU ili ne.

4. Obrada velikih geoprostornih podataka

4.1. Geoprostorni podaci

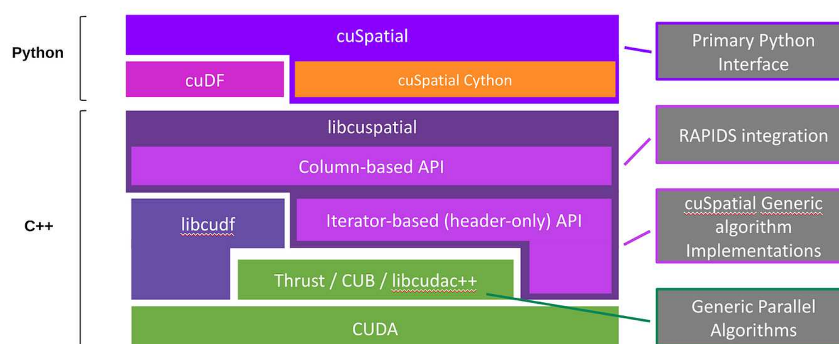
Geoprostorni podaci su informacije koje informacije koje se odnose na specifične geografske lokacije. Sadrže raznovrsne podatke poput koordinata, topografskih mapa, satelitskih snimaka i drugih prostornih atributa. Ovi podaci igraju ključnu ulogu u raznim aplikacijama, uključujući geografske informacijske sustave (GIS), urbanističko planiranje, navigaciju i okolišna istraživanja.

Geoprostorni podaci mogu se podijeliti na dvije glavne kategorije: vektorske i raster podatke. Kroz točke, linije i poligone, vektorski podaci predstavljaju geografske objekte kao što su ceste, rijeke i granice. Ovi podaci često se pohranjuju u formatima kao što su Shapefile od Esri i GeoJSON. S druge strane, raster podaci se sastoje od mreže ćelija ili piksela, gdje svaki piksel ima pridruženu vrijednost. Primjeri uključuju satelitske snimke i digitalne visinske modele, često pohranjene u formatima kao što su GeoTIFF.

Izvori geoprostornih podataka uključuju satelitske slike, GPS uređaje, zračne fotografije i terenski prikupljene podatke, čineći ih raznolikima. Na primjer, satelitske snimke omogućavaju prikupljanje podataka o zemljinoj površini u visokoj rezoluciji, dok GPS uređaji omogućavaju precizno praćenje lokacije u realnom vremenu. Ovi podaci su esencijalni za razne analize, od urbanističkog planiranja do praćenja promjena u okolišu.

4.2. Knjižnica Libcuspatial

Libcuspatial[3] je knjižnica osmišljena za ubrzavanje obrade geoprostornih podataka korištenjem GPU-ova. Njena arhitektura omogućava brzu i efikasnu analizu velikih količina geoprostornih podataka, što je presudno za aplikacije koje zahtijevaju visok stupanj prostorne analize. Libcuspatial koristi snagu GPU-ova za paralelizaciju obrade, omogućavajući izvođenje kompleksnih geoprostornih operacija u realnom vremenu.



Slika 4.1. Knjižnica Cuspatial [15]

4.2.1. Osnovne karakteristike knjižnice cuSpatial

cuSpatial pruža niz funkcionalnosti koje su ključne za obradu geoprostornih podataka. Među najvažnijim su:

- Prostorno pridruživanje: Omogućuje povezivanje geoprostornih objekata temeljem njihovih lokacija. Ova funkcionalnost je esencijalna za analize koje se bave prostornim odnosima između različitih skupova podataka. Na primjer, može se koristiti za pridruživanje točaka interesa unutar određenih poligona ili za identifikaciju objekata koji se nalaze unutar određene udaljenosti jedni od drugih.
- Računanje udaljenosti: Pruža alate za brzo računanje udaljenosti između geoprostornih objekata, što je korisno za aplikacije poput određivanja najbližih susjeda ili izračuna ruta. Na primjer, računanje udaljenosti između senzora i prijemnika je važno za analizu raspodjele resursa ili optimizaciju logistike.
- Analiza poligona: Omogućava analizu i manipulaciju poligonalnih podataka, uključujući operacije kao što su unija, presjek i razlika, te je ključno za zadatke kao

što su analiza teritorijalnih podjela i prostornih preklapanja. Recimo, unija poligona može se koristiti za stvaranje složenih područja interesa iz više izvora podataka, dok se presjek može koristiti za identificiranje zajedničkih područja između različitih skupova podataka.

4.2.2. Funkcionalnosti i primjene

cuSpatial nudi niz naprednih funkcionalnosti koje omogućuju detaljnu i brzu analizu geoprostornih podataka, prilagođenu specifičnim potrebama različitih aplikacija. Njena sposobnost da iskoristi GPU-ove za obradu podataka omogućava joj da pruži visok stupanj performansi i efikasnosti.

Prostorno pridruživanje u knjižnici cuSpatial omogućava korisnicima da učinkovito analiziraju prostorne odnose među različitim skupovima podataka. Primjerice, u urbanističkom planiranju, korisnici mogu brzo identificirati koje zgrade se nalaze unutar određenog radijusa od javnih usluga kao što su škole ili bolnice što omogućava planerima da donose informirane odluke na temelju preciznih prostornih analiza.

Računanje udaljenosti je još jedna bitna funkcionalnost koja se koristi u raznim industrijama. Na primjer, u logistici, cuSpatial omogućava optimizaciju ruta dostave, čime se smanjuju troškovi i povećava efikasnost operacija. Sposobnost brzog računanja udaljenosti između različitih lokacija omogućava tvrtkama da bolje planiraju svoje operacije i optimiziraju resurse.

Analiza poligona je također fundamentalna za mnoge primjene. U kontekstu okolišnih istraživanja, cuSpatial može doprinijeti analizi zemljišnih površina, identifikaciji promjena u korištenju zemljišta i praćenju utjecaja klimatskih promjena. Na primjer, analiza presjeka poligona može se koristiti za praćenje deforestacije ili procjenu utjecaja urbanizacije na prirodne ekosustave.

cuSpatial se koristi i u navigacijskim aplikacijama, gdje omogućava precizno praćenje i analizu kretanja objekata. To uključuje aplikacije poput praćenja vozila u realnom vremenu, optimizacije ruta i analize prometnih obrazaca. Korištenje GPU-ova omogućava izvođenje ovih operacija u stvarnom vremenu, što je ključno za aplikacije koje zahtijevaju brze i točne informacije.

U urbanističkom planiranju, cuSpatial pomaže planerima da analiziraju infrastrukturne podatke i donose odluke o razvoju gradskih područja. Brza analiza podataka o prometu, korištenju zemljišta i demografskim trendovima omogućava planerima da bolje razumiju potrebe zajednice i optimiziraju resurse za budući razvoj.

U zaključku, cuSpatial nudi napredne funkcionalnosti koje omogućuju brzu i detaljnu analizu geoprostornih podataka. Njen kapacitet da iskoristi GPU-ove za paralelnu obradu podataka čini je moćnim alatom za različite primjene, od urbanističkog planiranja do okolišnih istraživanja i logistike. Korisnici mogu iskoristiti ove funkcionalnosti za donošenje informiranih odluka i optimizaciju svojih operacija, čime se postižu bolji rezultati i povećava efikasnost.

4.2.3. Java Native Interface (JNI)

Java Native Interface (JNI) je infrastruktura koja omogućava Java kodu da integrira s aplikacijama i knjižnicama napisanim u drugim programskim jezicima, kao što je C++. To se postiže putem niza API-ja koji omogućuju Java aplikacijama da pozivaju nativne metode i obrnuto. Integracija JNI-ja u Java aplikaciju uključuje nekoliko koraka:

- Deklaracija nativnih metoda u Java klasi: Prvi korak je deklaracija nativnih metoda u Java klasi. Ove metode su implementirane u C++ kodu i pozivaju se iz Jave. Na primjer `public native void processSpatialData(String filePath);`.

