

Subscale Algorythmus

William Mendat,¹ Max Ernst,² Steven Schall,³ Matthias Reichenbach⁴

Abstract: Dieses Dokument beinhaltet eine Beschreibung der Leistungen im Zuge des Teamprojekts in dem Studiengang Informatik-Master der Hochschule Offenburg im SS2022 und WS2022/23. Dabei wird auf Leistungen bezüglich Coding, Testing und Konzeption als auch auf organisatorische Aspekte eingegangen.

Keywords: C++; Subscale; Cluster

1 Einleitung

Für die Ausführung des Subscale-Algorithmus kann die Ausführungszeit bei großen Datensätzen trotz paralleler Ausführung auf einer GPU lange dauern. Deshalb gibt es die Überlegung, den Subscale-Algorithmus nicht nur parallel auf einer GPU auszuführen, sondern diesen zu verteilen und auf mehreren Rechnern, oder gar Clustern, zusammen zu berechnen.

2 Eigene Subscale Implementierung

TODO MAX SCHREIB MEHR DARÜBER

Da der Vorgegebene Code schwer zu verstehen und lesen war, wurde versucht einen eigenen Subscale zu implementieren. Dabei wurden der Subscale in einzelne Schritte aufgesplittet, damit diese unabhängig implementiert werden können. Die einzelnen Schritte waren dabei folgende:

Während der Implementierung stellte sich raus, dass noch weitere Schritte fehlen, wie das Erkennen von Subspaces und das Kombinieren dieser Subspaces.

3 Buildsystem

Damit der Entwickelte Code getestet werden kann, muss dieser vernünftig mit allen Notwendigen Libraries und Abhängigkeiten kompiliert werden. Hierfür musste das vorhandene

¹ Hochschule Offenburg, Offenburg, Deutschland wmendat@stud-hs.offenburg.de

² Hochschule Offenburg, Offenburg, Deutschland wmendat@stud-hs.offenburg.de

³ Hochschule Offenburg, Offenburg, Deutschland wmendat@stud-hs.offenburg.de

⁴ Hochschule Offenburg, Offenburg, Deutschland mreichen@stud-hs.offenburg.de

3.2 Makefile

Um das Projekt zu kompilieren, mussten manuell viele Parameter manuell angepasst werden. Zum Beispiel musste der Pfad zum vcpkg in einem CMakeLists-File angepasst werden. Um solche umständlichen und nicht ganz trivialen Schritte zu vereinfachen, wurde das Makefile erstellt. Dieses ist in Listing 1 dargestellt.

```

1  .PHONY: about init install-dependencies build compile start-subscale start-server start-
    client kill-server clean
2
3  VCPKG_DIR := ./include/vcpkg
4  VCPKG := ./include/vcpkg/vcpkg
5
6  about:
7      @echo "Makefile to help manage subscale gpu project"
8      @echo "commands:"
9      @echo "    - init: make sure vcpkg ist cloned as submodule"
10     @echo "    - install-dependencies: install all vcpkg dependencies"
11     @echo "    - build: build cmake changes"
12     @echo "    - compile: compile the code"
13     @echo "    - start-subscale: start subscale local"
14     @echo "    - start-server: starts a server to which the client sends data to
        calculate"
15     @echo "                                default port is 8080, else set with    -p=<port-number>"
16     @echo "    - start-client: starts execution subscale distributed"
17     @echo "    - kill-server: kills all servers still running in background"
18     @echo "    - removes builded files"
19
20  init:
21      git submodule update --init --recursive && $(VCPKG_DIR)/bootstrap-vcpkg.sh && mkdir
        Proto/generated
22
23  install-dependencies:
24      $(VCPKG) install nlohmann-json && $(VCPKG) install mlpack && $(VCPKG) install grpc
25
26  build:
27      cmake -S . -B ./debug
28
29  compile:
30      cmake --build ./debug
31
32  start-subscale:
33      ./debug/Subscale/subscale
34

```

```
35 start-server:
36     ./debug/Server/server $(p)
37
38 start-client:
39     ./debug/Client/client
40
41 kill-server:
42     killall ./debug/Server/server
43
44 clean:
45     rm -rf ./debug
```

List. 1: Makefile

Durch die Verwendung dieser Makefile-Befehle soll es vereinfacht werden, den Code zum einen manuell in seiner eignen Umgebung ausführen zu können, als auch zum anderen automatisiert innerhalb eines Docker-Containers.

Da das `vcpkg` als Submodule integriert wurde, was die automatisierte Installation und Pfadfindung von `vcpkg` als auch `gRPC`, `mlpack` und `nlohmann-json` ermöglicht, können alle benötigten Schritte per `make init` (Zeilen 20f) und `make install-dependencies` (Zeilen 23f) sehr einfach und automatisiert installiert und integriert werden. Es besteht weiterhin die Möglichkeit in den CMakeLists-Files dies manuell anzupassen und diese beiden Makefile-Befehle zu ignorieren. Dies ist sinnvoll, wenn man Docker-Container besitzt, welche dies im Vorhinein bereits installiert haben oder man das `vcpkg` aus seiner eigenen Entwicklungsumgebung nutzen möchte.

