FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	GPU cloud
Student:	Bc. Michal Číla
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Systémové programování
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce zimního semestru 2019/20

Pokyny pro vypracování

1. Seznamte se s platformou cloudového řešení Openstack a možnostmi napojení GPU na tuto platformu.
2. Prozkoumejte možnosti využití Kubernetes orchestrace kontejnerů v rámci Openstacku a srovnejte efektivitu s VM (virtual machine).
3. Implementujte GPU cloud řešení na platformě Openstack pro účely výzkumu v oblasti AI a strojového učení.
4. Do uživatelského rozhraní Openstacku zvaného Horizon navrhněte a implementujte funkcionalitu pro tento GPU cloud, podporující dynamické plánování běhu virtuálních strojů (nebo kontejnerů) s přidělenými GPU prostředky tak, aby bylo plánování efektivní z hlediska využití GPU prostředků.
5. Srovnejte výkonnost GPU v rámci Openstacku versus GPU použité přímo na serveru.
6. Připravte softwarové řešení pro kontejnery nebo VM s GPU, které umožní výzkumníkům zadání úloh AI nebo strojového učení.
7. Proveďte validaci řešení GPU cloudu v ukázkové aplikaci výzkumu v oblasti AI či strojového učení.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

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

V Praze dne 6. května 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

GPU cloud

Bc. Michal Číla

Katedra teoretické informatiky
Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

8. ledna 2019

Poděkování

Rád bych poděkoval panu docentu Šimečkovi za vedení této práce, dále Zdeňku Jandovi za cenné rady při vývoji této práce a v neposlední radě bych rád poděkoval své snoubence a rodině za jejich podporu.

Prohlášení

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

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

V Praze dne 8. ledna 2019 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Michal Číla. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Číla, Michal. *GPU cloud.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato práce se zabývá virtualizací grafických karet na cloudové platformě Openstack. Práce se také zaměřuje na implementaci komponenty Openstacku, která virtuálním strojům dynamicky přiděluje grafické karty po omezeně dlouhou dobu.

Klíčová slova Openstack, cloud, grafická karta, virtualizace.

Abstract

This thesis deals with a virtualization of graphics cards in cloud platform Openstack. The aim of this thesis is also to implement a component of Openstack which dynamically allocates GPU cards to virtual machines for a limited time.

Keywords Openstack, cloud, graphics card, virtualization.

Obsah

Úvod **1**

1 Představení architektury Openstack **3**
1.1 Datacentrum budoucnosti . . . 3
1.2 Keystone . . . 4
1.3 Cinder . . . 4
1.4 Glance . . . 5
1.5 Nova . . . 5
1.6 Neutron . . . 6
1.7 Heat . . . 7
1.8 Horizon . . . 7
1.9 Ceilometer . . . 7
1.10 Fronta zpráv a databáze . . . 8
1.11 Openstack klient . . . 8
1.12 Shrnutí - zprovoznění virtuálního stroje . . . 9

2 Kubernetes orchestrace kontejnerů **11**
2.1 Docker a kontejnery . . . 11
2.2 Kubernetes . . . 12
2.3 Kubernetes v rámci Openstacku . . . 13
2.4 Srovnání výkonu kontejneru vs. výkonu VM . . . 14

3 Analýza a návrh **19**
3.1 Možnosti virtualizace grafických karet . . . 19
3.2 Návrh cloudu Openstack . . . 20
3.3 Návrh funkcionality . . . 24
3.4 Návrh architektury . . . 25

4 Realizace **31**
4.1 Instalace cloudu Openstack . . . 31

4.2 Zprovoznění virtualizace GPU na compute serveru 32
4.3 Flare 37
4.4 Vytváření vlastních obrazů 44

5 Testování a diskuse 47
5.1 Srovnání výkonnosti GPU 47
5.2 Ukázkové trénování neuronové sítě 50
5.3 Existující řešení 51
5.4 Možné rozšíření práce 52