- Implementacija nativnih metoda u C++ kodu: Nakon deklaracije nativnih metoda u Java klasi, sljedeći korak je implementacija tih metoda u C++ kodu. Korištenjem JNI API-ja, C++ kod može pristupati podacima iz Java aplikacije i koristiti funkcionalnosti knjižnici libcuspatial.

- Učitavanje nativnih knjižnica: Konačno, potrebno je učitati nativne C++ knjižnice unutar Java aplikacije. Ovo se postiže korištenjem `System.loadLibrary("libraryName");` metode u Javi.

4.3. Integracija knjižnice cuSpatial s Java kodom

Korištenjem JNI-ja, Java aplikacije mogu iskoristiti sposobnost knjižnice cuSpatial za obradu geoprostornih podataka na GPU-ovima. Proces integracije uključuje nekoliko ključnih koraka:

- Implementacija nativnih metoda u C++ kodu pomoću JNI API-ja, koristi se `JNIEXPORT *povratna vrijednost* JNICALL metoda`, u slučaju knjižnice cuSpatial u C++ kodu moramo pozivati te metode, to jest uvesti .hpp datoteke iz te knjižnice.

- Generiranje knjižnice iz C++ koda koji poziva nativne metode, koristi se G++.

- Deklaracija nativnih metoda u Java kodu koje prostupaju C++ kodu, koristi se ključna riječ `native`.

- Učitavanje nativne knjižnice koja je generirana iz C++ koda koji poziva odgovarajuću nativnu metodu, koristimo `System.loadLibrary("libraryName");` ili `System.load("libraryName");`.

- U slučaju korištenja platforme Spark Rapids, potrebno je nadjačati RapidsUDF metode koje osiguravaju da se kod u slučaju grešaka vezanih za grafičku karticu može izvoditi na procesoru, potrebno je implementirati `call()` i `evaluateColumnar()`. U prvoj funkciji navodimo što želimo da procesor radi ako je grafička kartica nedostupna dok u drugoj funkciji navodimo što želimo da se odvija na grafičkoj kartici.

5. Praktični dio

U praktičnom dijelu ovog rada cilj je pokazati izvođenje jedne nativne metode knjižnici cuSpatial na platformi Spark korištenjem grafičke kartice. Kod je napisan po uzoru Nvidiinih primjera za Spark[16]. Uz to, usporediti će se izvođenje na grafičkoj kartici i izvođenje na procesoru bez korištenja Rapids Akceleratora. Zbog neuspjelog izvođenja programa na grozdu Fakulteta Elektrotehnike i Računarstva, programi su izvođeni na računalu s kompatibilnom grafičkom karticom i moguće je pokazati ubrzanje u usporedbi s procesorom.

Primjer koji će biti pokazan je računanje udaljenosti između lokacija zadanih koordinata koje uzima zemljinu zakrivljenost u obzir. Generirano je od 10 tisuća do 50 milijuna nasumičnih parova koordinata u formatu Parquet i obavljeno je računanje udaljenosti i sortiranje tih udaljenosti. Ovo je dobar primjer za usporedbu između GPU i CPU procesiranja podataka i pokazivanja funkcioniranja platforme Spark Rapids jer sadrži primjer obrade podataka koji je jako zahtjevan ali također je jako paraleliziran. Također se očekuje razlika u ubrzanju ovisno koliko podataka obrađujemo, očekujemo da GPU ima veću prednost pri većoj količini podataka zbog duljeg vremena inicijalizacije za GPU u odnosu na CPU

5.1. Opis studijskog slučaja

5.1.1. Geoprostorni podatci i očekivani rezultati

Kao podatci za testiranje programskog rješenja izabrani su parovi koordinata na zemljinoj kugli i cilj je izračunati udaljenost među parovima. Računamo Haversineovu udaljenost koja računa udaljenost uzimajući u obzir promjer Zemlje, u ovom slučaju 6371 km. Takva računanja su zahtjevnija od računa udaljenosti na projekciji zemljine površine na

ravnoj plohi, ali su točnija.

Za generiranje podataka napravljena je Python skripta koja generira 50 milijuna redaka od kojih svaki redak ima nasumično generirani par koordinata zadanih geografskom širinom i duljinom. Primjer retka: `30.2834, 50.3423, -45.2942, 120.4532`. Podatci su spremljeni u formatu Parquet koji podatke sprema stupčano, što ga čini pogodnim za obradu velikih količina podataka, u suprotnosti s formatom CSV koji podatke organizira po retcima.

Programsko rješenje osmišljeno je tako da čita podatke u formatu Parquet, računa udaljenost para koordinata u svakom retku, te na svaki redak dodaje novi podatak s izračunatom udaljenosti. Na kraju sortira podatke po izračunatoj udaljenosti i sprema sve u novu Parquet izlaznu datoteku.

5.1.2. Specifikacije računala za izvođenje programskog primjera

Za obradu navedenih podataka korišten je laptop s sljedećim specifikacijama:

- CPU: Intel Core i7-9750H (6 fizičkih jezgri, 12 logičkih jezgri)
- GPU: Nvidia GeForce RTX 2060 (1920 CUDA jezgri, 6gb VRAM)
- Memorija: 16gb
- Operativni sustav: Ubuntu 22.04

Specifikacije programskih platformi i knjižnica:

- Apache Spark - 3.3.1
- CUDA Toolkit - 11.8
- cuSpatial, cuDF, rmm - 23.02 (korištena platforma MiniConda za instalaciju)
- Java - 8
- GCC, G++ - 11

5.2. Razvoj programskog rješenja

Za implementaciju rješenja korištena su 3 programska jezika: C++, Java i Python. C++ poziva nativnu metodu iz knjižnice cuSpatial i implementira knjižnicu JNI kako bi Java program mogao pristupiti toj funkciji.

```
//HaversineDistanceNative.cpp
#include <jni.h> //potrebno za interakciju s JNI
#include <cuspatial/distance/haversine.hpp> //funkcija za Haversine
    udaljenost
#include <memory>
#include <cudf/column/column.hpp>
#include <cudf/column/column_view.hpp>

extern "C" {
    JNIEXPORT jlong JNICALL
        Java_com_example_haversinedistance_HaversineDistance_computeHaversine
      (JNIEnv* env, jclass, jlong lat1Addr, jlong lon1Addr, jlong lat2Addr,
          jlong lon2Addr) {

        auto lat1_col = reinterpret_cast<cudf::column_view const*>(lat1Addr);
        auto lon1_col = reinterpret_cast<cudf::column_view const*>(lon1Addr);
        auto lat2_col = reinterpret_cast<cudf::column_view const*>(lat2Addr);
        auto lon2_col = reinterpret_cast<cudf::column_view const*>(lon2Addr);

        // Compute the Haversine distance
        auto distances = cuspatial::haversine_distance(
            *lat1_col, *lon1_col, *lat2_col, *lon2_col);

        // Return the native pointer to the new column
        return reinterpret_cast<jlong>(distances.release());
    }
}
```

Programski kod iznad koristi se kao veza između Jave i knjižnice cuSpatial koja podržava izvršavanje metoda na GPU-u. Zbog potrebe da se ovaj kod poziva u Javi, potrebno je koristiti `extern "C"` koji osigurava da JNI može prepoznati funkciju. JNI također zahtjeva korištenje `JNIEXPORT jlong JNICALL Java_com_example_haversinedistance_HaversineDistance_computeHaversine`, gdje definiramo povratni tip i koristimo makroe koji su potrebni za JNI. Ime JNI funkcije mora odgovarati paketu, klasi i metodi u Java kodu.

