Universität Bielefeld
11. Juni 2022

Erweiterung der de.NBI Cloud um gewichtete GPU- und Speicher-Filter zur optimierten Nutzung von Grafikprozessoren

Henry Spanka

Erstgutachter: Herr Prof. Dr. Alexander Sczyrba

Zweitgutachter: Herr M.Sc. Alex Walender

Bachelorarbeit

Studiengang: Kognitive Informatik

Matrikelnummer: 3081606

Technische Fakultät

Universität Bielefeld

Eingereicht am: 11. Juni 2022

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und gelieferte Datensätze, Zeichnungen, Skizzen und graphische Darstellungen selbstständig erstellt habe. Ich habe keine anderen Quellen als die angegebenen benutzt und habe die Stellen der Arbeit, die anderen Werken entnommen sind - einschließlich verwendeter Tabellen und Abbildungen - in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht.

Universität Bielefeld

Bielefeld, den 11. Juni 2022

Henry Spanka

Motivation

Das de.NBI (Deutsches Netzwerk für Bioinformatik) stellt eine Cloud-Lösung für Bioinformatik an mehreren Standorten in Deutschland zur Verfügung [1]. Einer dieser Standorte ist am CeBiTec an der Universität Bielefeld angesiedelt. Im Fokus steht dabei mikrobielle Genetik und Metagenomik. Dabei werden große Datenmengen in der Cloud analysiert wie zum Beispiel die Oxford Nanopore Sequenzdaten.

In Bielefeld werden dafür neben standardisierten Instanzen, auch sogenannte „high memory“ Instanzen mit mehr lokalen Speicherplatz (bis zu 2 TB RAM und 5 TB Festplatte) angeboten. Für hardware-beschleunigte Instanzen, die Anwendungen im Bereich maschinelles Lernen ausführen, werden spezielle virtuelle Maschinen mit Grafikkarten (GPUs) bereitgestellt.

Derzeit findet keine Priorisierung der Hardwareanforderungen dieser Instanzen statt. Dabei kann es zu einer nicht optimalen Verteilung der Hardware-Ressourcen kommen. Standardisierte oder high memory Instanzen werden auf Hypervisor erstellt auf denen Grafikprozessoren zur Verfügung stehen, wodurch nicht mehr ausreichend CPU-, RAM- und Festplattenressourcen vorhanden sind um GPU-basierte Instanzen auszuführen.

Ziel dieser Arbeit ist es die Zuweisung der Instanzen an Hypervisor entsprechend zu optimieren, damit möglichst viele Instanzen ausgeführt werden können.

Inhaltsverzeichnis

1 Grundlagen

1.1 Virtuelle Maschine

Eine virtuelle Maschine (VM) ist eine Umgebung in der CPU, Speicher, Netzwerkschnittstelle und Storage von einem physischen Hardware-System bereitgestellt werden und durch einen Hypervisor (Software) provisioniert und getrennt werden [2].

Kernel-based Virtual Machine (KVM) ist eine Open-Source Virtualisierungstechnologie die im Linux-Kernel integriert ist. Dadurch kann ein Hypervisor mehrere isolierte virtuelle Umgebungen ausführen. KVM wird von allen gängigen Linux-Distributionen und X86-Hardware unterstützt [3].

1.2 Ansible

Ansible ist ein Softwaretool zur plattformübergreifenden Automatisierung von Abläufen. Primär wird es von IT Experten verwendet um das Ausrollen von Applikationen, Updates oder Konfigurationen zu vereinfachen, dabei ist kein Agent auf den Ziel-Computern notwendig. Die Automatisierung wird in Form von Skripten geschrieben, was die Versionierung erleichtert und Anwenderfreundlich ist. Dies sind einfache Dateien im YAML-Format und das ausführen idempotent. Das bedeutet, dass ein mehrfaches Ausführen eines Playbooks keine negativen Effekte hat wenn das System bereits korrekt konfiguriert ist [4].

1.3 OpenStack

OpenStack ist eine Cloud Software die eine große Anzahl von Compute-, Storage-, und Netzwerkressourcen verwaltet. Durch eine Web-Anwendung kann die Cloud verwaltet und Ressourcen provisioniert werden [5].

OpenStack besteht aus einzelnen Projekten die bestimmte Dienste wie Computing, Networking oder Storage anbieten und API Schnittstellen (Application Programming Interface) zur Verfügung stellen [6].

OpenStack wurde im Juli 2010 gemeinsam von RackSpace und NASA als Open-

Source Projekt vorgestellt um NASA's Nebula Platform und Rackspace's Cloud Dateispeicherdienst zu vereinen. Mittlerweile beteiligen sich viele namenhafte Firmen an der Weiterentwicklung der einzelnen Projekte [7].

Die Projekte sind modular aufgebaut und interagieren untereinander über APIs. Zu den wichtigsten Projekten zählen [7]:

Nova: Nova ist der primäre Compute-Dienst und für das Planen, Erstellen und Löschen von virtuellen Maschinen zuständig. Nova unterstützt neben dem KVM-Hypervisor auch Hyper-V, VMware ESXi oder Xen.

Glance: Glance ist ein Abbild-Dienst der für das Hochladen, die Verwaltung und das Löschen von Abbildern zuständig ist. Glance unterstützt unter anderem Ceph, NFS oder Swift als Backend [8].

Neutron: Neutron verwaltet die Netzwerkinfrastruktur und stellt die Verbindung zwischen Instanzen sicher. Dazu zählen zum Beispiel Software-Defined-Networking (SDN) Technologien wie Open vSwitch (OVS) oder Open Virtual Network (OVN).

Cinder: Cinder ist eine Block Storage Komponente und erstellt, verwaltet und löscht persistente Block Devices. Diese können in die Instanzen eingehängt werden.

Swift: Swift ist eine weitere Storage Komponente das hochverfügbaren Speicher ähnlich wie Amazon S3 anbietet. Dabei können ungeordnete Daten-Objekte über REST-APIs verwaltet werden.