Závěr 55

Literatura 57

A Seznam použitých zkratek 61

B Obsah přiloženého CD 63

Seznam obrázků

1.1 Znázornění komunikace jednotlivých komponent Openstacku [1] . . 10

2.1 Porovnání architektur kontejneru a virtuálního stroje 12
2.2 Architektura Dockeru 13
2.3 Architektura Openstack Magnum 14

3.1 Tři možnosti virtualizace grafických karet 19
3.2 Návrh cloudu Openstack 21
3.3 Znázornění nosičů grafických karet 23
3.4 Diagram životního cyklu jedné úlohy 25
3.5 Návrh architektury projektu 26

4.1 Znázornění autentikace požadavků 39

5.1 Náhodný výběr obrázků z 10 tříd datasetu `CIFAR-10` 51
5.2 Schéma komunikace VM a grafického akcelerátoru v cloudu Amazon 52

Seznam tabulek

2.1 Unixbench měření - test na jednom CPU jádře 16
2.2 Unixbench měření - test na čtyřech CPU jádrech 16

5.1 UnifiedMemoryPerf test 49
5.2 `MatrixMul_nvrtc` test 49
5.3 Bandwidth test - GPU na serveru 49
5.4 Bandwidth test - GPU ve virtuálním stroji 50

Úvod

V dnešní době se stále více a více rozrůstá obliba cloudových řešení. Možná větší oblibě se však těší umělá inteligence či strojové učení. Tato dvě odvětví se však neobejdou bez grafických karet, samotné virtuální stroje v cloudu nestačí.

Tato práce se zabývá možností virtualizací grafických karet v rámci open-source cloudové platformy Openstack. Jelikož nejsou grafické karty ani v dnešní době levná záležitost, je součástí této práce i návrh a implementace komponenty do Openstacku, nazvané Openstack Flare, která má za cíl dynamicky plánovat běh virtuálních strojů s přidělenými GPU prostředky tak, aby byly grafické karty využity co nejlépe.

V první kapitole se seznámíme s platformou Openstack a jejími komponentami. Zároveň se podíváme na to, co vše obnáší vytvoření virtuálního stroje na platformě Openstack.

V druhé kapitole se krátce seznámíme s kontejnery a jejich orchestrací. Dále se podíváme na možnost orchestrace kontejnerů v rámci Openstacku pomocí projektu Openstack Magnum.

Třetí kapitola se zabývá návrhem GPU cloudu, možnostmi virtualizace grafických karet a návrhem funkcionality a architektury komponenty Openstack Flare.

Ve čtvrté kapitole se blíže seznámíme s realizací tohoto projektu. Podíváme se například, co obnášela instalace cloudu Openstack a jeho úprava pro podporu virtualizace grafických karet. Kromě realizace komponenty Openstack Flare je v této kapitole popsána i možnost vytváření vlastních obrazů do Openstacku, což můžou ocenit například výzkumníci v oblasti umělé inteligence či strojového učení.

V poslední kapitole si v několika testech srovnáme výkon virtualizované grafické karty oproti kartě nevirtualizované a provedeme ukázkové trénování neuronové sítě v tomto GPU cloudu.

Kapitola **1**

Představení architektury Openstack

V této kapitole je čerpáno z [1].

1.1 Datacentrum budoucnosti

V dnešní době stále existuje spousta firem či organizací, které používají drahé IT systémy ve svých datacentrech. Tyto systémy se postupem času stávají neudržitelné. Jedná se často o proprietární systémy, které jsou nákladné na provoz a údržbu. Openstack je open-source software pro tvorbu privátních a veřejných cloudů. Je sestaven z mnoha nezávislých komponent, které budou detailněji popsány níže. Openstack je příslibem datacenter budoucnosti - datacenter, ve kterých budou operátoři a administrátoři skutečným pánem infrastruktury, to vše díky robustnímu, škálovatelnému a automatizovanému řešení.

Většina komponent Openstacku je napsána v jazyce Python a poskytuje *REST API*, které slouží jako externí komunikační rozhraní pro ostatní komponenty či pro koncového uživatele. Komponenty spolu komunikují pomocí fronty zpráv, kterou může obstarávat například software *RabbitMQ*. Zprávy se díky tomu například neztratí po síti.