Um das Projekt an sich über CMAKE zu bauen, wird dies durch `make build` (Zeilen 26f) vereinfacht. Das Kompilieren des Codes wird auch durch das Makefile unterstützt. Dazu kann mit `make compile` (Zeilen 29f) der Code kompiliert werden.

Wenn man den Subscale-Algorithmus starten möchte, bietet das Makefile ebenfalls Möglichkeiten, dies vereinfacht zu tun. So kann mit `make start-subscale` (Zeilen 32f) die lokale Ausführung auf einem Rechner gestartet werden. Über `make start-server` (Zeilen 35f) beziehungsweise `make start-client` (Zeilen 38f) kann der Server beziehungsweise Client für die verteilte Ausführung gestartet werden. Wenn man den Server startet, hat man die Möglichkeit den Port, über welchen dieser gestartet werden soll, zu setzen. So kann man mit `make start-server -p=8080` sagen, dass der Server auf den Port 8080 hören soll.

Falls man einen Server richtig beenden möchte, weil dieser aus unerklärlichen Gründen weiterhin im Hintergrund läuft, kann man dies auch über den Befehl `make kill-server` tun und erspart sich die manuelle PID-Findung.

Zuletzt kann auch alles, was durch das Projekt gebaut wurde, wieder mit `make clean` (Zeilen 44f) entfernt werden.

3.3 Umstrukturierung

Der Vorgegebene Code war nicht dafür ausgelegt diesen zu verteilen, deshalb muss die Struktur angepasst werden. Dies ist vor allem auch notwendig damit unabhängig voneinander gearbeitet werden kann ohne das Merge-Konflikte entstehen. Der Vorgegebene Code hatte folgende Struktur:

```

├── data
├── packages
├── SubscaleGPU
│   └── ...
├── CMakeLists.txt
└── config.json

```

Der Komplette Subscale-Algorithmus befand sich in dem Ordner *SubscaleGPU*, die dazugehörigen Daten in dem *data* Ordner. Zusätzlich benötigte Libraries befanden sich im *packages* Ordner. Die Aktuelle Struktur erschwerte das hinzufügen einer Client-Server-Architektur. Der Code muss umstrukturiert werden, damit gleichzeitig an Client, Server und dem Subscale-Algorithmus gearbeitet werden kann. Der Komplette Subscale Algorithmus wurde in einen eigenen Ordner verschoben und als Statische Library weiterentwickelt, somit kann dieser einfach im Client und Server eingebunden werden. Des Weiteren benötigt sowohl Client als auch Server die Protobuf-Dateien, dafür wurden die .proto-Dateien in einen eigenen Ordner gelegt und von Client und Server verwendet zum Entwickeln der Schnittstellen. Die daraus entstandene Struktur sieht folgendermaßen aus:

```

├── Client
├── include
│   └── vcpkg
├── Proto
│   └── subscale.proto
├── Server
├── SubscaleGPU
│   ├── data
│   └── Config
└── CMakeLists.txt

```

4 Codebasis Umstrukturierung

Wir haben uns dann als Team dazu entschlossen, dass die eigene Implementierung gescheitert war und etwas anderes versucht werden musste. Die Idee war es, die vorhandene Implementierung zu nutzen, aber da der Code sehr schwer zu verstehen war und auch die Zeit so langsam ein Problem wurde, nur noch die Slices auf einem Cluster zu parallelisieren. Das Vorgehen dabei war es, den vorhandenen Code nicht wirklich zu verändern sondern, nur die vorhandenen Funktionen wieder zu verwenden.

Der originale Code hatte eine Klasse *LocalSubspaceTable*, die für das Darstellen der *Subspace*-Tabelle zuständig war. Diese Klasse wurde sowohl für die *Slices* als auch für die gesamten *Subspaces* verwendet. Meine Idee war es dann, den anderen im Team eine Funktion namens *calculateRemote* bereitzustellen, die als Parameter die *Labels*, den *minBound* des Slices und den *maxBound* des Slices bekommt und den berechneten Slice als Form der *LocalSubspaceTable*-Klasse zurückgibt.

Im Folgenden wird zunächst der Aufbau des originalen Code erläutert, um anschließend die Umsetzung der *calculateRemote*-Methode von unten nach oben darzustellen.