Keystone: Keystone stellt Authentifizierungs- und Autorisierungsdienste für Nutzer bereit. Daneben ist auch eine Anbindung an externe Identitätsdienste wie LDAP oder Active Directory möglich.

Weitere Dienste sind unter anderem Horizon (Web Frontend), Designate (DNS Dienst), Ironic (Bare Metal) oder Ceilometer (Monitoring) [9].

OpenStack Kolla ist ein weiteres Projekt welches Container und Tools zum Ausrollen einer OpenStack Cloud bereitstellt. Kolla-Ansible ist dabei ein Unterprojekt, dass das automatische Ausrollen der Container mit Ansible erlaubt. Dies ist für Produktive-, aber auch für Entwicklungsumgebungen vom Vorteil, da es ein reproduzierbares und automatisches Deployment ermöglicht [10].

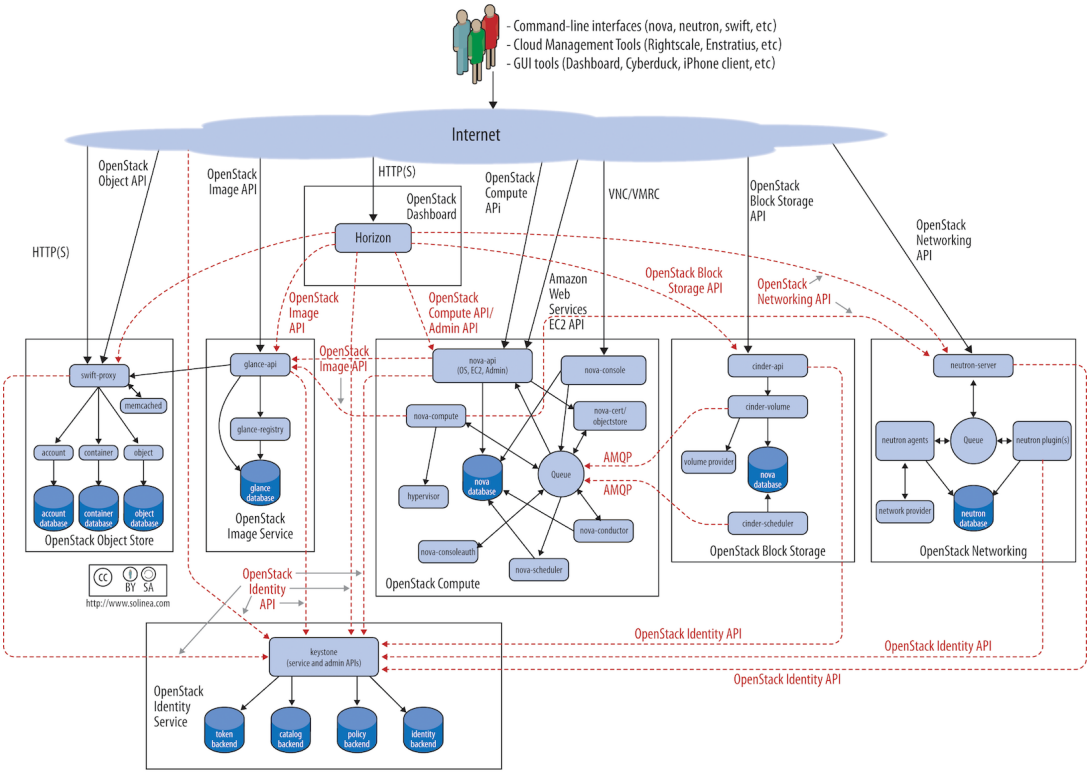

Abbildung 1.1: OpenStack Architektur [11]

1.4 Docker

Docker ist eine offene Plattform für die Entwicklung, Verteilung und Ausführung von Applikationen. Die Applikationen werden dabei in einer isolierten Umgebung ausgeführt die bereits alle Abhängigkeiten beinhaltet. Weiter benötigen Docker-Container weniger Ressourcen als KVM-basierte Umgebungen und sind daher besonders für die Isolation von kleinen Applikationen geeignet. Zum Beispiel kann ein Docker Container mit einem Ubuntu Abbild unter Windows gestartet werden und dort Linux-Applikationen ausgeführt werden. OpenStack Kolla nutzt ebenfalls Docker für die Bereitstellung der Container [12].

1.5 PyScaffold

PyScaffold ist ein Projekt-Generator für Python. Damit können Python-Pakete erzeugt und mittels dem Python Paketmanager *pip* installiert werden. Projekte lassen sich automatisch mit Git versionieren und die generierten Artefakte, sogenannte *Python Wheels*, mit einer Versionsnummer versehen [13].

2 Entwicklungsumgebung

Wie bereits im vorherigen Kapitel beschrieben bietet sich Kolla zum Deployment von OpenStack Umgebungen an. Dabei können die einzelnen Dienste modular bereitgestellt werden [14].

Für die Entwicklung von OpenStack Nova Filter und Weigher werden dabei folgende Dienste benötigt:

- Glance
- Horizon
- Keystone
- Neutron
- Nova

Kolla installiert und konfiguriert in der Standardkonfiguration alle oben genannten notwendigen Dienste. Darüber hinaus lassen sich alle Konfigurationsdateien anpassen und eigene Pakete installieren welche für die Erweiterung des Nova Schedulers notwendig sind [14].

2.1 Anforderungen

Kolla setzt auf Ansible Playbooks für das automatische Deployment.

Es werden die folgenden Betriebssysteme unterstützt [15]:

- CentOS Stream 8
- Debian Bullseye (11)
- openEuler 20.03 LTS SP2
- RHEL 8 (veraltet)
- Rocky Linux 8
- Ubuntu Focal (20.04)

Daneben bietet Kolla die Abbilder der OpenStack Dienste in mehreren Varianten an [15]:

- CentOS
- Debian
- RHEL
- Ubuntu

Als Betriebssystem für die Entwicklungsumgebung im Rahmen dieser Bachelorarbeit wurde Ubuntu Focal (20.04) zusammen mit Ubuntu Abbilder der OpenStack Dienste genutzt. Kolla und die OpenStack Dienste basieren auf dem Yoga Release der am 30. März 2022 veröffentlich wurde [16].