`auto lat1_col = reinterpret_cast<cudf::column_view const*>(lat1Addr);` pretvara `jlong` adrese u `cudf::column_view const*` pointere koji se mogu slati funkciji za izračun udaljenosti. Na kraju pretvaramo povratnu vrijednost natrag u `jlong` i s funkcijom `distances.release()` vraćamo pokazivač, oslobađamo vlasništvo i predajemo ga Java kodu. Iz ovog koda generiramo knjižnicu .so koju Java kod može učitati.

```
//HaversineDistance.java
package com.example.haversinedistance;

import ai.rapids.cudf.ColumnVector;
import com.nvidia.spark.RapidsUDF;
import org.apache.spark.sql.api.java.UDF4;

public class HaversineDistance implements UDF4<Double, Double, Double,
    Double, Double>, RapidsUDF {

    // Native method declaration for columnar processing
    private native long computeHaversine(long lat1Addr, long lon1Addr, long
        lat2Addr, long lon2Addr);

    @Override
    public Double call(Double lat1, Double lon1, Double lat2, Double lon2) {
        final int R = 6371; // Radius of the Earth in kilometers
        double latDistance = Math.toRadians(lat2 - lat1);
        double lonDistance = Math.toRadians(lon2 - lon1);
        double a = Math.sin(latDistance / 2) * Math.sin(latDistance / 2)
```

```
                + Math.cos(Math.toRadians(lat1)) *
                  Math.cos(Math.toRadians(lat2))
                + Math.sin(lonDistance / 2) * Math.sin(lonDistance / 2);
        double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
        double distance = R * c; // Convert to kilometers

        return distance;
    }

    @Override
    public ColumnVector evaluateColumnar(int numRows, ColumnVector... args) {
        UDFNativeLoader.ensureLoaded();
        ColumnVector lat1 = args[0];
        ColumnVector lon1 = args[1];
        ColumnVector lat2 = args[2];
        ColumnVector lon2 = args[3];
        long nativeResult = computeHaversine(lat1.getNativeView(),
            lon1.getNativeView(), lat2.getNativeView(), lon2.getNativeView());
        return new ColumnVector(nativeResult);
    }
}
```

Java kod iznad daje implementaciju koju Spark može koristiti u SparkSQL-u kao funkciju. Spark poziva tu funkciju kao UDF (User defined Function). Klasa HaversineDistance implementira 2 sučelja, `UDF4` i `RapidsUDF`. `UDF4` je potreban kako bi Sparku mogli dati CPU implementaciju za izvođenje izračuna. To radimo tako da nadjačamo metodu `call()` koja izračunava udaljenost na klasičan Java način i služi kao sigurnosna mreža ukoliko Spark ne može naći GPU ili on nije dostupan. Kome Rapids šalje podatke red po red stoga je takva implementacija inherentno sporija zbog slanja više manjih podataka.

Za implementaciju GPU izračuna koristimo `evaluateColumnar(int numRows, ColumnVector... args)`[14]. Njoj Rapids šalje vektore tipa `columnVector`, pri čemu svaki takav podatak sadrži cijeli vektor stupca. `UDFNativeLoader.ensureLoaded();`

je funkcija koja osigurava da je knjižnica koja je generirana iz C++ koda koja implementira izračun udaljenosti učitana. Nadalje pozivamo nativnu funkciju koja računa udaljenost i vraćamo nazad izračun udaljenosti tipa `columnVector`. Glavna razlika koju ovdje vidimo između CPU i GPU je da implentacija za CPU prima podatke redak po redak, dok implementacija za GPU odmah prima sve podatke u jednom pozivu. Ovaj Java kod se kompajlira u JAR datoteku koju Spark može učitati.

```
//haversine_distance_calc.py
import sys
import time
from time import sleep
from pyspark.sql import SparkSession
from pyspark.sql.functions import array, col, concat_ws

if __name__ == '__main__':
    if len(sys.argv) < 3:
        raise Exception("Requires input and output paths.")

    inputPath = sys.argv[1]
    outputPath = sys.argv[2]

    spark = SparkSession.builder.getOrCreate()

    # Register the UDF
    spark.udf.registerJavaFunction("computeHaversine",
        "com.example.haversinedistance.HaversineDistance", None)

    # Read the input data
    #df = spark.read.csv(inputPath, header=True, inferSchema=True)
    df = spark.read.parquet(inputPath)

    # Compute the haversine distance using the UDF
    df = df.filter("lat1 is not NULL and lon1 is not NULL and lat2 is not
        NULL and lon2 is not NULL")
```

```
df = df.selectExpr('lat1', 'lon1', 'lat2', 'lon2',
    'computeHaversine(lon1, lat1, lon2, lat2) as distance')
df = df.sort(col("distance"))

# Write the output data
start_time = time.time()
#df.write.mode("overwrite").csv(outputPath, header=True)
df.write.mode("overwrite").parquet(outputPath)
end_time = time.time()

print(f"==> Processing took {round(end_time - start_time, 2)} seconds")
spark.stop()
```

Navedeni kod iznad učitava UDF funkciju koju smo prethodno implementirali u Java kodu. On prima ulaznu Parquet datoteku i put do mjesta gdje treba spremiti izlazne podatke.

`spark = SparkSession.builder.getOrCreate()` stvara Spark okolinu. `spark.udf.registerJavaFunction()` poziva Java funkciju kao UDF koji se može koristiti u SQL pozivima. Zatim čitamo Parquet datoteku i spremamo je u `Dataframe`, filtriramo potencijalne NULL objekte jer RAPIDS ne podržava takve objekte. U nastavku pozivamo SQL naredbu koja koristi našu metodu za računanje udaljenosti potom sortiramo sve po stupcu koji ima spremljenu izračunatu udaljenost. Nakon toga krenemo računati vrijeme obrade podataka, koje možemo krenuti zbog tu zbog lijene obrade podataka koja će započeti tek kada pozovemo akciju. U ovom slučaju pozivamo akciju koja piše podatke i Spark automatski odrađuje sve transformacije koje smo prethodo odredili. Za kraj pozivamo `spark.stop()` koji zaustavlja Spark okolinu.

5.3. Instalacija i pokretanje primjera

Za pokretanje ovog primjera korišten je operativni sustav Ubuntu 22.04 s kompatibilnom Nvidia 2060 grafičkom karticom. U sljedećim poglavljima objašnjena će biti instalacija svih potrebnih alata za izgradnju i pokretanje projekta.

5.3.1. Instalacija potrebnih alata

Sve verzije alata i programa su zadnje kompatibilne u trenutku pisanja ovog rada.

Java

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install openjdk-8-jdk openjdk-8-jre
```

gcc i g++

Provjeriti verzije s naredbama `gcc -version` i `g++ -version`, ako verzija nije 11.x, koristiti sljedeće naredbe.

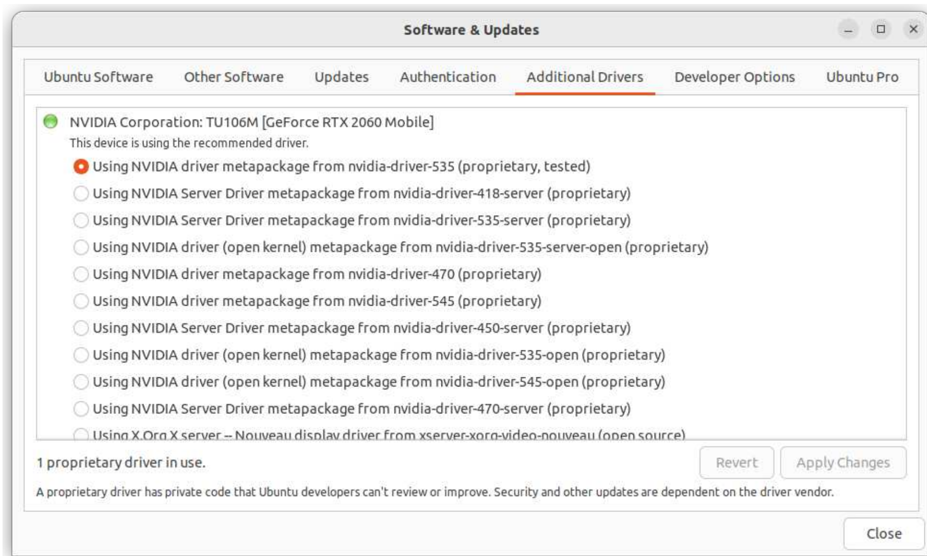
```
sudo apt update
sudo apt install gcc-11 g++-11 -y
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-11 11
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-11 11
#s sljedecim naredbama odabrati verziju 11 oba programa
sudo update-alternatives --config gcc
sudo update-alternatives --config g++
```

mvn, cmake i ninja

```
sudo apt install maven
sudo apt install cmake
sudo apt install ninja-build
```

Nvidia driveri i CudaToolkit

Driveri za grafičku karticu instalirani su preko Ubuntu Software Updates



Slika 5.1. Nvidia instalacija drivera

Zatim je potrebno instalirati CudaToolkit, u postavkama instalacije potrebno odznačiti instalaciju drivera jer su već instalirani.