4.1 Aufbau des originale Code

Der Code bestand im wesentlichen aus einer abstrakten Klasse *ISubscale* und zwei konkreten Implementierungen *Subscale* (*Cuda*-Implementation) und *SubscaleSec* (normale Implementation) wie im folgenden gezeigt:

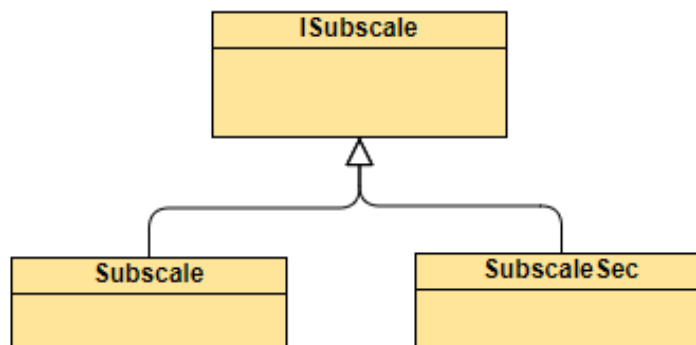


Abb. 2: Subscale

ISubscale selber hatte eine Funktion *calculateClusterCandidates* die Vorbereitungen getroffen hat und die konkrete Implementierung sollte die Funktion *calculateAllSlices* überschreiben, die den Zweck hatte, alle *Slices* zu erstellen und auf die Festplatte zu schreiben. Außerdem verfügte *ISubscale* noch über eine Funktion *combineAllSlices* die die einzelnen Slices von der Festplatte geholt hat, die Slice verengert hat und dann zu der endgültigen Subspace Tabelle hinzugefügt hat.

4.2 calculateSlice

Anfangen am untersten Ende brauchten die beiden konkreten Implementationen eine neue Funktion, die alle nötigen Informationen bekommt, um einen Slice zu erstellen und als Rückgabe dann den Slice zurückgibt. Da sich die konkreten Implementationen nur in der Erzeugung der konkreten Klassen unterscheidet, wird in dieser Section der generelle Code gezeigt und in den folgenden Abschnitten nur die wesentlichen Unterschiede.

```

1 LocalSubspaceTable* calculateSlice(...) {
2     // Vorbereitung (wird in den nächsten Sektionen gezeigt)
3     auto numberOfEntries = 0;
4
5     // initialize
6     denseUnitCreator->init(coreSets, labelsArray, numberOfPoints, config->minPoints);
7     subspaceJoiner->init(TICKET_SIZE);
8     // create dense units
9     denseUnitCreator->createDenseUnits(minSigBound, maxSigBound);
10

```

```

11 // filter out all dense units that only appear in one dimension and copy dense units to a
    subspace table
12 numberOfEntries = tableManager->duToSSTable(condensedSsTableWrapper->getPtr(),
    denseUnitTableWrapper->getPtr(), duTableSize);
13
14 // check if slice is empty
15 if (numberOfEntries > 0)
16 {
17     // join all entries by subspace
18     subspaceJoiner->join(condensedSsTableWrapper->getPtr(), numberOfEntries);
19
20     // condense table
21     numberOfEntries = tableManager->condenseTable(condensedSsTableWrapper->getPtr(),
    subspaceTableWrapper->getPtr(), ssTableSize);
22
23     // copy table to local memory
24     tableManager->deviceToLocal(localSubspaceTable, condensedSsTableWrapper->getPtr(),
    numberOfEntries);
25 }
26 // Free Memory
27 return localSubspaceTable;
28 }

```

List. 2: calculateSlice

Wie in Listing 2 zu sehen ist, werden die Berechnungsklassen initialisiert und anschließend werden die *DenseUnits* erzeugt, um diese dann in eine Subspace Tabelle hinzuzufügen. Sollte der *Slice* nicht leer sein, dann werden die Einträge noch nach den Subspaces zusammengefügt. Als letztes wird noch der Speicher von der Grafikkarte zum Host kopiert (ist bei der sequentiellen Abarbeitung nicht nötig).

4.3 calculateSlice Sequentiell

In dieser Sektion werden noch die spezifischen Erzeugungen der Berechnungsklassen für die Sequentielle Abarbeitung gezeigt.

```

1 LocalSubspaceTable* calculateSlice(...) {
2     // dense unit table
3     denseUnitTableWrapper = new LocalDenseUnitTable(config->minPoints, numberOfDimensions,
    duTableSize);
4     // subspace table 1
5     localSubspaceTable = new LocalSubspaceTable(numberOfPoints, numberOfDimensions,
    condensedSsTableSize);
6     // subspace table 2

```



```

7   subspaceTableWrapper = new LocalSubspaceTable(numberOfPoints, numberOfDimensions,
          ssTableSize);
8   // subspace table 3 (same as 1 because one fewer table is needed when all tables are on host
          memory)
9   condensedSsTableWrapper = localSubspaceTable;
10  // calculation classes
11  DenseUnitCreatorSeq* denseUnitCreator = new DenseUnitCreatorSeq(denseUnitTableWrapper->
          getPtr(), duTableSize);
12  SubspaceJoinerSeq* subspaceJoiner = new SubspaceJoinerSeq(subspaceTableWrapper->getPtr(),
          ssTableSize);
13  TableManagerSeq* tableManager = new TableManagerSeq();
14
15  // [...]
16 }