Für eine **all-in-one** Standard-Installation von OpenStack werden mindestens **2 Netzwerkschnittstellen**, **8 GB RAM** und **40 GB Festplattenspeicher** benötigt [17].

Zur sinnvollen Erweiterung des Nova Schedulers werden jedoch mindestens zwei Nova Compute Nodes benötigt. Ein Weighing einer einzelnen Node wäre trivial. Als Entwicklungsumgebung wird daher ein **multinode** Deployment bestehend aus einer Controller-, mindestens zwei Compute- und einer Deployment Node verwendet. Alle Nodes können dabei als Virtuelle Maschinen betrieben werden. Die Anforderungen des Controllers orientieren sich dabei an denen einer **all-in-one** Installation. Die Compute Nodes müssen den individuellen Anforderungen genügen und benötigen ebenfalls zwei Netzwerkschnittstellen.

OpenStack Kolla nutzt dabei eine Schnittstelle für die Verwaltung und Konfiguration der Cloud Dienste. Die weitere Netzwerkschnittstelle wird von OpenStack Neutron zur Bereitstellung von Netzwerkdiensten der Instanzen benötigt.

Für diese Bachelorarbeit wurden Droplets (Virtuelle Maschinen) vom Anbieter DigitalOcean genutzt. Diese werden standardgemäß bereits mit zwei Netzwerkschnittstellen ausgeliefert weshalb hier keine weitere Konfiguration notwendig ist.

3 OpenStack Nova

Das OpenStack Projekt Nova (Compute) ist für die Provisionierung von virtuellen Instanzen zuständig. Dabei sind die Dienste **Keystone**, **Glance**, **Neutron** und **Placement** erforderlich mit denen Nova interagiert [18].

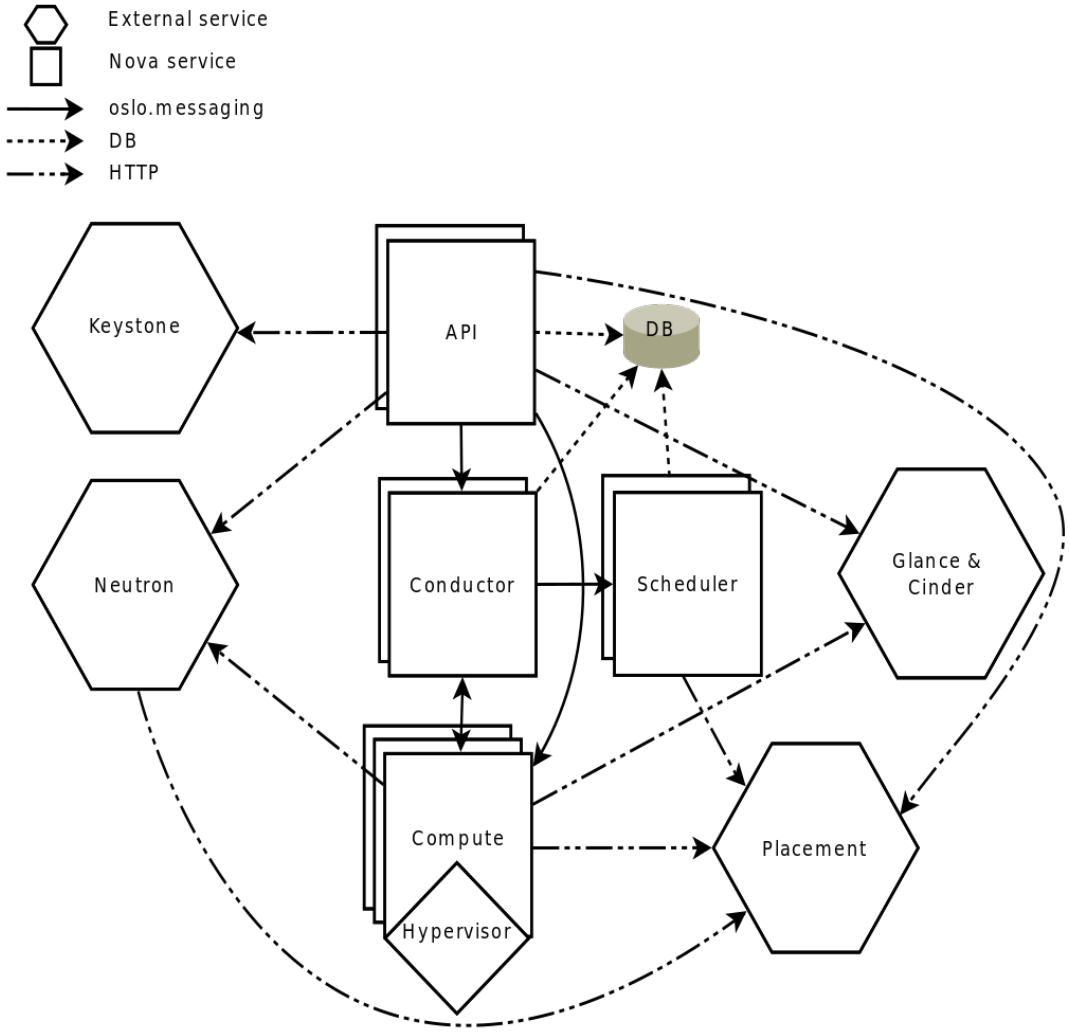

Abbildung 3.1: Nova System Architektur [19]

Neue Instanzen können dabei über die Nova API erstellt werden. Über eine Message Queue wird die **Request Specification** dann an den Nova Conductor gesendet. Dieser koordiniert die einzelnen Aufgaben und agiert als Datenbank Proxy [19]. Der Scheduler ist dabei für die Entscheidung zuständig auf welcher Compute Node eine Instanz allokiert werden kann und welche Node dafür am besten geeignet ist. Der Placement Dienst allokiert dabei die Ressourcen und bekommt von den Compute Nodes Informationen

über die aktuelle Hardwarespezifikation (Traits) und Auslastung (Resources).