```
wget https://developer.download.nvidia.com/compute/cuda/11.8.0/\
local_installers/cuda_11.8.0_520.61.05_linux.run
sudo sh cuda_11.8.0_520.61.05_linux.run
```

Provjeriti uspješnost instalacije s naredbama `nvidia-smi` za drivere i `nvcc -version` za CudaToolkit

Instalacija platforme Spark i odgovarajućeg Rapids plugin-a

```
sudo mkdir -p /opt/spark
cd /opt/spark
sudo wget https://dlcdn.apache.org/spark/spark-3.3.1/\
spark-3.3.1-bin-hadoop3.tgz
sudo tar -xvf spark-3.3.1-bin-hadoop3.tgz
```

Provjeriti uspješnost instalacije s naredbom `spark-shell`.

S stranice `https://nvidia.github.io/spark-rapids/docs/archive.html` skinuti verziju Rapids Plugin-a verzije 23.02 te napraviti novi direktorij `sudo mkdir /opt/sparkRapidsPlugin` i tu premjestiti skinutu .jar datoteku. U isti direktorij kopirati datoteku koja se nalazi na `/opt/spark/spark-3.3.1-bin-hadoop3/examples/src/main/scripts/getGpusResources.sh`.

Instalacija Miniconda3 i postavljanje okoliša

```
mkdir -p ~/miniconda3
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh \
-O ~/miniconda3/miniconda.sh
bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3
rm -rf ~/miniconda3/miniconda.sh
~/miniconda3/bin/conda init bash

conda create --name rapids-11.8
conda activate rapids-11.8

conda install libcuspatial-23.02.00-cuda11_g6fe3841_0.tar.bz2
conda install libcudf-23.02.00-cuda11_g5ad4a85b9d_0.tar.bz2
conda install librmm-23.02.00-cuda11_g48e8f2a8_0.tar.bz2
```

Postavljanje varijabli okoline

Otvoriti datoteku `/.bashrc` u tekst uređivaču i dodati sljedeće linije.

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export PATH=$JAVA_HOME/bin:$PATH

export SPARK_HOME=/opt/spark/spark-3.3.1-bin-hadoop3
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/bin

export SPARK_RAPIDS_DIR=/opt/sparkRapidsPlugin
export SPARK_RAPIDS_PLUGIN_JAR=${SPARK_RAPIDS_DIR}/\
rapids-4-spark_2.12-23.02.0.jar
```

```
export PATH=/usr/local/cuda/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-11.8/lib64\
${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

Pokrenuti naredbu `source .bashrc`.

5.3.2. Postavljanje i pokretanje primjera

Postavljanje strukture primjera

Ovako je potrebno postaviti strukturu programa za izgradnju. Također se može skinuti s Github poveznice s svim Python skriptama - `https://github.com/nikolaradelic/DiplomskiRad/tree/main`.

```
/home/user/DiplomskiRad
|-- haversine_distance_calc.py
|-- generate_parquet.py
|-- haversineDistance
    |-- pom.xml
    |-- src
        |-- main
            |-- java
            |   |-- com
            |       |-- example
            |           |-- haversinedistance
            |               |-- HaversineDistance.java
            |               |-- UDFNativeLoader.java
            |-- native
                |-- CMakeLists.txt
                |-- src
                    |-- HaversineDistanceNative.cpp
```

Python skripte nalaze se u privitku na kraju diplomskog rada. Također je potrebno napraviti dvije nove datoteke, to su pom.xml i CMakeLists.txt potrebne za izgradnju koda. Te datoteke se nalaze u privitkuna kraju diplomskog rada.

Izgradnja izvršnog koda

Potrebno se pozicionirati u direktorij s pom.xml datotekom i pokrenuti naredbu `mvn package`.

Nakon uspješne izgradnje izvršnog koda potrebno je pokrenuti Python skripte za generaciju nasumičnih podataka generate_parquet.py koji se nalazi u privitku na kraju diplomskog rada.

Pokretanje primjera

Za pokretanje primjera potrebno je pozicionirati se u direktorij `DiplomskiRad` i pokrenuti sljedeću Spark naredbu. Potrebno je samo promijeniti ime korisnika u postavljanju .jar datoteke ukoliko se radilo u Home direktoriju operativnoga sustava.

```
$SPARK_HOME/bin/spark-submit
    --master local[*] #lokalno izvodenje s svim dostupnim jezgrama
    --driver-memory 8g #memorija alocirana driver procesu
    --conf spark.task.cpus=1 #broj procesora
    --conf spark.rapids.sql.enabled=true #omogucena obrada na GPU
    --conf spark.rapids.sql.explain=ALL #javljanje funkcija koje se obraduju
        na GPU
    --conf spark.sql.adaptive.enabled=true #dodatna RAPIDS optimizacija
    --conf spark.plugins=com.nvidia.spark.SQLPlugin #Rapids SQL podrska
    --conf spark.executor.resource.gpu.discoveryScript= \
    $SPARK_RAPIDS_DIR/getGpusResources.sh #skripta koja javlja dostupne
        graficke kartice
    --conf spark.executor.resource.gpu.amount=1 #broj grafickih kartica
    --conf spark.rapids.sql.concurrentGpuTasks=1 #broj istovremenih zadataka
        na GPU
    --conf spark.rapids.memory.pinnedPool.size=2G
    --conf spark.sql.session.timeZone=UTC
    --driver-java-options "-Duser.timezone=UTC"
    --conf spark.executor.extraJavaOptions="-Duser.timezone=UTC"
    --conf spark.sql.legacy.timeParserPolicy=LEGACY
    --conf spark.rapids.sql.exec.CollectLimitExec=true
```