```

List. 3: calculateSlice Sequentiell

4.4 calculateSlice Sequentiell

In dieser Sektion werden noch die spezifischen Erzeugungen der Berechnungsklassen für die *Cuda* Abarbeitung gezeigt.

```

1 LocalSubspaceTable* calculateSlice(...) {
2   // dense unit table
3   denseUnitTableWrapper = new DeviceDenseUnitTable(config->minPoints, numberOfDimensions,
          duTableSize);
4   // subspace table 1
5   condensedSsTableWrapper = new DeviceSubspaceTable(numberOfPoints, numberOfDimensions,
          condensedSsTableSize);
6   // subspace table 2
7   subspaceTableWrapper = new DeviceSubspaceTable(numberOfPoints, numberOfDimensions,
          ssTableSize);
8   // subspace table 3
9   localSubspaceTable = new LocalSubspaceTable(numberOfPoints, numberOfDimensions,
          condensedSsTableSize);
10  DenseUnitCreator* denseUnitCreator = new DenseUnitCreator(denseUnitTableWrapper->getPtr(),
          duTableSize, config->threadsPerBlock, config->denseUnitsPerThread);
11  SubspaceJoiner* subspaceJoiner = new SubspaceJoiner(subspaceTableWrapper->getPtr(),
          ssTableSize, config->threadsPerBlock);
12  TableManager* tableManager = new TableManager(config->threadsPerBlock);
13  // [...]
14 }

```

List. 4: calculateSlice Cuda

4.5 calculateClusterCandidatesRemote

Die nächst obere Stufe war die *calculateClusterCandidatesRemote*-Methode, die alle notwendigen Informationen für die *calculateSlice* vorbereitet. Diese sieht wie folgt aus:

```
1 LocalSubspaceTable* calculateClusterCandidatesRemote(...) {
2     CsvDataHandler* csvDataHandler = new CsvDataHandler();
3     auto points = csvDataHandler->read(config->dataPath.c_str(), ',');
4     delete csvDataHandler;
5
6     auto numberOfDimensions = points[0].values.size();
7     // Shrink allocated memory of vector
8     points.shrink_to_fit();
9     CoreSetCreator* coreSetCreator = new CoreSetCreator();
10    // generate core sets
11    auto coreSets = coreSetCreator->createCoreSets(points, config->minPoints, config->epsilon);
12    delete coreSetCreator;
13
14    auto result = calculateSlice(coreSets, labes, numberOfDimensions, points.size(), min, max);
15
16    return result;
17 }
```

List. 5: calculateClusterCandidatesRemote

Diese hat die *CoreSets* erzeugt und die weiteren Informationen weiter runter gegeben.

4.6 calculateRemote

Der letzte Schritt war dann, die *calculateRemote*-Methode, die lediglich die *config* gelesen hat und dann die konkrete Implementierung des *Subscales* erzeugt hat.

```
1 LocalSubspaceTable* calculateRemote(...) {
2     SubscaleConfig* config = new SubscaleConfig();
3     config->readJson("Config/config.json");
4     ISubscale* subscale;
5     if (config->runSequential) {
6         subscale = new SubscaleSeq(config);
7     } else {
8         subscale = new Subscale(config);
9     }
10    return subscale->calculateClusterCandidatesRemote(lables, min, max);
11 }
```

List. 6: calculateRemote

5 gRPC

Der Subscale-Algorithmus wird per gRPC-Protokoll auf mehrere Server verteilt. Hierfür muss die gRPC-Library in das Projekt eingebunden werden, sowie Benötigte Protobuf-Files für die verwendeten Nachrichten. Nachdem die Library eingebunden ist und die Schnittstelle definiert, ist anhand der Protobuf-Files, muss der Client- und Server-Code implementiert werden. Im Folgenden Abschnitt wird die gRPC-Schnittstelle, mit deren Ausgetauschten Nachrichten und Methoden erläutert. Die Konkrete Implementierung dieser Schnittstelle wird in Kapitel 6 erläutert.

5.1 gRPC Schnittstelle

Damit der Subscale-Algorithmus verteilt werden kann muss eine Client-Server-Architektur aufgebaut werden. Dazu benötigt es ein Protokoll mit dem Client und Server Kommunizieren. Das hier verwendete Protokoll ist gRPC. Damit mit gRPC ein Client und Server erstellt werden kann, muss eine gemeinsame Schnittstelle definiert werden. Die Schnittstelle wird mittels Protobuf definiert. Hierfür wird Response- und Request-Nachrichten definiert, sowie der Service und dessen Methoden. Der Service wurde folgendermaßen definiert:

```
1 service SubscaleRoutes {
2     rpc RemoteSubscale (RemoteSubscaleRequest) returns (RemoteSubspaceResponse);
3 }
```

List. 7: gRPC-Service Definition

Dabei wird eine Methode definiert, welche ein *RemoteSubscaleRequest*-Nachricht an den Server sendet und eine *RemoteSubspaceResponse*-Nachricht zurückschickt. Dieser Service wird von dem Server implementiert und der Client kann diesen dann Aufrufen.