3.1 Nova Scheduler

Nova Scheduler implementiert den **Filter Scheduler**, der die Hosts anhand von Eigenschaften filtert und gewichtet [20].

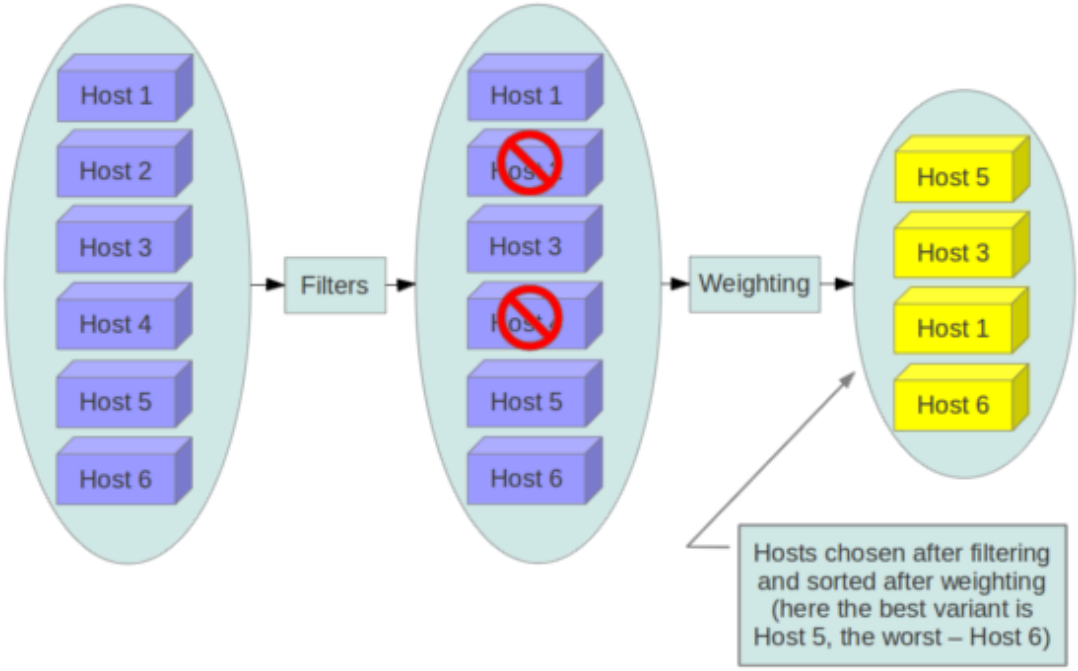

Abbildung 3.2: Nova Filter Workflow [20]

3.1.1 Filter

In einem ersten Schritt wird geprüft welche Hosts den Anforderungen genügen. Alle konfigurierten **Filter** müssen dabei den Host akzeptieren [20].

Sei F die Menge aller Filter, dann muss gelten:

$$\forall f \in F : f(Host, RequestSpec) = True$$

Andernfalls wird der Host nicht weiter für das Scheduling der Instanz betrachtet.

3.1.2 Weigher

Danach werden die akzeptierten Hosts mit sogenannten **Weigher** gewichtet und sortiert.

Das Gewicht eines Hosts berechnet sich aus [21]:

$$weight = m_1 * norm(w_1) + m_2 * norm(w_2) + ...$$

Dabei ist w_i das Gewicht und m_i der Multiplikator des **Weigher** i. Das Gewicht w_i wird dabei zwischen 0 und 1 normalisiert. Über den Multiplikator lässt sich der Einfluss bestimmter **Weigher** ändern. Ist $w_i = 0$ oder $m_i = 0$, dann nimmt der Weigher keinen Einfluss auf die Gewichtung.

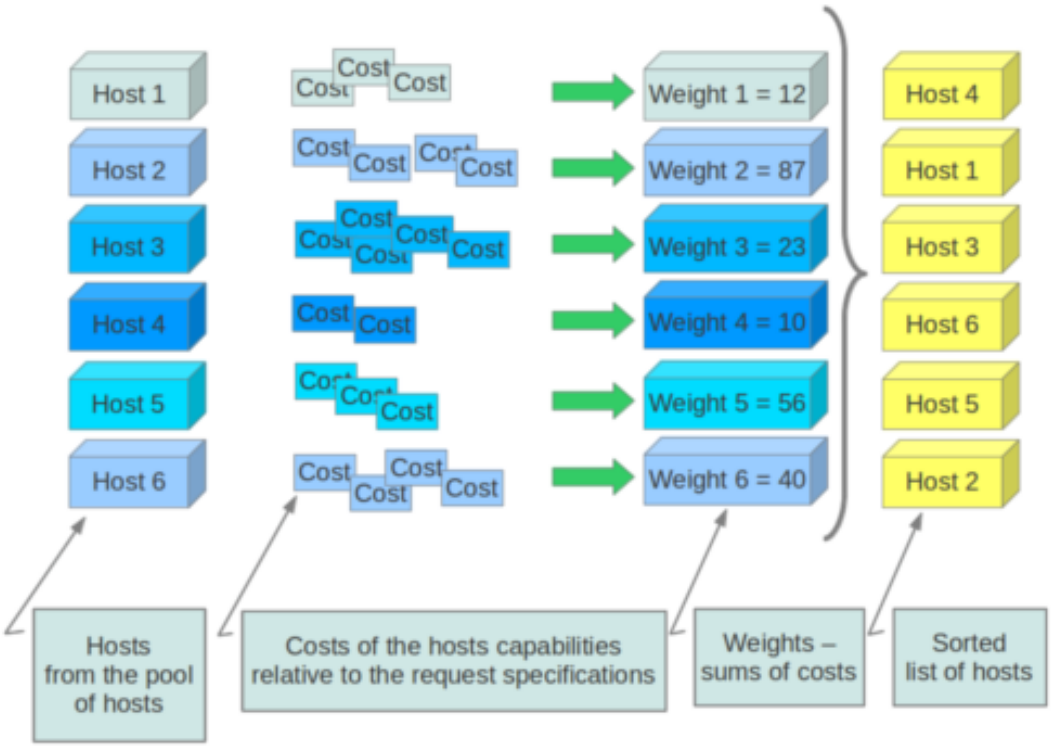

Abbildung 3.3: Nova Weighting Hosts [20]