```
--files $SPARK_RAPIDS_DIR/getGpusResources.sh
--jars $SPARK_RAPIDS_DIR/rapids-4-spark_2.12-23.02.0.jar \
,/home/user/DiplomskiRad/haversineDistance/ \
target/haversineDistance-24.04.0-SNAPSHOT.jar
haversine_distance_calc.py test_input.parquet output
```

Napomene

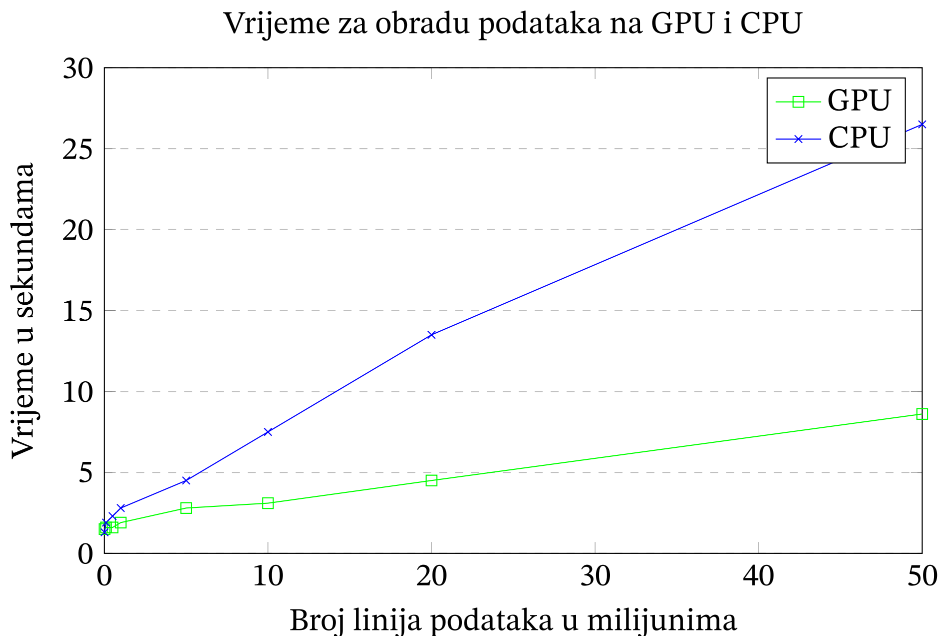
U izvršavanju koda, moguće je pojavljivanje greške vezane uz datoteku libgdal.so.31, to jest ne nalaženje iste. U tom slučaju pokrenuti sljedeću naredbu `sudo ln -s /usr/lib/libgdal.so.30 /usr/lib/libgdal.so.31`. Ta naredba stavlja simboličku poveznicu na datoteku libgdal.so.30 kako bi ju sustav vidio kao datoteku libgdal.so.31.

5.4. Evaluacija rješenja

Programsko rješenje je pokrenuto s obradom od 10 tisuća do 50 milijuna redaka enkodiranih u formatu Parquet. Cilj je bio pokrenuti program prvo s `-conf spark.rapids.sql.enabled=true` kako bi osigurali da će se obrada odrađivati na grafičkoj kartici i izmjeriti vrijeme obrade, a zatim pokrenuti program s `-conf spark.rapids.sql.enabled=false` koji onemogučuje provedbu na grafičkoj kartici i tjera Spark da koristi `call()` metodu iz `org.apache.spark.sql.api.java.UDF4` paketa što osigurava da se program vrti samo na procesoru.

Dobiveni rezultati su provjereni i dobivena rješenja su identična. Usporedbom vremena izvođenja rezultati su pokazali da je program izvođen na grafičkoj kartici do 3.1 puta brži u opsegu ovog testiranja. Graf ispod prikazuje vrijeme obrade podataka na jednoj osi i broj podataka u milijunima redaka na drugoj osi. Jasno se vidi kako grafička kartica povećava prednost s većom količinom podataka. To je očekivano s obzirom da je prijenos podataka i vrijeme za inicijalizaciju GPU-a pri malim količinama podataka ne zanemariv dio vremena. Pri količinama 10 tisuća redaka i manje se ispostavilo da je vrijeme obrade sporije na GPU u usporedbi s CPU, jer CPU izračuna sve podatke dok se podatci prenesu na GPU i inicijalizacija se odvije. No kako podatci rastu brža obrada i veća paralelizacija prevagnu i GPU već na 500 tisuća retka ima ubrzanje od 50%, dok na 50 milijuna retka ima ubrzanje od 3.1 puta naspram CPU. Naravno ovo ubrzanje je

idealizirani slučaj zbog visoko paralelne prirode ovih podataka i akcija koje se događaju nad njima. Da smo imali akcije i transformacije koje su namijenjene za serijsku obradu, zahtjeva puno I/O operacija s ostatkom računala i GPU, razlika bi bila puno manja ili bi čak CPU bio brži.



Dvije slike ispod pokazuju DAG izvođenja programa na GPU, 5.2. i CPU, 5.3. Na tim grafovima jasno možemo vidjeti gdje je koja implementacija učinkovitija. Učitavanje podataka je brže na CPU jer ima direktan pristup RAM memoriji u koju Spark učitava podatke, dok GPU mora prebacivati podatke iz RAM-a u VRAM (RAM memorija grafičke kartice). Jedna faza koju CPU mora odraditi, a GPU ne je pretvorba iz stupčanog tipa podataka u retčani tip. CPU nema podršku za rad s stupčanim podatcima dok GPU ima. Sljedeća je faza filtriranja podataka za koje GPU ima posebnu fazu dok CPU pretvorbu podataka, filtriranje i izračun udaljenosti spaja u jednu fazu zvanu `WholeStageCodgen`, ona u jeziku SparkSQL spaja više Java funkcija u jednu.

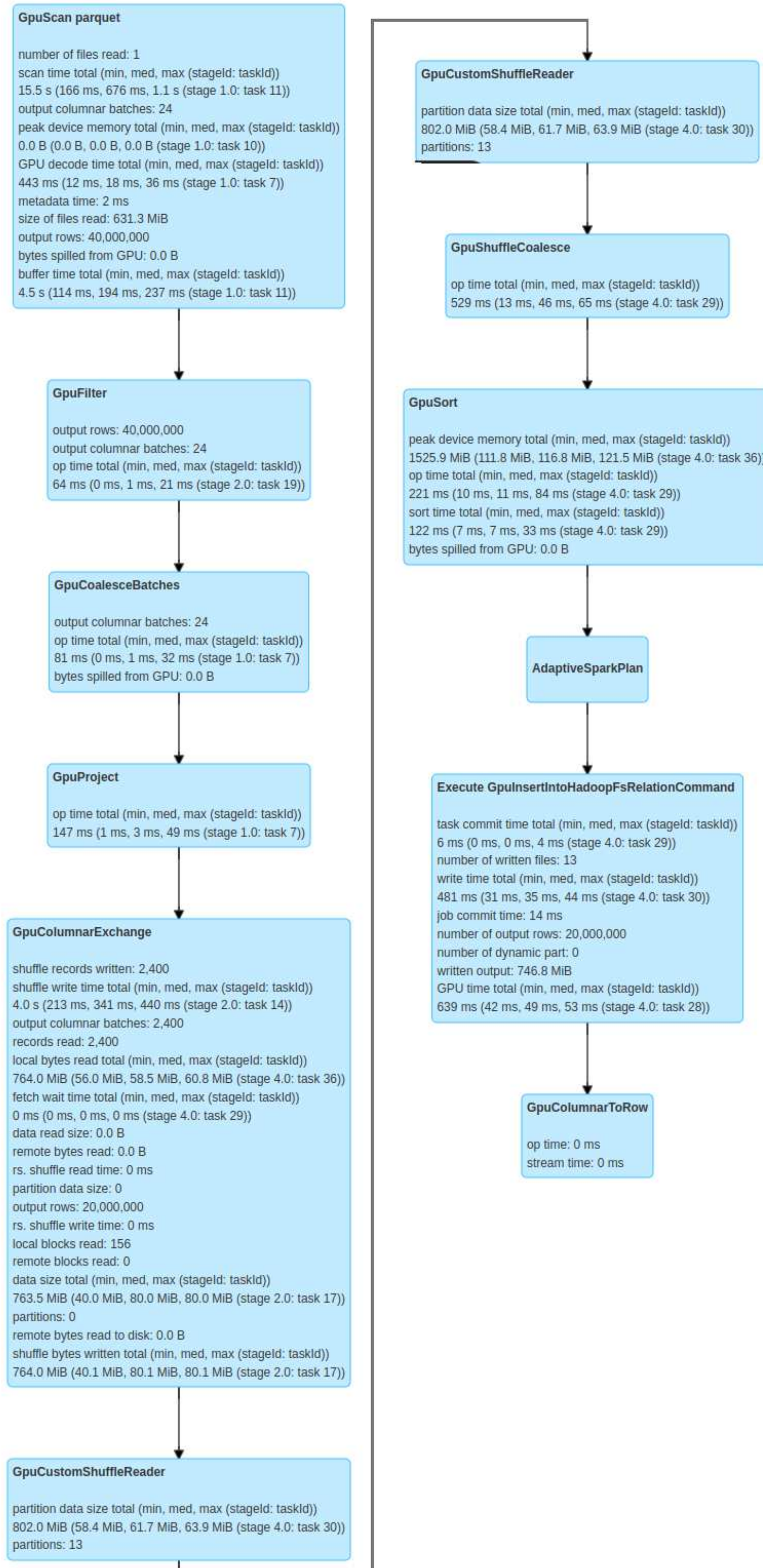
S tim je gotova prva faza obrade i za sljedeću fazu koja sortira podatke potrebno je odraditi `Exchange` koja raspoređuje podatke po čvorovima izvršiteljima. To je nužna faza za sortiranje jer su podatci dosad bili nasumično poredani. U praksi te operacije želimo što više smanjiti jer su skupe, pogotovo na velikim grozdovima gdje se podatci prebacuju preko mreže. Sljedeća faza koja je vidljivo brža na GPU je samo sortiranje, primarno jer sortiranje zahtjeva puno čitanja i premještanja podataka, a veza između RAM memorije

i CPU je puno sporija nego veza između VRAM memorije i GPU. Zadnja operacija koja je skupa na CPU je pretvaranje iz retčanih podataka u stupčani tip, to je jer je izlazni tip podataka u formatu Parquet. GPU to ne mora raditi jer on cijelo vrijeme radi s stupčanim podatcima i samo ih mora zapisati.

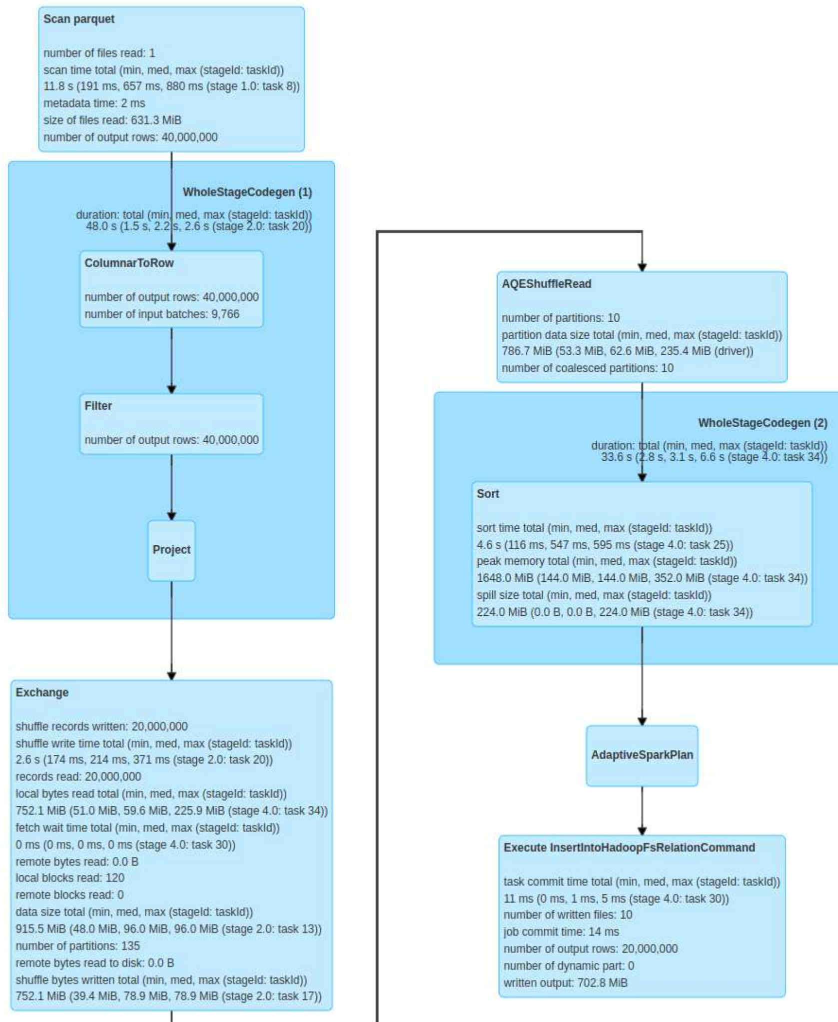
Details for Query 0

Submitted Time: 2024/06/24 15:47:24
Duration: 5 s
Succeeded Jobs: 1 2 3 4

Show the Stage ID and Task ID that corresponds to the max metric



Slika 5.2. DAG GPU



Slika 5.3. DAG CPU

6. Zaključak

Napretkom i razvojem današnjeg svijeta potrebno je obrađivati sve veće količine podataka. Kako su te količine danas u petabajtima i eksabajtima, vertikalno skaliranje računalne moći nije dovoljno i potrebno je horizontalno skalirati računala. Zbog čega je nastala potreba za raspodijeljenu raspodjelu i obradu podataka. Jedna od popularnih platformi za takvu obradu je Apache Spark, ona koristi takozvanu *in-memory* obradu podataka koja prvo učitava podatke na RAM memoriju i takve ih obrađuje što je dovelo do ubrzanja do 100 puta naspram prijašnjih tehnologija.

Jedna od novih inovacija u raspodijeljenoj obradi je upotreba grafičkih procesora. Oni za razliku od tradicionalnih procesora s 2 do 64 jake jezgre koriste procesore s nekoliko tisuća manjih jezgri. Takve jezgre omogućavaju izuzetno učinkovito paralelno računanje podataka i funkcija koje se daju paralelizirati. Tvrtka Nvidia je 2020. predstavila akcelerator RAPIDS kao dodatak za Apache Spark. On omogućuje da Spark uz dosad brzu obradu ugradi i podršku za grafičke procesore. On radi obrade nad DataFrame objektima kroz SQL upite. Osim toga, nudi knjižnice koje olakšavaju uklapanje RAPIDS-a s Sparkom kao cuDF, cuSpatial, cuGraph. Knjižnica na koju se ovaj rad fokusira je cuSpatial za obradu geoprostornih podataka. Ona još nema izravnu Java podršku pa je potrebno iz Jave direktno pozivati nativne metode iz C++ knjižnice libcuSpatial. Za to je nužno koristiti knjižnicu JNI i napraviti vlastitu knjižnicu koja poziva nativne metode.