Openstack tedy jako cloud obstarává správu virtuálních strojů, které budeme dále označovat jako *VM*, z anglického názvu *virtual machine*. Tyto virtuální stroje uchovávají svá data na takzvaných svazcích. Každé *VM* spadá do projektu (projekty často bývají uskupovány do domén), má přiřazenou jednu či více sítí. O každou část životního cyklu *VM* se stará jiná komponenta Openstacku, tyto komponenty si nyní detailněji popíšeme.

1.2 Keystone

Keystone je jedna ze základních komponent Openstacku a má v podstatě dvě role - autentikaci a autorizaci. Veškerá komunikace mezi jednotlivými komponentami Openstacku jde skrze Keystone, který ověří, zda má uživatel dostatečná oprávnění provést požadovanou akci. Keystone operuje s různými autentikačními mechanismy, jako jsou jméno/heslo, či použití tokenů k systémům založených na autentikaci.

Jedna z možností správy identit v Keystone je vlastní SQL databáze, avšak Keystone umožňuje integraci i s často užívanými autentikačními systémy, například *LDAP* či *PAM*. *LDAP* je aplikační protokol pro dotazování a modifikaci adresářových služeb nad TCP/IP [2]. *PAM* je autentikační modul, který citelně usnadňuje přidělování přístupových práv uživatelům [3]. Díky těmto možnostem se pak Keystone stará pouze o autorizaci požadavků, autentikaci přenechává třetí straně.

Keystone umožňuje použití více autentikačních systémů najednou. K tomu využívá domény, které jsou stejné jako domény shlukující projekty. Například může mít Keystone nastavenou doménu `default`, která využívá vlastní SQL databázi pro správu identit. V této doméně jsou obsaženi systémoví uživatelé Openstacku (neboť každá komponenta má vlastního uživatele) a také projekt `admin`, což je základní a první projekt vytvořený v Openstack cloudu. Dále může mít doménu `projects`, ve které už budou obsaženy konkrétní projekty pro konkrétní uživatele. Tato doména může mít nastavený *LDAP* pro správu identit. V případě pádu služby *LDAP* je tedy stále zajištěna funkčnost Openstack cloudu.

1.3 Cinder

Cinder se stará o persistentní *block storage*. Jeho cílem je poskytnout *VM* úložiště pomocí svazků, které můžeme chápat jako pevné disky pro *VM*. Mezi hlavní úkoly Cinderu patří:

- správa svazků: Cinder se stará o tvorbu, mazání svazků,
- správa snapshotů: snapshot k danému svazku můžeme chápat jako uchování stavu daného svazku v určitém okamžiku; snapshoty jsou určeny především k zálohování svazků,
- připojení či odpojení svazků od VM,
- klonování svazků,
- vytváření svazků ze snapshotů,
- tvorba obrazů ze svazků a naopak (více o obrazech v 1.4)

Cinder podporuje mnoho backendů pro uchovávání svazků, snapshotů a obrazů. Mezi nejčastěji používané patří *Ceph*, *NFS* či *GlusterFS*.

1.4 Glance

Glance slouží jako registr pro obrazy. Obrazem rozumíme vzor pro svazek, který již obsahuje data, například operační systém, ze kterého je pak možné spustit *VM*. Glance podporuje obrazy pro různé virtualizační platformy, např. *KVM/Qemu*, *XEN*, *VMware*, *Docker* a další. Obrazy bývají uloženy ve stejném úložišti společně se svazky a snapshoty. Úkolem Glance tedy není uchovávat obrazy, nýbrž být pouze registrem obrazů a metadat, která má dané *VM* obsahovat.