3.1.3 Placement

Nach erfolgter Gewichtung versucht der Nova Scheduler die Ressourcen des **Request Spec** auf dem Host mit dem höchsten Gewicht zu allokieren. Die einzelnen Ressourcetypen werden dabei als sogenannte **Classes** verwaltet [22]. Dies sind zum Beispiel **VCPU**, **MEMORY_MB** und **DISK_GB**. Erst wenn diese Allokierung

erfolgreich ist, wird die Instanz auf dem Host erstellt. Schlägt die Allokierung oder Erstellung, sogenanntes **Spawning**, fehl wird versucht die Instanz auf dem Host mit dem nächstkleineren Gewicht zu erstellen.

3.1.4 Metadata und Extra Specs

Nova bietet zusätzlich die Möglichkeit Metadaten an Hosts oder sogenannte Extra Specs an Flavor zu setzen. Ein Flavor definiert dabei eine bestimmte Hardwarekonfiguration einer Instanz [23]. Die Daten werden dann vom Nova Scheduler oder eigenen Filter und Weigher genutzt. Das Extra Spec **quota:vif_outbound_peak=65536** beschränkt zum Beispiel die maximale ausgehende Netzwerk-Übertragungsrate auf 512 Mbit/s.

3.1.5 Erweiterung des Nova Schedulers

Im Beispiel ist ein Filter implementiert, der es nur erlaubt eine einzelne Instanz pro **Request Spec** zu erstellen.

```
from nova.objects.request_spec import RequestSpec
from nova.scheduler.host_manager import HostState
from nova.scheduler import filters


class SingleInstancePerRequestFilter(filters.BaseHostFilter):
    RUN_ON_REBUILD = False


    def host_passes(
        self, host_state: HostState, request_spec: RequestSpec
    ):
        # Only allow user to schedule a single instance at a time
        return request_spec.num_instances == 1
```

Eine Implementierung eines Weighers der einen Host anhand seiner Instanzen gewichtet wird im folgenden Beschrieben. Dadurch werden Hosts mit mehr Instanzen priorisiert (Stacking). Über die Metadata der Host Aggregates (instance_count_weight_multiplier) kann dann die Wichtigkeit des Weighers bestimmt werden. Ein Aggregate ist dabei eine

Gruppierung von Hosts die gemeinsame Eigenschaften besitzen. Ein negatives Gewicht führt dabei zu einem Spreading, daher die Instanzen werden möglichst breit auf den Hosts verteilt.

```
from nova.scheduler import weights
from nova.scheduler.host_manager import HostState
from nova.objects.request_spec import RequestSpec

class InstanceCountWeigher(weights.BaseHostWeigher):
    def weight_multiplier(self, host_state: HostState):
        """Override the weight multiplier."""
        return utils.get_weight_multiplier(
            host_state, 'instance_count_weight_multiplier',
            1.0)

    def _weigh_object(
        self, host_state: HostState, request_spec: RequestSpec
    ):
        return host_state.num_instances # Stack based on Instances
```

Darüber hinaus kann ebenfalls im Filter und Weigher auf Metadata und Extra Specs zugegriffen werden. Ein Zugriff auf das Attribut **customscope:someattribute** ist Beispielsweise so möglich:

```
value = request_spec.flavor.extra_specs.get('customscope:someattribute')
```

Seit dem OpenStack Release Ussuri und der Compute API Version 2.86 unterstützt Nova die Validierung von Flavor Extra Specs [24]. Horizon nutzt aktuell eine ältere Version der API weshalb dort die Validierung nicht erfolgt. Die Validierung muss aktiv vom Client angefordert werden in dem eine entsprechende API Version genutzt wird. Mit der OpenStack CLI ist das Beispielsweise wie folgt möglich:

```
openstack --os-compute-api-version 2.86 flavor set \
--property customattribute:someattribute=optionA $FLAVOR
```

Für die Validierung von eigenen Scopes und Attributen kann ein Validator implementiert werden [20]. Dieser muss unter dem Python Entrypoint **nova.api.extra_spec_validator** registriert werden und für den Dienst Nova API sichtbar sein.

```
from nova.api.validation.extra_specs import base


def register():
    validators = [
        base.ExtraSpecValidator(
            name='customscope:someattribute',
            description='Custom Enum Attribute',
            value={
                'type': str,
                'enum': [
                   'optionA',
                   'optionB'
                ]
            }
        ),
    ]

    return validators
```

4 Allokierung und Gewichtung von GPUs

In der de.NBI Cloud werden GPUs über die **PCI Passthrough** Funktionalität des Hypervisors in die Instanz durchgereicht. Dabei werden bestimmte Flavors mit Extra Specs versehen um PCI Ressourcen anzufordern:

```
openstack flavor set \
--property "pci_passthrough:quadro"="quadro:2" $FLAVOR
```

Im obigen Beispiel werden zwei PCI Geräte vom Typ Quadro angefordert. Der Alias muss dann zusätzlich in der Konfiguration der Nova Compute und API Dienste gesetzt werden [25]. Vendor ID, ProductID und DeviceType sind dabei geräteabhängig und können mit dem Befehl `lspci -nn` ausgelesen werden. Dabei ist es erforderlich, dass der Gerätetyp auch zu der Passthrough Whitelist des Hosts hinzugefügt wird.

```
[pci]
alias = {
    "vendor_id":"10de",
    "product_id":"1cb3",
    "device_type":"type-PCI",
    "name":"quadro"
}
passthrough_whitelist = { "vendor_id": "10de", "product_id": "1cb3" }
```

Der von Nova mitgelieferte PCIPassthrough Filter sorgt dafür, dass nur Hosts für das Scheduling in Betracht gezogen werden, welche auch die angeforderten PCI Ressourcen bereitstellen können [26]. Eine Gewichtung wird dabei nicht vorgenommen.