U sklopu rada napravljeno je programsko rješenje koje poziva nativnu metodu za izračunavanje Haversine udaljenosti iz knjižnice cuSpatial u C++, poziva te metode u Javi i radi UDF (engl. user defined function) koji je moguće pozivati u kodu Python koji preko jezika SparkSQL poziva taj UDF kao funkciju SQL. Također, napravljena je implementacije koja računa udaljenost na procesoru kako bi se mogla usporediti vremena izvođenja.

Rezultati su pokazali ubrzanja do 3.1 puta, što nije potpuno ubrzanje od 4 puta koje tvrdi tvrtka NVIDIA, ali je dokaz da je moguće značajno ubrzati rad Sparka za odgovarajuće podatke i funkcije. Isto tako, zanimljivo je kako brzina RAPIDS-a jako ovisi o količini podataka. Na primjeru koji računa udaljenost na 50 milijuna podataka, ubrzanje je bilo 3.1 puta, dok na 10 tisuća podataka CPU ima blagu prednost. To je zbog skupog prebacivanja podataka na GPU i inicijalizacije. No, zbog velike količine podataka to vrijeme je zanemarivo i brzina obrade postaje glavni čimbenik.

Iako su dobiveni rezultati pokazali da je moguće značajno ubrzati obradu podataka korištenjem grafičkih kartica, potrebno je uzeti u obzir da nisu svi programi i podatci kompatibilni s paralelnom prirodom grafičkih kartica i obrada takvih podataka će biti brža na klasičnim procesorima. Potrebno je dobro proučiti svoje potrebe, podatke i programe kako bi se procijenilo ako je obrada na GPU-ovima isplativa promjena.

Literatura

[1] Apache Spark, https://spark.apache.org/, [mrežno; stranica posjećena: svibanj 2024.].

[2] NVIDIA, https://developer.nvidia.com/rapids, [mrežno; stranica posjećena: svibanj 2024.].

[3] ——, https://docs.rapids.ai/api/libcuspatial/stable/, [mrežno; stranica posjećena: svibanj 2024.].

[4] GCU, https://www.gcu.edu/blog/engineering-technology/what-are-4-vs-big-data, [mrežno; stranica posjećena: svibanj 2024.].

[5] Brady Betze, https://postperspective.com/review-nvidias-founders-edition-rtx-4090-an-editors-perspective/, [mrežno; stranica posjećena: svibanj 2024.].

[6] NVIDIA, https://www.nvidia.com/en-us/data-center/v100/, [mrežno; stranica posjećena: svibanj 2024.].

[7] Amit Joshi, https://medium.com/@amitjoshi7/spark-architecture-a-deep-dive-2480ef45f0be, [mrežno; stranica posjećena: svibanj 2024.].

[8] Naveen Nelamali, https://sparkbyexamples.com/spark/sparksession-explained-with-examples/, [mrežno; stranica posjećena: svibanj 2024.].

[9] Ashutosh Kumar, https://medium.com/@ashutoshkumar2048/dag-in-apache-spark-a3fee17f7494, [mrežno; stranica posjećena: svibanj 2024.].

[10] Daniel Ciocirlan, https://dzone.com/articles/reading-spark-dags, [mrežno; stranica posjećena: svibanj 2024.].

[11] Apache Spark, https://spark.apache.org/docs/3.0.0-preview/rdd-programming-guide.html, [mrežno; stranica posjećena: svibanj 2024.].

[12] ——, https://spark.apache.org/sql/, [mrežno; stranica posjećena: svibanj 2024.].

[13] Itamar Faran, https://medium.com/geekculture/dask-or-spark-a-comparison-for-data-scientists-d4cba8ba9ef7, [mrežno; stranica posjećena: svibanj 2024.].

[14] NVIDIA, https://nvidia.github.io/spark-rapids/docs/additional-functionality/rapids-udfs.html, [mrežno; stranica posjećena: svibanj 2024.].

[15] Michael Wang, https://medium.com/rapids-ai/cuspatial-at-the-end-of-2022-more-integrated-more-user-friendly-and-more-open-736660d908e3, [mrežno; stranica posjećena: svibanj 2024.].

[16] NVIDIA, https://github.com/NVIDIA/spark-rapids-examples, [mrežno; stranica posjećena: svibanj 2024.].

Sažetak

Raspodijeljena obrada velikih geoprostornih podataka na grafičkim procesorima

Nikola Radelić

Apache Spark je programska platforma za raspodijeljenu obradu velikih podataka na više računala. Trenutno je jedna od vodećih platformi otvorenog koda. Nvidia je 2020. godine predstavila akcelerator RAPIDS za platformu Spark koja omogućuje obradu podataka na grafičkim procesorima. Tvrtka Nvidia tvrdi ubrzanja za određene programe do 4 puta. U ovom radu istražena je platforma Apache Spark, akcelerator RAPIDS i knjižnica cuSpatial kako bi se provjerilo ubrzanje obrade podataka u usporedbi s tradicionalnim procesorima. Iz C++ knjižnice cuSpatial preko knjižnice JNI pozvana je nativna metoda koju Spark može koristiti kao SQL naredbu za obradu podataka. Pokazana su ubrzanja do 3.1 puta u usporedbi s obradom na procesoru.

Ključne riječi: spark, rapids, cuspatial, cuda, nvidia, apache, sql, udf, geoprostorni podatci, raspodijeljena obrada, jni

Abstract

Distributed Processing of Big Geosptial Data on Graphics Processors

Nikola Radelić

Apache Spark is a software platform for distributed processing of big data across multiple computers. It is currently one of the leading open-source platforms. In 2020, Nvidia introduced the RAPIDS accelerator for the Spark platform, which enables data processing on graphics processors. Nvidia claims up to 4x speed improvements for certain programs. This paper explores the Apache Spark platform, the RAPIDS accelerator, and the cuSpatial library to verify data processing speedups compared to traditional processors. A native method from the C++ cuSpatial library was called via the JNI library, which Spark can use as an SQL command for data processing. Speedups of up to 3.1x were demonstrated compared to processing on a CPU.

Keywords: spark, rapids, cuspatial, cuda, nvidia, apache, sql, udf, geospatial data, distributed computing, jni

Privitak A: The Code