1.5 Nova

Nova je původní komponentou Openstacku. Ve starších verzích obsluhovala i úkoly, které převzal Cinder a Neutron (o Neutronu více v 1.6). Jedná se o jednu z nejsložitějších součástí Openstacku. Úkolem Novy je správa nejen virtuálních strojů, ale i nosičů, na kterých virtuální stroje přebývají. Složitost Novy se odvíjí od toho, že Nova musí komunikovat s obrovským množstvím ostatních komponent, aby dosáhla požadovaného výsledku od uživatele - funkčního *VM*, na které se uživatel může přihlásit. Nova se sestává především z následujících částí:

- **nova-api**: přijímá a odpovídá na volání uživatele, ale i nosičů, které obsluhuje; přes toto API uživatel například vytváří či maže instance,
- **nova-compute**: démon, který je umístěn na nosičích určených pro běh virtuálních strojů; jeho úkolem je vytváření a mazání virtuálních strojů na daném nosiči,
- **nova-scheduler**: plánuje, na který nosič má být virtuální stroj vytvořen - zabraňuje například situaci vytvoření virtuálního stroje na nosiči, který je již přetížen, ale jeho úkolem je i rozmístění instancí mezi správné nosiče kvůli zachování vysoké dostupnosti (High Availability),
- **nova-conductor**: zabezpečující prvek, který zabraňuje nosičům v přímém přístupu k databázi; tím je zvýšena bezpečnost databáze v případě, že je kompromitován jeden či více nosičů.

Parametry vytvářeného virtuálního stroje lze definovat pomocí tzv. *flavor*. Flavor v sobě obsahuje informace například o počtu virtuálních CPU jader či RAM paměti, kterou má daný virtuální stroj obdržet. Obsahuje však mnoho dalších informací, jak můžeme vidět na Výstupu příkazové řádky 1.1,

například jsme schopni nastavit velikost tzv. *ephemeral* disku - to je disk, který slouží jako úložiště pro virtuální server, nemůžeme jej však pokládat za persistentní úložiště, neboť toto úložiště je předpokládáno jako dočasné, není totiž definováno, co se s takovýmto diskem stane po vypnutí virtuálního serveru či po vypnutí nosiče tohoto serveru. Takový disk využívá například nástroj Packer, popsaný v sekci 4.4. Dále je možné například nastavit poměr vstupního a výstupního objemu dat přeneseného do/z virtuálního stroje, či velikost odkládacího prostoru *swap*.

```
$ openstack flavor show 1CPU_1GB
+----------------------------+--------------------------------------+
| Field                      | Value                                |
+----------------------------+--------------------------------------+
| OS-FLV-DISABLED:disabled   | False                                |
| OS-FLV-EXT-DATA:ephemeral  | 0                                    |
| access_project_ids         | None                                 |
| disk                       | 0                                    |
| id                         | ccccf775-e739-4ad4-bfbd-4e10a1636e95 |
| name                       | 1CPU_1GB                             |
| os-flavor-access:is_public | True                                 |
| properties                 |                                      |
| ram                        | 1024                                 |
| rxtx_factor                | 1.0                                  |
| swap                       |                                      |
| vcpus                      | 1                                    |
+----------------------------+--------------------------------------+
```

Výstup příkazové řádky 1.1: Definice *flavor* předávající jedno virtuální CPU jádro a 1 GB RAM

1.6 Neutron

Neutron je komponenta, která se stará o sítě. Vytváří v Openstacku jakousi virtuální síťovou infrastrukturu, a to včetně směrovačů, přepínačů a propojů mezi stroji a přepínači. Sestává se ze základních prostředků:

- **porty**: porty v Neutronu odpovídají propojům mezi virtuálními stroji a virtuálními přepínači; každý port má přiřazenou MAC adresu a IP adresu,
- **sítě**: sítě jsou v Neutronu definovány jako izolované segmenty na vrstvě L2 v modelu ISO-OSI; operátor si může představit síť v Openstacku jako logický přepínač, který je implementován pomocí virtualizačních síťovacích nástrojů, jako je například Open vSwitch; narozdíl od fyzických sítí můžou vytvářet tyto sítě jak operátoři, tak obyčejní uživatelé,

- **podsítě**: podsíť je v Neutronu reprezentovaná jako blok IP adres přiřazený k dané síti.