4.1 Scheduling in virtuellen Umgebungen

Aufgrund der Besonderheit des Schedulings von GPUs ist es nicht ohne weiteres möglich dies in einer virtuellen Testumgebung abzubilden. Der Aufbau einer Testumgebung basierend auf physischer Hardware wäre mit entsprechenden Aufwand und Kosten

verbunden und erschwert den schnellen Aufbau einer Testumgebung. Wünschenswert wäre daher eine Implementierung die ein Scheduling in einer virtuellen Umgebung ermöglicht.

4.2 Virtueller GPU Filter

Zur Filterung der Hosts wird ein neuer Filter **GpuVirtualFilter** implementiert.

Tabelle 4.1: Attribute des GPU Filters

Attribut	Typ	Beschreibung
enabled	bool	Aktiviert GPU Filter
model	string	GPU Name/Alias
count	integer	Anzahl an GPUs

Die neu definierten Attribute in Tabelle 4.1 werden dabei vom Filter unter dem Scope **gpu_filter** genutzt.

Jeder Host mit verfügbaren Grafikeinheiten wird in ein Aggregate gruppiert und mit den Metadaten **model** (GPU Typ) und **count** (Anzahl vorhandender GPUs) versehen.

Ein Flavor kann GPU Ressourcen anfordern in dem es die Extra Specs setzt:

1. **enabled:** Aktivierung des GPU Filter
2. **model:** Angeforderter GPU Typ
3. **count:** Anzahl angeforderter GPUs

Der Filter akzeptiert einen Host wenn mindestens eine Aussage wahr ist:

1. Der Filter ist deaktiviert, d.h. Extra Spec **enabled** != True.
2. GPU Typ (**model**) von Flavor und Host identisch und die Summe von Extra Spec **count** aller Instanzen und RequestSpec ist kleiner oder gleich der Host Metadata **count**.

Physische GPUs sind dabei nicht notwendig. Die Filterung wird anhand von Attributen der Flavor und Host Aggregates vorgenommen. Über die Kombination des Flavors mit den in Kapitel 4 genannten Extra Specs werden dann die virtuellen Ressourcen an die physische Hardware gebunden.

4.3 Gleichzeitiges Scheduling mehrerer Instanzen

Der Nova Scheduler lässt sich unabhängig von einander mehrfach starten. Dies dient zum Beispiel der Lastverteilung und Redundanz. Dabei ist nicht garantiert, dass Instanzen sequenziell voneinander scheduled werden [27]. Dadurch kann es passieren, dass ein Filter einen Host akzeptiert, jedoch vor der Allokierung der Instanz ein weiterer Nova Scheduler Prozess den selben Filter durchläuft und eine Allokierung auf dem selben Host vornimmt. Wenn der Filter in einer Entwicklungsumgebung ohne die Verknüpfung mit PCI Requests (Kapitel 4) ausgeführt wird, führt dies zu Inkonsistenzen, da keine wirkliche Allokation von Ressourcen stattfindet. Im folgenden wird ein Ansatz vorgestellt, der auf einen virtuellen Filter verzichtet und stattdessen die Nova Placement API zur Allokierung nutzt.

4.4 GPU Ressourceklasse

Wie bereits in Kapitel 3.1.3 beschrieben, wird die Allokierung von Hardwareressourcen vom Nova Placement Dienst verwaltet. Nova bietet dabei die Möglichkeit der Erweiterung um eigene Ressourceklassen mit dem Präfix **CUSTOM_** [28].

Zur Erweiterung der Resource Provider werden standardgemäß vom Nova Compute Dienst Konfigurationsdateien im YAML-Format in dem Pfad **/etc/nova/provider_config/** gelesen [29].

Auf den Compute Nodes mit Unterstützung für GPU Ressourcen wird eine neue Ressourceklasse mit dem Namen **CUSTOM_GPU** erstellt. Im Beispiel nehmen wir an, dass maximal 2 GPUs zur Verfügung stehen (total). Das Modell (Quadro) wird dabei über ein **Trait** abgebildet. Dies sind Eigenschaften der Ressourceklasse. Die weiteren Parameter sind für das GPU Scheduling nicht notwendig und können in der OpenStack Dokumentation eingesehen werden [30]. Auch hier ist keine Präsenz von PCI Hardware notwendig. Die Verknüpfung mit GPUs erfolgt, identisch zum GPU Filter (Kapitel 4.2), über PCI Requests mit Hilfe von Extra Specs des Flavors (Kapitel 4).

Ein weiterer Vorteil ist, dass die Nutzung der GPUs nun in einer eigenen Ressourceklasse überwacht wird und über die Placement API ausgelesen werden kann. Daneben ist die Allokierung auch in Testumgebungen atomar und konsistent.

```
# /etc/nova/provider_config/gpu.yaml
meta:
  schema_version: '1.0'
providers:
  - identification:
      uuid: $COMPUTE_NODE
    inventories:
      additional:
        - CUSTOM_GPU:
            total: 2
            reserved: 0
            min_unit: 1
            max_unit: 10
            step_size: 1
            allocation_ratio: 1.0
    traits:
      additional:
        - 'CUSTOM_GPU_QUADRO'
```

Die Ressourceklasse kann dann in den Extra Specs des Flavors angefordert werden.

Dafür wird ein Attribut im Format resources:$CUSTOM_RESOURCE_CLASS=$N hinzugefügt [31]. Zur Anforderung von 2 GPUs nutzen wir daher resource:CUSTOM_GPU=2.

Die Ressourceklasse unterscheidet nicht zwischen verschiedenen GPU-Typen. Die Erzwingung eines bestimmten Typen (hier Quadro) ist mit dem Extra Spec trait:CUSTOM_GPU_QUADRO=required möglich [32].