Die Request-Nachricht für den Aufruf beinhaltet die zuvor erstellten *lables* des Subscale-Algorithmus, sowie die Minimale- und Maximale Signatur für den zugehörigen Bereich. Die *lables* werden dabei als Liste übertragen, siehe Listing 8.

```
1 message RemoteSubscaleRequest {
2     repeated uint64 labels = 1;
3     uint64 minSignature = 2;
4     uint64 maxSignature = 3;
5 }
```

List. 8: gRPC-Request Message

Die Response-Nachricht bildet die Subscale-Tabelle ab und besteht aus einer Liste an *Entry*-Nachrichten sowie Drei weiteren Variablen, welche der Subscale-Algorithmus berechnet. Die *Entry*-Nachricht besteht aus einer Liste von *ids* und einer Liste von *dimenisons*. Sie

repräsentiert einen Eintrag in der Subscale-Tabelle und somit einen möglichen Cluster Kandidat.

```
1 message Entry {
2     repeated uint32 ids = 1;
3     repeated uint32 dimensions = 2;
4 };
5
6 message RemoteSubspaceResponse {
7     repeated Entry entries = 1;
8
9     int32 tableSize = 2;
10    int32 idsSize = 3;
11    int32 dimensionsSize = 4;
12 }
```

List. 9: gRPC-Response Message

Nachdem diese Schnittstelle mittels der Protobuf-Datei definiert ist, kann der Client und Server unabhängig voneinander entwickelt werden.

6 Verteilung

Im Folgenden Abschnitt werden die Implementierung der gRPC-Schnittstellen, sowie die Client- und Server Main Methoden erläutert und Probleme, die währenddessen aufgetreten sind.

6.1 Client

Main-Methode Der Client Code berechnet die *lables* für die gesamten Punkte sowie Minimale- und Maximale Signatur. Danach wird an jeden Server die *labels*, sowie Minimale- und Maximale Signatur gesendet und dessen Antwort in einen Vektor gespeichert. Damit die keine Race-Conditions entstehen muss das Schreiben in den Vektor synchronisiert werden mittels einem *mutex*. Zum Schluss muss auf alle Server gewartet werden damit die Tabellen zusammengeführt werden können.

```
1     std::mutex m;
2     grpc::ChannelArguments args;
3     args.SetLoadBalancingPolicyName("round_robin");
4     for (auto i{0}; i < config->splittingFactor; ++i)
5     {
6         workers.emplace_back(std::thread{[&](int index)
7                                     {
```

```

8      Client::Client client{grpc::CreateCustomChannel(addr, grpc::
      InsecureChannelCredentials(), args)};
9      auto result = client.remoteCalculation(labels, minSigBounds[index],
      maxSigBounds[index]);
10     m.lock();
11     tables.push_back(result);
12     m.unlock();
13     std::cout << "Table " << index << std::endl; },
14             i));
15 }
16
17 for (auto &worker : workers)
18     worker.join();

```

List. 10: Client Main-Methode

Nachdem alle Server den Subscale-Algorithmus ausgeführt haben und die Ergebnisse an den Client zurückgeschickt sind, werden anhand von den *RemoteSubspaceResponse* Objekten die Subspace-Tabelle aufgebaut. Diese Subspace-Tabelle beinhaltet alle möglichen Cluster Kandidaten und kann dem DB-Scan Algorithmus übergeben werden.

gRPC-Schnittstelle Der Client muss anhand der *labels* und Minimalen und Maximalen Signatur das Request Objekt befüllen und dies an den Server senden. Bei Positiver Antwort, wird die Response zurückgegeben und bei negativer Response eine Exception ausgelöst.

```

1      RemoteSubspaceResponse Client::remoteCalculation(std::vector<unsigned long long> labels,
      unsigned long long min, unsigned long long max)
2      {
3          auto* request = new RemoteSubscaleRequest();
4          request->set_minsignature(min);
5          request->set_maxsignature(max);
6          *request->mutable_labels() = {labels.begin(), labels.end()};
7
8          RemoteSubspaceResponse response;
9          ClientContext context;
10         Status status;
11         status = _stub->RemoteSubscale(&context, *request, &response);
12         delete request;
13
14         if (status.ok())
15             return response;
16
17         std::cout << "Error code: " << status.error_code() << std::endl;
18         std::cout << "Error Details: " << status.error_details() << std::endl;

```

```
19     std::cout << "Error Message: " << status.error_message() << std::endl;
20     throw std::runtime_error("GRPC Request Failed");
21 }
```

List. 11: Client gRPC-Aufruf

Client Konfiguration Damit der Client die Adressen der Server kennt, wurde eine Config-Datei zum Client hinzugefügt. Diese Konfiguration wird als JSON-Datei hinterlegt und beinhaltet alle Adressen der Server:

```
1 {
2   "servers": [
3     "127.0.0.1:2510",
4     "127.0.0.1:2511",
5     "127.0.0.1:2512",
6     "127.0.0.1:2513",
7     "127.0.0.1:2514"
8   ]
9 }
```