Architekturu Neutronu můžeme shrnout do tří hlavních komponent:

- **neutron server**: tento démon příjímá API požadavky a následně je předává příslušnému neutron pluginu, aby provedl požadovanou akci,
- **neutron plugin**: plugin se stará o skutečnou práci, například připojení či odpojení portu, vytváření sítí či podsítí, nebo adresace sítě,
- **neutron agent**: agent je umístěn na všech nosičích a síťových uzlech; stará se o příjímání příkazů od pluginů a provádí požadované změny.

Neutron obsahuje vlastní *firewall* pro virtuální servery. Využívá k tomu tzv. *security groups*. Každá *security group* může obsahovat pravidla příchozí a odchozí. Je například možné k virtuálnímu serveru povolit přístup pouze přes určitý port na daném protokolu (TCP či UDP) či přístupy zakazovat [4].

1.7 Heat

Heat je orchestrační nástroj Openstacku. Orchestrací rozumíme automatickou koordinaci a řízení komplexních počítačových systémů. Díky této komponentě si uživatelé mohou vytvářet tzv. *stacks*, které si můžeme představit jako sadu jednoho či více virtuálních strojů připravených provádět požadovanou akci. Toho Heat dosahuje pomocí šablonových souborů, které můžou být ve formátu *JSON* nebo *YAML*. V těchto souborech můžeme nadefinovat například *VM*, ale i svazky, sítě, porty a mnohé jiné prostředky, které ve výsledku tvoří ucelený *stack*. Díky této komponentě, která je základním kamenem tohoto projektu, může uživatel například vytvořit aplikaci sestávající z několika *VM* jediným stiskem tlačítka.

1.8 Horizon

Horizon je webové rozhraní celého Openstacku. Jde o webový frontend pro všechny komponenty Openstacku, spojuje je tak dohromady. Jeho úkolem je pouze se dotazovat na komponenty Openstacku skrze jejich API a zobrazovat uživateli požadované informace, které mu daná komponenta vrací. Horizon je navržený tak, aby byl snadno rozšiřitelný pro operátory pro daný cloud.

1.9 Ceilometer

Ceilometer je komponenta sbírající statistiky o využívání služeb Openstacku. Sbírá informace i o virtuálních strojích běžících v rámci Openstacku. Je hojně

využíván ve veřejných instancích Openstacku, kdy jsou přeprodávány virtuální servery koncovým uživatelům. Díky technologii Ceilometer je totiž možné účtovat koncovému uživateli konkrétní využité prostředky v rámci Openstack cloudu, například využitý procesorový čas, využité úložiště a další.

1.10 Fronta zpráv a databáze

Pomocí fronty zpráv se jednotlivé komponenty mezi sebou doruzumívají. Jde o místo, kde se sdílejí veškeré informace mezi různými démony. Obrovskou výhodou tohoto řešení je možnost kupit více požadavků najednou.

Databáze Openstacku ukládá většinu stavových informací celé infrastruktury, například stav virtuálního stroje, volné IP adresy v podsíti, či kvóty v daném projektu. Jde o persistentní úložiště uchovávající stav infrastruktury celého cloudu.

1.11 Openstack klient

Pro práci s Openstack cloudem lze využívat kromě uživatelského prostředí Horizon i CLI klienta. Buď můžeme použít jednotlivé klienty pro práci s danou komponentou nebo uceleného klienta `python-openstackclient`, který komunikuje se všemi komponentami. Tedy například klient `python-novaclient` obsahuje příkaz `nova list --all-tenants`, který vypíše na příkazovou řádku seznam virtuálních serverů ze všech projektů. Analogický příkaz obecného klienta je `openstack server list --all-projects`.

Aby se však všichni tito klienti mohli spojit se správným cloudem, musí mít zadané informace o tomto cloudu. To znamená předávat klientovi především tyto proměnné:

- `OS_AUTH_URL`: specifikuje cestu k Openstack cloudu, resp. k jeho Keystone serveru; příklad: `http://gpucloud.com:35357/v3`,
- `OS_PROJECT_DOMAIN_NAME`: název domény, ve které je obsažen projekt, ke kterému chceme přistupovat,
- `OS_USER_DOMAIN_NAME`: název Keystone domény - Keystone poté v dané doméně hledá přihlašujícího se uživatele,
- `OS_PROJECT_NAME`: jméno projektu, ke kterému chceme přistupovat,
- `OS_USERNAME`: uživatelské jméno, kterým přistupujeme ke cloudu,
- `OS_PASSWORD`: heslo, kterým se autentikujeme v Keystone,
- `OS_IDENTITY_API_VERSION`: verze Keystone API (dnes se používá verze 3),

- `OS_IMAGE_API_VERSION`: verze Glance API (dnes se používá verze 2).