4.5 Gewichtung

In den Kapitel 4.2 und 4.4 wurden zwei verschiedene Ansätze zur Filterung von GPU Ressourcen vorgestellt. Dabei wurde bisher keine Gewichtung berücksichtigt.

Dazu wird ein neuer GPU Weigher **GpuVirtualWeigher** implementiert.

Tabelle 4.2: Attribute des GPU Weighers

Attribut	**Typ**	**Beschreibung**
enabled	bool	Aktiviert GPU Weigher
mode	enum (stack,spread)	Art der Gewichtung
resource	string	Resource/Extra Spec der gewichtet werden soll

Die in Tabelle 4.2 definierten Attribute werden vom Weigher unter dem Scope **gpu_weigher** genutzt. Ein zu gewichtenes Flavor muss dabei alle Attribute definieren.

Ist der Weigher deaktiviert, d.h. Extra Spec **enabled** != True, dann wird der Weigher ignoriert ($w = 0$). Andernfalls summiere den Wert des Extra Spec **resource** aller Instanzen des Hosts. Dieser gibt an wie viele GPUs aktuell auf dem Host verwendet werden.

Über das Attribut **mode** kann ein Stacking oder Spreading der Instanzen erreicht werden. Beim Stacking ist dabei das Gewicht identisch zu der Anzahl der genutzten GPUs. Ist ein Spreading gewünscht, dann wir das Gewicht negiert. Dadurch wird das Gewicht eines Hosts dann reduziert je mehr GPUs des Hosts bereits verwendet werden. Dies kann auch genutzt werden um Instanzen ohne GPU Anforderung von GPU-fähigen Hosts fernzuhalten.

Tabelle 4.3: Beispiel einer Gewichtung

Host	**Genutzte GPUs**	**Stacking**		**Spreading**	
		w	$norm(w)$	w	$norm(w)$
compute1	3	3	1	-3	0
compute2	2	2	0.66	-2	0.33
compute3	1	1	0.33	-1	0.66
compute4	0	0	0	0	1

Durch die Normierung werden die Hosts zwischen 0 und 1 gewichtet. Über den Multiplikator kann dann das Gewicht des Weighers verstärkt oder abgeschwächt werden.

5 Stacking von Arbeitsspeicher

Zusätzlich zur Gewichtung von GPUs ist ein Stacking von Instanzen auf Basis des Arbeitsspeichers wünschenswert. Der Nova RAM Weigher verteilt die Instanzen standardgemäß so breit wie möglich [21]. Dies führt bei einer hohen Auslastung der OpenStack Cloud dazu, dass kein oder wenige Hosts Instanzen mit viel Arbeitsspeicher zur Verfügung stellen können.

Wie in Kapitel 3.1.2 beschrieben wird jeder Weigher mit einem Multiplikator multipliziert. Dieser lässt sich entweder in der Nova Konfiguration oder über Host Aggregate Metadaten setzen. Über einen negativen Wert kann dabei ein Stacking anstatt Spreading erzielt werden [20].

5.1 Kombination von Multiplikatoren

Die optimale Allokierung von Instanzen auf Hosts lässt sich nicht immer vorhersagen. Jeder Host kann Arbeitsspeicher bereitstellen, jedoch verfügt nicht jeder Host über GPUs. Daher betrachten wir GPUs generell als die „wertvollere“ Ressource.

Über Multiplikator kann der GPU Weigher gegenüber dem RAM Weigher priorisiert werden. Setzen wir den Multiplikator des GPU Weighers auf 2.0 und den des RAM Weighers auf -1.0, dann wird die GPU Ressource doppelt gewichtet. In Kombination mit anderen Weigher wie CPU oder Festplattenspeicher kann dann eine optimale Auslastung erzielt werden.

6 Zusammenfassung und Ausblick

Ziel der Bachelorarbeit war die Optimierung der Allokierung von Grafikprozessoren in der de.NBI Cloud. Ein besonderes Augenmerk wurde auf die Unabhängigkeit von physischer Hardware gelegt, um eine Entwicklung in Testumgebungen zu ermöglichen. Dabei wurden Grundlagen zur Entwicklung von Filtern und Weigher zur Optimierung von Ressourcen aufgezeigt und zwei Lösungsansätze vorgestellt. Die Nutzung einer Ressourcenklasse in Kombination mit einem Weigher hat dabei alle Anforderungen erfüllt und lässt sich leicht auch auf andere Ressourcen übertragen.

Die Bachelorarbeit dient dabei auch als Grundlage für weitere Anpassungen des Scheduling der de.NBI Cloud. Besonders ist aktuell eine Einschränkung der Anzahl von GPUs pro Projekt nicht möglich. Dadurch kann ein Nutzer der Cloud mehr GPUs anfordern als gewünscht. Nova unterstützt keine Erweiterung der Projekt Quotas, bietet aber eine experimentelle Implementierung der Unified Limits an [33]. Die de.NBI Cloud nutzt zum heutigen Stand den Train Release und unterstützt keine Unified Limits, die erst im Yoga Release eingeführt wurden.

Eine spätere Implementierung von Filtern oder Ressourceklassen in Kombination mit Unified Limits wäre wünschenswert um die Anzahl der GPUs pro Nutzer einzuschränken. Dabei bietet sich auch eine Erweiterung des Perun Keystone Adapters der de.NBI Cloud an, um Unified Limits mit der Identitäts- und Zugriffsverwaltung Perun zu synchronisieren.

Quellen

[1] de.NBI. de.NBI Cloud, 2022. URL https://cloud.denbi.de/about/bielefeld/. [Online; abgerufen am 11-Mai-2022].

[2] Red Hat, Inc. Virtuelle Maschine, 2019. URL https://www.redhat.com/de/topics/virtualization/what-is-a-virtual-machine. [Online; abgerufen am 11-Mai-2022].

[3] Red Hat, Inc. Was ist KVM?, 2018. URL https://www.redhat.com/de/topics/virtualization/what-is-KVM. [Online; abgerufen am 11-Mai-2022].