List. 12: Config beispiel

Eine Standard Config-Datei ist im Repository hinterlegt, es kann jedoch eine nicht Versionierte Config-Datei angelegt werden, um die Server Adressen anzupassen. Die Config ist als Singleton implementiert und kann mittels der *get*-Methode instantiiert und abgerufen werden. Die Methode prüft dabei auf eine *config.override.json*-Datei, falls diese nicht vorhanden ist wird eine Standard Konfiguration geladen und dabei die Server Adressen als Vektor gespeichert.

```
1 Config* Config::get()
2 {
3     if (instance == nullptr)
4     {
5         instance = new Config();
6         auto* f = new std::fstream("Client/config.override.json");
7         if (f->is_open())
8         {
9             std::cout << "found override Config" << std::endl;
10            instance->data = nlohmann::json::parse(*f);
11        }
12        else
13        {
14            std::cout << "Using Standard Config" << std::endl;
15            auto* f = new std::fstream("Client/config.json");
```

```

16         instance->data = nlohmann::json::parse(*f);
17     }
18     data.at("servers").get_to(servers);
19 }
20 return instance;
21 }

```

List. 13: Client-Config

6.2 gRPC Loadbalancing

Die gRPC Library und Implementierung des Service konnte ohne Probleme durchgeführt werden, jedoch gab es Probleme bei den Verwendungen mehrerer Server. Es wurde zuerst versucht pro Anfrage an einen Server einen neuen Client zu erstellen mit einer anderen Adresse, dies funktionierte jedoch nicht. Da gRPC es nicht ermöglicht mehrere Clients zu erstellen und der Code somit immer abstürzte. Nach langem Recherchieren stellte sich heraus, dass gRPC eine Loadbalancing Option ermöglicht. Somit kann eine Liste (bzw. ein String mit Komma separierten Adressen) übergeben werden und eine Loadbalancing Eigenschaft. Siehe Listing 10 Zeile Drei.

6.3 Server

Der Server Code führt den Kompletten Subscale-Algorithmus aus, dabei nutzt er die übermittelten *lables*, sowie die Minimale und Maximale Signatur, um mögliche Cluster Kandidaten zu berechnen.

Main-Methode Die Main Methode Startet den Server, mit dem übergebenen Port und Registriert den Implementierten *SubscaleRoutesService*. Danach wartet dieser auf eingehende Requests von einem Client.

```

1     std::string port;
2     if (argc > 1)
3         port.assign(argv[argc - 1]);
4     else
5         port.assign("2510");
6
7     std::string server_address = "localhost:";
8     server_address.append(port);
9     Server::SubscaleRoutesImpl service(server_address);
10
11     ServerBuilder builder;

```

```

12  builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
13  builder.RegisterService(&service);
14  std::unique_ptr<grpc::Server> server{builder.BuildAndStart()};
15
16  std::cout << "Server listening on " << server_address << std::endl;
17  server->Wait();
18  return 0;

```

List. 14: Server Main-Methode

gRPC-Service Implementation Die Implementation des gRPC-Service startet per aufruf der *Remote::calculateRemote*-Methode den Subscale-Algorithmus. Um die Methode zu verwenden, müssen die *labels* in einen Vektor überführt werden. Das Ergebnis des Subscale-Algorithmus ist ein Tupel, welches die Tabelle mit möglichen Cluster-Kandidaten beinhaltet und eine Anzahl von Einträgen in den Tabellen. Dieses Tupel wird in die *RemoteSubspaceResponse*-Nachricht überführt. Da die gRPC-Nachrichten nicht mit den Vektoren übermittelt werden können, müssen diese dementsprechend umgewandelt werden. Dabei werden beim Überführen Einträge mit einer *dimension* oder *id* Größe von Null herausgefiltert.

```

1  Status SubscaleRoutesImpl::RemoteSubscale(ServerContext *context, const
    RemoteSubscaleRequest *request, RemoteSubspaceResponse *response)
2  {
3      auto result = Remote::calculateRemote({std::begin(request->labels()), std::end(request
        ->labels())}, request->minsignature(), request->maxsignature());
4      auto table = std::get<0>(result);
5      auto numberOfEntries = std::get<1>(result);
6      auto tableSize = 0;
7
8      for (auto i = 0; i < numberOfEntries; i++)
9      {
10         auto ids = table->getIdsVec(i);
11         auto dimensions = table->getDimensionsVec(i);
12
13         if (ids.size() == 0 || dimensions.size() == 0)
14             continue;
15
16         Entry* entry = response->add_entries();
17         *entry->mutable_dimensions() = {dimensions.begin(), dimensions.end()};
18         *entry->mutable_ids() = {ids.begin(), ids.end()};
19
20         tableSize++;
21     }
22     response->set_idssize(table->getIdsSize());
23     response->set_dimensionssize(table->getDimensionsSize());

```



```
24     response->set_tablesize(tableSize);  
25  
26     return Status::OK;  
27 }
```

List. 15: Server gRPC Implementation

7 Cluster

Für das Testen der neu implementierten Verteilung wird das Projekt auf einem Cluster ausgeführt. Dazu werden von der Hochschule einige Rechner zur Verfügung gestellt.