Předat tyto parametry můžeme dvěma způsoby:

1. použitím argumentů u zadaných příkazů: Tato možnost je nejméně přívětivá, neboť při každém použití příkazu Openstack klienta je třeba zadávat všechny tyto informace, například výše uvedeného vypsání všech virtuálních serverů v Openstacku bychom dosáhli příkazem:

```
openstack --os-auth-url http://gpucloud.com:35357/v3
--os-project-domain-name default --os-user-domain-name projects
--os-project-name project --os-username username --os-password password
--os-identity-api-version 3 --os-image-api-version 2
server list --all-projects
```

2. použitím tzv. RC souboru definujícího dané prostředí: Tato možnost je vyžívána nejčastěji. Do souboru si nadefinujeme dané proměnné například takto:

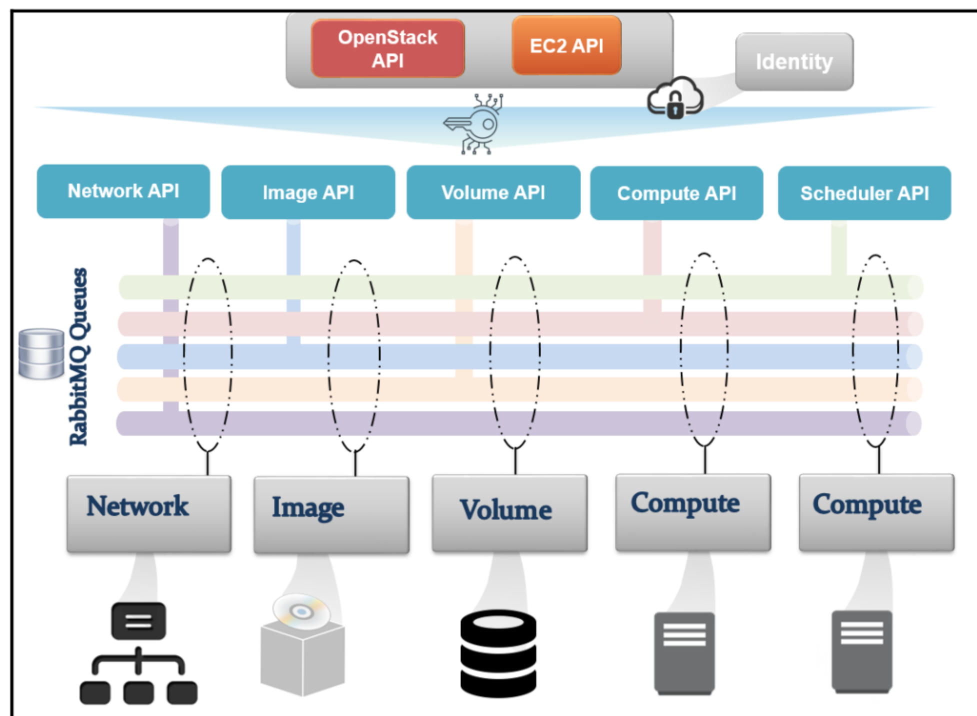
```
export OS_AUTH_URL=http://gpucloud.com:35357/v3
export OS_PROJECT_DOMAIN_NAME=default
export OS_USER_DOMAIN_NAME=projects
export OS_PROJECT_NAME=project
export OS_USERNAME=username
export OS_PASSWORD=password
export OS_IDENTITY_API_VERSION=3
export OS_IMAGE_API_VERSION=2
```

   Tyto proměnné uložené například v souboru `openstackrc` v příkazové řádce aktivujeme příkazem `source openstackrc`. Poté již můžeme s klientem pracovat a budeme připojeni na správný Openstack cloud [5].