```
//HaversineDistanceNative.cpp
#include <jni.h> //potrebno za interakciju s JNI
#include <cuspatial/distance/haversine.hpp> //funkcija za Haversine
#include <memory>
#include <cudf/column/column.hpp>
#include <cudf/column/column_view.hpp>

extern "C" {
    JNIEXPORT jlong JNICALL
        Java_com_example_haversinedistance_HaversineDistance_computeHaversine
      (JNIEnv* env, jclass, jlong lat1Addr, jlong lon1Addr, jlong lat2Addr,
         jlong lon2Addr) {

        auto lat1_col = reinterpret_cast<cudf::column_view const*>(lat1Addr);
        auto lon1_col = reinterpret_cast<cudf::column_view const*>(lon1Addr);
        auto lat2_col = reinterpret_cast<cudf::column_view const*>(lat2Addr);
        auto lon2_col = reinterpret_cast<cudf::column_view const*>(lon2Addr);

        // Compute the Haversine distance
        auto distances = cuspatial::haversine_distance(
            *lat1_col, *lon1_col, *lat2_col, *lon2_col);

        // Return the native pointer to the new column
        return reinterpret_cast<jlong>(distances.release());
    }
}
```

```java
//HaversineDistance.java
package com.example.haversinedistance;

import ai.rapids.cudf.ColumnVector;
import com.nvidia.spark.RapidsUDF;
import org.apache.spark.sql.api.java.UDF4;

public class HaversineDistance implements UDF4<Double, Double, Double,
    Double, Double>, RapidsUDF {

    // Native method declaration for columnar processing
    private native long computeHaversine(long lat1Addr, long lon1Addr, long
        lat2Addr, long lon2Addr);

    @Override
    public Double call(Double lat1, Double lon1, Double lat2, Double lon2) {
        final int R = 6371; // Radius of the Earth in kilometers
        double latDistance = Math.toRadians(lat2 - lat1);
        double lonDistance = Math.toRadians(lon2 - lon1);
        double a = Math.sin(latDistance / 2) * Math.sin(latDistance / 2)
                + Math.cos(Math.toRadians(lat1)) *
                    Math.cos(Math.toRadians(lat2))
                + Math.sin(lonDistance / 2) * Math.sin(lonDistance / 2);
        double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
        double distance = R * c; // Convert to kilometers

        return distance;
    }

    @Override
    public ColumnVector evaluateColumnar(int numRows, ColumnVector... args) {
        UDFNativeLoader.ensureLoaded();
        ColumnVector lat1 = args[0];
        ColumnVector lon1 = args[1];
```

```
        ColumnVector lat2 = args[2];
        ColumnVector lon2 = args[3];
        long nativeResult = computeHaversine(lat1.getNativeView(),
            lon1.getNativeView(), lat2.getNativeView(), lon2.getNativeView());
        return new ColumnVector(nativeResult);
    }
}
```

```java
//UDFNativeLoader.java
package com.example.haversinedistance;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.URL;

/** Loads the native dependencies for UDFs with a native implementation */
public class UDFNativeLoader {
  private static final ClassLoader loader =
      UDFNativeLoader.class.getClassLoader();
  private static boolean isLoaded;

  /** Loads native UDF code if necessary */
  public static synchronized void ensureLoaded() {
    if (!isLoaded) {
      try {
        String os = System.getProperty("os.name");
        String arch = System.getProperty("os.arch");
        File path = createFile(os, arch, "haversineudfjni");
        System.load(path.getAbsolutePath());
        isLoaded = true;
      } catch (IOException e) {
        throw new RuntimeException(e);
      }
    }
  }

  /** Extract the contents of a library resource into a temporary file */
  private static File createFile(String os, String arch, String baseName)
```

```
      throws IOException {
    String path = arch + "/" + os + "/" + System.mapLibraryName(baseName);
    File loc;
    URL resource = loader.getResource(path);
    if (resource == null) {
      throw new FileNotFoundException("Could not locate native dependency " +
          path);
    }
    try (InputStream in = resource.openStream()) {
      loc = File.createTempFile(baseName, ".so");
      loc.deleteOnExit();
      try (OutputStream out = new FileOutputStream(loc)) {
        byte[] buffer = new byte[1024 * 16];
        int read = 0;
        while ((read = in.read(buffer)) >= 0) {
          out.write(buffer, 0, read);
        }
      }
    }
    return loc;
  }
}
```

```python
//haversine_distance_calc.py
import sys
import time
from time import sleep
from pyspark.sql import SparkSession
from pyspark.sql.functions import array, col, concat_ws

if __name__ == '__main__':
    if len(sys.argv) < 3:
        raise Exception("Requires input and output paths.")

    inputPath = sys.argv[1]
    outputPath = sys.argv[2]

    spark = SparkSession.builder.getOrCreate()

    # Register the UDF
    spark.udf.registerJavaFunction("computeHaversine",
        "com.example.haversinedistance.HaversineDistance", None)