7.1 Zugriff

Um generell auf das Cluster zugreifen zu können, werden n Personen benötigt. Dies liegt daran, dass das Cluster über bwLehrPool-Remot erreichbar ist. Ein anderes Cluster konnte uns zu dem Zeitpunkt nicht zur Verfügung gestellt werden. Da wir genug Personen im Team waren, um eine einigermaßen aussagekräftige Anzahl von Rechnern bereitstellen zu können, war dies für uns kein Problem. Jeder der n Personen meldet sich dann auf einem der Rechner in dem IMLA Pool an. Dort können dann die Container, welche später genauer erläutert werden, gestartet werden. Da diese alle in einem Netzwerk verfügbar sind, entsteht keine Schwierigkeit eine Verbindung zwischen den Rechnern des Pools herzustellen.

7.2 Dockerfile

Um ein Server oder Client auf einem Cluster-Rechner zu starten, wird dieser innerhalb eines Docker-Containers gestartet. Dieser Gedanke entstand aufgrund der Gegebenheiten, dass nicht immer alle benötigten Tools auf einem System installiert sind. Durch das Benutzen eines Docker-Containers können auf diesen, sofern für diesen alle benötigten Tools installiert sind, zum Beispiel die `nvidia-container-runtime`, alle noch nicht vorhandenen aber benötigten Tools nachinstalliert werden. In dem folgenden Listing 16 ist dieser Ansatz zu erkennen.

```
1 FROM nvcr.io/nvidia/cuda:12.0.0-devel-ubuntu20.04  
2  
3 WORKDIR /subscale  
4  
5 RUN apt update  
6 RUN DEBIAN_FRONTEND=noninteractive TZ=Etc/UTC apt install -y --allow-unauthenticated --no-  
    install-recommends tzdata
```

```
7  RUN apt install -y --allow-unauthenticated --no-install-recommends git curl zip unzip tar  
    pkg-config gfortran python3 cmake nano  
8  
9  COPY . .  
10  
11 RUN make init  
12 RUN make install-dependencies  
13 RUN make build  
14 RUN make compile
```

List. 16: Dockerfile

CUDA-Images gibt es in drei Varianten und sind über das öffentliche NVIDIA Hub Repository verfügbar.

- base: enthält ab CUDA 9.0 das absolute Minimum (libcudart) für den Einsatz einer vorgefertigten CUDA Anwendung.
- runtime: erweitert das Basis-Image um alle gemeinsam genutzten Bibliotheken des CUDA-Toolkits.
- devel: erweitert das Runtime-Image, indem es die Compiler-Toolchain, die Debugging-Tools, die Header und die statischen Bibliotheken hinzufügt.

Um bereits ein System zu haben, welches mit CUDA kompatibel ist, wurde sich für die devel-Version entschieden. Dabei wurde auch die neuste Version benutzt. Schlussendlich ist in Zeile 1 das verwendete Docker-CUDA-Image “nvcv.io/nvidia/cuda:12.0.0-devel-ubuntu20.04”, für welches sich entschieden wurde, zu sehen. Aufgrund einer Empfehlung von Prof. Dr.-Ing. Janis Keuper wird der Container direkt von NVIDIA genommen und nicht von Docker Hub.

Leider ist auch auf dem CUDA-Image nicht alles installiert, was benötigt wird, um das Projekt zu kompilieren. Aus diesem Grund werden von Zeile 5 bis Zeile 7 alle notwendigen Tools installiert. Dabei gab es vor allem die Schwierigkeit, cmake zu installieren. Als Lösung dafür musste, getrennt von allem, der Befehl in Zeile 5 `RUN apt update` und anschließend in Zeile 6 `tzdata` installiert werden. Dies ermöglicht die Umgehung des manuellen Setzens der Zeitzone, welche beim normalen Installieren von cmake auftritt. Auch war es notwendig, das `DEBIAN_FRONTEND` auf `nointeractive` zu setzen.

In Zeile 9 ist durch `COPY . .` der Befehl zu erkennen, welcher das Projekt in den Container kopiert. Anschließend werden von Zeile 11 bis 14 die normalen Befehle, welche benötigt werden, um den Code zu kompilieren, ausgeführt. Dafür wird das zuvor beschriebene Makefile verwendet, was an dieser Stelle eine sehr große Vereinfachung darstellt.

In dem Dockerfile wird kein `EXPOSE` festgelegt, um die spätere Containererstellung zu vereinfachen. Falls man auf einem Rechner testweise mehrere Container starten möchte, sollten nicht dieselben Ports exposed werden. Auch kann so sehr einfach automatisiert werden, welcher Container mit welchem Port laufen soll. Im späteren Verlauf des Dokuments wird aufgezeigt, wie der Port bei dem `docker run`-Befehl freigegeben wird.