1.12 Shrnutí - zprovoznění virtuálního stroje

Proces zprovoznění virtuálního stroje zahrnuje interakci mezi hlavními komponentami Openstacku. Jednotlivé služby spolu komunikují pomocí fronty zpráv, kterou může obstarávat například software *RabbitMQ*. Každá informace je verifikována a přenášena jiné službě pomocí sběrnice zpráv. Nicméně každý požadavek musí být autentikován - o to se stará Keystone. Počínaje autentikační službou Keystone se tedy pomocí následujících kroků dopracujeme k funkčnímu virtuálnímu stroji (komunikace těchto komponent je znázorněna na Obrázku 1.1 - zdroj [1]):

- pomocí registru obrazů Glance získáme požadovaný obraz, který služba Cinder použije k vytvoření svazku pro virtuální server,
- Cinder vytvoří svazek (což je v danou chvíli de facto kopie obrazu),
- Nova pomocí služby `nova-scheduler` naplánuje, který nosič je nejvhodnější pro požadovaný virtuální stroj, a následně informuje službu `nova-compute` běžící na vybraném nosiči, že se stává nosičem daného virtuálního stroje,



Obrázek 1.1: Znázornění komunikace jednotlivých komponent Openstacku [1]

- Neutron následně připojí požadovanou síť k tomuto virtuálnímu stroji, přiřadí mu IP adresu a nastaví požadované *security groups*,
- uživatel se pak pomocí zadaného SSH klíče může na server přihlásit.

Kapitola 2

Kubernetes orchestrace kontejnerů

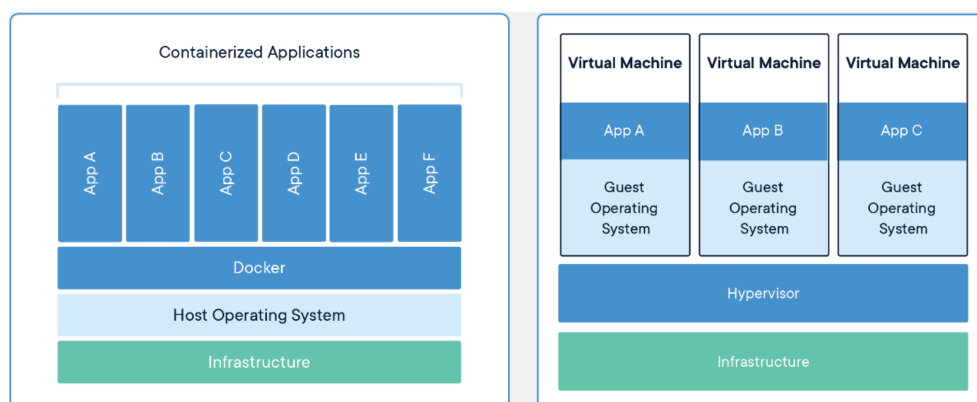
V této kapitole se seznámíme s kontejnery a jejich orchestrací pomocí Kubernetes. Následně se budeme zabývat možností orchestrace kontejnerů v rámci Openstacku a nakonec budeme porovnávat výkon kontejnerů s výkonem virtuálních strojů.

2.1 Docker a kontejnery

Kontejner je standardizovaná jednotka software, která spojuje zdrojový kód nějaké aplikace a jejích závislostí do jednoho balíčku, který obsahuje vše potřebné pro běh dané aplikace: zdrojový kód, runtime prostředí, systémové nástroje, knihovny a nastavení. Kontejnery obsluhuje software Docker. Kontejnerů může v rámci jednoho hostitelského stroje běžet více, všechny však sdílí jádro hostitelského operačního systému. Kontejnery obvykle zabírají méně místa na disku a mohou zvládnout více aplikací najednou.