    # Read the input data
    #df = spark.read.csv(inputPath, header=True, inferSchema=True)
    df = spark.read.parquet(inputPath)

    # Compute the haversine distance using the UDF
    df = df.filter("lat1 is not NULL and lon1 is not NULL and lat2 is not
        NULL and lon2 is not NULL")
    df = df.selectExpr('lat1', 'lon1', 'lat2', 'lon2',
        'computeHaversine(lon1, lat1, lon2, lat2) as distance')
    df = df.sort(col("distance"))

    # Write the output data
    start_time = time.time()
    #df.write.mode("overwrite").csv(outputPath, header=True)
```

```
df.write.mode("overwrite").parquet(outputPath)
end_time = time.time()

print(f"==> Processing took {round(end_time - start_time, 2)} seconds")
spark.stop()
```

```
//generate_parquet.py
import pandas as pd
import random
import pyarrow as pa
import pyarrow.parquet as pq

# Function to generate random latitude and longitude
def generate_random_coordinates():
    lat = random.uniform(-90, 90)
    lon = random.uniform(-180, 180)
    return lat, lon

# Define the Parquet file path
parquet_file_path = 'test_input.parquet'

# Function to generate rows with random coordinates
def generate_random_rows(row_count):
    data = {'lat1': [], 'lon1': [], 'lat2': [], 'lon2': []}
    for _ in range(row_count):
        lat1, lon1 = generate_random_coordinates()
        lat2, lon2 = generate_random_coordinates()
        data['lat1'].append(lat1)
        data['lon1'].append(lon1)
        data['lat2'].append(lat2)
        data['lon2'].append(lon2)
    return data

# Number of rows to generate (you can change this value)
row_count = 20000000

# Generate the data rows
data = generate_random_rows(row_count)

# Convert to a DataFrame
```

```
df = pd.DataFrame(data)

# Write the data to a Parquet file
table = pa.Table.from_pandas(df)
pq.write_table(table, parquet_file_path)

print(f"Test Parquet file created at {parquet_file_path} with {row_count}
    rows")
```

```xml
//pom.xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
            http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>haversineDistance</artifactId>
  <version>24.04.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <java.major.version>8</java.major.version>
    <!--The last compatible plugin version is v23.02-->
    <rapids.version>23.02.0</rapids.version>
    <scala.binary.version>2.12</scala.binary.version>
    <spark.version>3.2.0</spark.version>
    <udf.native.build.path>${project.build.directory}/\
    cpp-build</udf.native.build.path>
    <CMAKE_CXX_FLAGS/>
    <CPP_PARALLEL_LEVEL>10</CPP_PARALLEL_LEVEL>
  </properties>

  <dependencies>
    <dependency>
        <groupId>com.nvidia</groupId>
        <artifactId>rapids-4-spark_${scala.binary.version}</artifactId>
        <version>${rapids.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
```

```
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_${scala.binary.version}</artifactId>
      <version>${spark.version}</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>
    <resources>
      <resource>
        <directory>${project.build.directory}/native-deps/</directory>
      </resource>
    </resources>
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>3.0.0</version>
        <executions>
          <execution>
            <id>cmake</id>
            <phase>validate</phase>
            <configuration>
              <target>
                <mkdir dir="${udf.native.build.path}"/>
                <exec dir="${udf.native.build.path}"
                      failonerror="true"
                      executable="cmake">
                  <arg value="${basedir}/src/main/native"/>
                  <arg value="-DCMAKE_CXX_FLAGS=${CMAKE_CXX_FLAGS}"/>
                </exec>
                <exec failonerror="true"
                      executable="cmake">
                  <arg value="--build"/>
                  <arg value="${udf.native.build.path}"/>
```

```
            <arg value="-j${CPP_PARALLEL_LEVEL}"/>
            <arg value="-v"/>
          </exec>
        </target>
      </configuration>
      <goals>
        <goal>run</goal>
      </goals>
    </execution>
  </executions>
</plugin>
<plugin>
  <artifactId>maven-resources-plugin</artifactId>
  <version>3.2.0</version>
  <executions>
    <execution>
      <id>copy-native-libs</id>
      <phase>validate</phase>
      <goals>
        <goal>copy-resources</goal>
      </goals>
      <configuration>
        <overwrite>true</overwrite>
        <outputDirectory>${project.build.directory}/native-deps/\
        ${os.arch}/${os.name}\</outputDirectory>
        <resources>
          <resource>
            <directory>${udf.native.build.path}</directory>
            <includes>
              <include>libhaversineudfjni.so</include>
            </includes>
          </resource>
        </resources>
      </configuration>
```

```
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
<profiles>
  <!-- This profile may not be needed -->
  <profile>
    <id>assembly-udf-jar</id>
    <build>
      <plugins>
        <plugin>
          <artifactId>maven-assembly-plugin</artifactId>
          <configuration>
            <descriptorRefs>
              <descriptorRef>jar-with-dependencies</descriptorRef>
            </descriptorRefs>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
</project>
```

```
//CMakeLists.txt

cmake_minimum_required(VERSION 3.20.1 FATAL_ERROR)

project(HAVERSINEUDFJNI VERSION 23.02.0 LANGUAGES C CXX CUDA)

# - build type

# Set a default build type if none was specified
set(DEFAULT_BUILD_TYPE "Release")

# - compiler options

set(CMAKE_POSITION_INDEPENDENT_CODE ON)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_COMPILER $ENV{CXX})
set(CMAKE_CXX_STANDARD_REQUIRED ON)

if(CMAKE_COMPILER_IS_GNUCXX)
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wno-unknown-pragmas")
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS}
        -Wno-error=deprecated-declarations")
endif(CMAKE_COMPILER_IS_GNUCXX)

# - find cuda toolkit
find_package(CUDAToolkit REQUIRED)

# - find cudf

find_path(
    CUDF_INCLUDE "cudf"
    HINTS "$ENV{CONDA_PREFIX}/include"
          "${CONDA_PREFIX}/include"
```

```
        "$ENV{CUDF_ROOT}/include"
        "${CUDF_ROOT}/include"
    REQUIRED)

find_library(
    CUDF_LIB "cudf"
    HINTS "$ENV{CONDA_PREFIX}/lib"
        "${CONDA_PREFIX}/lib"
        "$ENV{CUDF_ROOT}/lib"
        "${CUDF_ROOT}/lib"
    REQUIRED)

set(CUDACXX_INCLUDE "$ENV{CONDA_PREFIX}/include/rapids/libcudacxx")

# - find cuspatial

find_path(
    CUSPATIAL_INCLUDE "cuspatial"
    HINTS "$ENV{CONDA_PREFIX}/include"
        "${CONDA_PREFIX}/include"
        "$ENV{CUSPATIAL_ROOT}/include"
        "${CUSPATIAL_ROOT}/include"
    REQUIRED)

find_library(
    CUSPATIAL_LIB "cuspatial"
    HINTS "$ENV{CONDA_PREFIX}/lib"
        "${CONDA_PREFIX}/lib"
        "$ENV{CUSPATIAL_ROOT}/lib"
        "${CUSPATIAL_ROOT}/lib"
    REQUIRED)

# - find JNI
```

```
find_package(JNI REQUIRED)
if(JNI_FOUND)
    message(STATUS "JDK with JNI in ${JNI_INCLUDE_DIRS}")
else()
    message(FATAL_ERROR "JDK with JNI not found, please check your
        environment")
endif(JNI_FOUND)

# - library targets

add_library(haversineudfjni SHARED "src/HaversineDistanceNative.cpp")

# Override RPATH
set_target_properties(haversineudfjni PROPERTIES BUILD_RPATH "\$ORIGIN")

# include directories
target_include_directories(
    haversineudfjni
    PUBLIC "${JNI_INCLUDE_DIRS}"
           "${CUDF_INCLUDE}"
           "${CUDACXX_INCLUDE}"
           "${CUDAToolkit_INCLUDE_DIRS}"
           "${CUSPATIAL_INCLUDE}")

# - link libraries

target_link_libraries(haversineudfjni ${CUDF_LIB} ${CUSPATIAL_LIB})
```