[4] Red Hat, Inc. What is Ansible?, o. J. URL https://opensource.com/resources/what-ansible. [Online; abgerufen am 11-Mai-2022].

[5] Open Infrastructure Foundation. Documentation for Yoga (April 2022), 2022. URL https://docs.openstack.org/yoga/. [Online; abgerufen am 11-Mai-2022].

[6] Red Hat, Inc. Understanding OpenStack, 2021. URL https://www.redhat.com/en/topics/openstack. [Online; abgerufen am 11-Mai-2022].

[7] Canonical Ltd. What is OpenStack?, o. J. URL https://ubuntu.com/openstack/what-is-openstack. [Online; abgerufen am 11-Mai-2022].

[8] Open Infrastructure Foundation. Glance - Image service, 2020. URL https://docs.openstack.org/kolla-ansible/latest/reference/shared-services/glance-guide.html. [Online; abgerufen am 11-Mai-2022].

[9] Open Infrastructure Foundation. OpenStack Cloud Platform Software, 2021. URL https://www.openstack.org/software/. [Online; abgerufen am 11-Mai-2022].

[10] Open Infrastructure Foundation. Welcome to Kolla's documentation!, 2022. URL https://docs.openstack.org/kolla/yoga/. [Online; abgerufen am 11-Mai-2022].

[11] Open Infrastructure Foundation. Arch Design, 2018. URL https://docs.openstack.org/arch-design/design.html. [Online; abgerufen am 11-Mai-2022].

[12] Docker Inc. Docker overview, 2022. URL https://docs.docker.com/get-started/overview/. [Online; abgerufen am 11-Mai-2022].

[13] Florian Wilhelm. Pyscaffold, 2022. URL https://pypi.org/project/PyScaffold/. [Online; abgerufen am 11-Mai-2022].

[14] Open Infrastructure Foundation. Kolla Ansible, 2022. URL https://opendev.org/openstack/kolla-ansible. [Online; abgerufen am 05-Juni-2022].

[15] Open Infrastructure Foundation. Kolla Ansible Support Matrix, 2022. URL https://docs.openstack.org/kolla-ansible/yoga/user/support-matrix. [Online; abgerufen am 05-Juni-2022].

[16] Open Infrastructure Foundation. Openstack Yoga Release, 2022. URL https://releases.openstack.org/yoga/index.html. [Online; abgerufen am 05-Juni-2022].

[17] Open Infrastructure Foundation. Kolla Quick Start, 2022. URL https://docs.openstack.org/kolla-ansible/yoga/user/quickstart.html. [Online; abgerufen am 05-Juni-2022].

[18] Open Infrastructure Foundation. Docs: Openstack Nova, 2021. URL https://docs.openstack.org/nova/yoga/. [Online; abgerufen am 06-Juni-2022].

[19] Open Infrastructure Foundation. Nova System Architecture, 2022. URL https://docs.openstack.org/nova/yoga/admin/architecture.html. [Online; abgerufen am 06-Juni-2022].

[20] Open Infrastructure Foundation. Compute Schedulers, 2021. URL https://docs.openstack.org/nova/yoga/admin/scheduling.html. [Online; abgerufen am 06-Juni-2022].

[21] Open Infrastructure Foundation. Scheduler/Normalized Weights, o. J. URL https://wiki.openstack.org/wiki/Scheduler/NormalizedWeights. [Online; abgerufen am 06-Juni-2022].

[22] Open Infrastructure Foundation. Placement, 2019. URL https://docs.openstack.org/placement/yoga/. [Online; abgerufen am 06-Juni-2022].

[23] Open Infrastructure Foundation. Docs: Flavors, 2021. URL https://docs.openstack.org/nova/yoga/user/flavors.html. [Online; abgerufen am 06-Juni-2022].

[24] Open Infrastructure Foundation. Nova API Version Request, 2021. URL https://github.com/openstack/nova/blob/stable/yoga/nova/api/openstack/api_version_request.py#L234. [Online; abgerufen am 06-Juni-2022].

[25] Open Infrastructure Foundation. Attaching physical PCI devices to guests, 2022. URL https://docs.openstack.org/nova/yoga/admin/pci-passthrough.html. [Online; abgerufen am 07-Juni-2022].

[26] Open Infrastructure Foundation. Nova PCI Passthrough Filter, 2017. URL https://github.com/openstack/nova/blob/stable/yoga/nova/scheduler/filters/pci_passthrough_filter.py. [Online; abgerufen am 07-Juni-2022].

[27] Open Infrastructure Foundation. Nova Scheduling, 2021. URL https://docs.openstack.org/nova/yoga/reference/scheduling.html. [Online; abgerufen am 09-Juni-2022].

[28] Open Infrastructure Foundation. Managing Resource Providers Using Config Files, 2019. URL https://docs.openstack.org/nova/yoga/admin/managing-resource-providers.html. [Online; abgerufen am 09-Juni-2022].

[29] Open Infrastructure Foundation. Nova Configuration Options, 2019. URL https://docs.openstack.org/nova/yoga/configuration/config.html. [Online; abgerufen am 09-Juni-2022].

[30] Open Infrastructure Foundation. Provider Configuration File, 2019. URL https://specs.openstack.org/openstack/nova-specs/specs/ussuri/approved/provider-config-file.html. [Online; abgerufen am 09-Juni-2022].

[31] Open Infrastructure Foundation. Allow custom resource classes in flavor extra specs, 2017. URL https://specs.openstack.org/openstack/nova-specs/specs/pike/implemented/custom-resource-classes-in-flavors.html. [Online; abgerufen am 09-Juni-2022].

[32] Open Infrastructure Foundation. Request traits in Nova, 2018. URL https://specs.openstack.org/openstack/nova-specs/specs/queens/implemented/request-traits-in-nova.html. [Online; abgerufen am 09-Juni-2022].

[33] Open Infrastructure Foundation. Unified Limits, 2022. URL https://docs.openstack.org/keystone/latest/admin/unified-limits.html. [Online; abgerufen am 11-Juni-2022].

Abbildungsverzeichnis

Tabellenverzeichnis