7.3 Docker-Ausführung

Um den Container mit allen Konfigurationen zu bauen, wird das oben genannte Dockerfile verwendet. Dabei wird mit dem Befehl

```
docker build -t subscale .
```

der Container gebaut. Der Befehl wird in demselben Verzeichnis wie das Dockerfile ausgeführt. Dies kann einige Zeit in Anspruch nehmen, da dabei zuerst der Container heruntergeladen (falls noch nicht bereits geschehen) wird sowie das gesamte Projekt und auch Sub-Module kompiliert werden. Vor allem das `mlpack` scheint eine sehr zeitaufwendige Kompilierung zu haben. Anschließend kann mit dem Befehl

```
docker run --name subscaleGPUServer --gpus all --rm -it -p 8080:8080  
subscale make start-server p=8080
```

der Container gestartet werden. Durch die Namensgebung mit `--name` ist im anschließenden Schritt die Handhabung mit dem laufenden Container einfacher. Auch muss mit `--gpus all` der Zugriff auf GPU-Ressourcen ermöglicht werden. Die Flag `--rm` ist optional, aber empfohlen, da dies nach Beenden des Containers diesen automatisch entfernt. Anzumerken ist, dass dabei nicht das Image, welches durch den `docker build` Befehl entstanden ist, gelöscht wird. Durch `-p 8080:8080` wird der Port 8080 des Containers auf 8080 nach außen gemappt. Dann kann dieser Port dem `make start-server` als `p=8080` übergeben werden, woraufhin dieser direkt mit dem Port 8080 gestartet wird. Durch die Flag `-it` bekommt man die Sicht auf den Server, wodurch zu erkennen ist, ob ein Request des Clients ankam oder nicht.

Den Client, welcher die Berechnung startet, sollte nun über den folgenden Befehl ausgeführt werden.

```
docker run --name subscaleGPUClient --gpus all --rm -it  
subscale /bin/bash
```

Dadurch wird eine Konsole innerhalb des Containers geöffnet. Auch hier ist die Nutzung von `-it` sinnvoll. Dies liegt vor allem daran, dass dort auch das Ergebnis gespeichert wird. Anschließend kann durch `make start-client` der Prozess gestartet werden. Dafür muss natürlich die `Client/config.json` richtig angepasst sein.

Generell erhält man die IP-Adresse eines laufenden Containers über den folgenden Befehl.

```
docker container inspect subscaleGPUServer | grep -i IPAddress
```

Hierbei wird sichtbar, dass die Namensgebung den Vorteil bietet, nicht erst die ID des Containers herausfinden zu müssen. So ist es auch an dieser Stelle einfacher, eine Automatisierung zu ermöglichen.

8 Benchmarking

Für das Benchmarking wurde von jedem Teammitglied jeweils ein Docker-Container mit Server gestartet. Anschließend wurde auf einem der Cluster-Rechner der Client gestartet und die Ausführung gemessen. Das bedeutet, dass die gesamte Ausführungszeit des Clients das Messergebnis ergibt. Bei der sequentiellen Messung wird ebenfalls innerhalb des Containers der Subscale-Algorithmus gestartet, jedoch nicht die verteilte Version.

8.1 Sequentiell

Die sequentielle Berechnung innerhalb eines Containers ergab die folgende Ausführungszeit.

Ausführungsmessung	Zeit
1	2.685 s
2	2.659 s
3	2.675 s
4	2.681 s
5	2.711 s
6	2.657 s
7	2.679 s
8	2.662 s
9	2.667 s
10	2.676 s
Durchschnitt	2.675 s

Hierbei ist auffällig, dass sich alle Messungen sehr nahe beieinander befinden und davon auszugehen ist, dass der gemessene Wert eine gute Basis für einen Vergleich darstellt.

8.2 Verteilt

Bei der verteilten Berechnung über mehrere Container und Rechner hinweg ergaben sich folgende Messungen.

Ausführungsmessung	Zeit
1	10.736 s
2	11.285 s
3	10.982 s
4	10.924 s
5	10.859 s
6	10.908 s
7	11.753 s
8	11.813 s
9	11.348 s
10	11.633 s
Durchschnitt	11.224 s

Hier gibt es Abweichungen um bis zu einer Sekunde. Eine mögliche Quelle dieser Abweichungen besteht darin, dass das Netzwerk nicht immer gleich auf Anfrage und Antwort reagieren kann, was auf die nicht alleinige Nutzung dieses Netzwerks zurückzuführen ist.

8.3 Vergleich

Bei dem Vergleich der durchschnittlichen Laufzeit ist zu erkennen, dass der verteilte Algorithmus langsamer ist als der Sequentielle. In der folgenden Tabelle nochmal die langsamste und schnellste Messung, als auch der Durchschnitt aller Messungen gegenübergestellt.

Typ	Sequentiell	Verteilt
langsamste Messung	2.711 s	11.813 s
schnellste Messung	2.657 s	10.736 s
durchschnitt der Messungen	2.675 s	11.224 s

Die Vermutung liegt nahe, dass der uns zur Verfügung gestellten Testdatensatz nur bedingt für eine effizientere Ausführung durch eine Verteilung geeignet ist, da dieser dafür potenziell zu klein ist. Bei einem Datensatz, welcher noch sehr viel mehr Daten enthält, könnte sich der Overhead, welcher durch den Netzwerkverkehr entsteht, im Verhältnis zu dem lokalen Berechnen wieder lohnen. Jedoch muss dies in einem folgenden Versuch überprüft werden, wenn auch solch ein Datensatz zur Verfügung steht.

Literaturverzeichnis