Naproti tomu virtuální stroje jsou abstrakcí fyzického hardware, z čehož se z jednoho hostitelského serveru stává více serverů virtuálních. Každý virtuální stroj obsahuje vlastní kopii operačního systému, což zabírá více místa na disku. Virtuální stroje často nabíhají pomaleji než kontejnery. Jsou však schopným emulátorem celého operačního systému, nabízí tedy více možností než kontejnery, které jsou určeny pouze pro jednu či více aplikací. Na Obrázku 2.1 (zdroj: [6]) můžeme vidět porovnání architektur kontejneru a virtuálního stroje. Hlavní rozdíl mezi kontejnery a virtuálními stroji je tedy v tom, co přesně virtualizují. Kontejner má za úkol virtualizovat operační systém, zatímco virtuální stroj má za úkol virtualizovat hardware.

Docker je multiplatformní nástroj představený v roce 2013. Byl původně vydán pro linuxové systémy, ale postupem času se jeho podpora rozrostla i pro Windows a dnes je již možné Docker kontejnery provozovat například i



Obrázek 2.1: Porovnání architektur kontejneru a virtuálního stroje

v cloudu [6].

2.1.1 Docker architektura

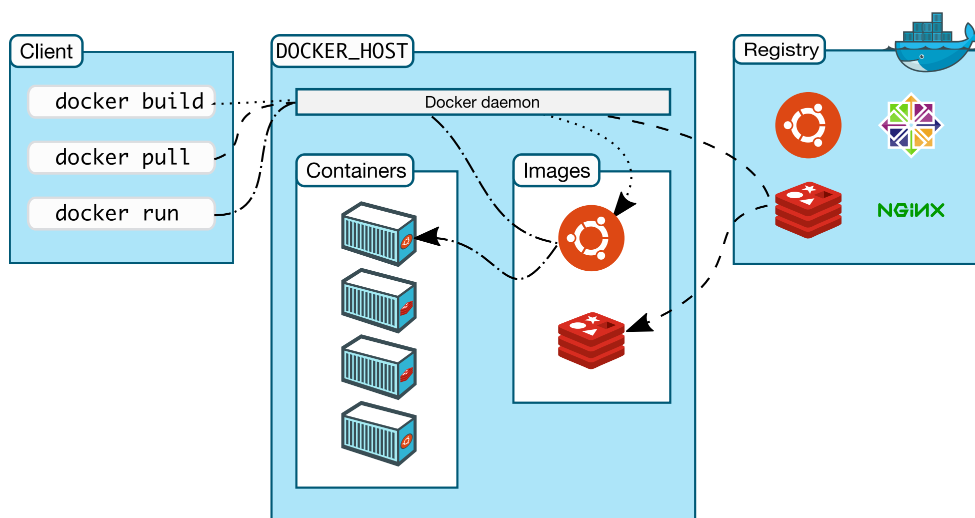
Architektura Dockeru je znázorněna na Obrázku 2.2 (zdroj: [7]). Docker používá architekturu klient - server. Docker klient komunikuje s Docker démonem, který se stará o tvorbu, distribuci a mazání kontejnerů. Klient může být umístěn na stejném zařízení společně s Docker démonem nebo se může připojovat k Docker démonu vzdáleně pomocí REST API.

Mezi další činnosti Docker démona patří práce s obrazy (*images*). Obraz může obsahovat buď odlehčenou verzi operačního systému, či přímo nějaký software (aplikaci). Z tohoto obrazu poté démon vytváří dané kontejnery.

Obrazy mohou být uloženy lokálně nebo ve vzdáleném Docker Registry. Docker Registry je registr obrazů, zastává podobnou funkcionalitu jako Openstack Glance popsaný v sekci 1.4.

2.2 Kubernetes

Kubernetes je nadstavba pro Docker kontejnery. Jedná se o open-source platformu pro správu kontejnerů. Umožňuje automatické nasazení kontejnerů i jejich škálování. V případě vydání nové verze kontejnerizované aplikace umí snadno nasadit tuto verzi a kontejnery běžící na staré verzi automaticky zlikvidovat. Jedná se o moderní přístup k vývoji nových aplikací, umožňující jejich rychlé testování a nasazení do produkčního provozu. Díky tzv. *self-healing* schopnosti Kubernetes rychle reaguje na pád kontejneru (například zapříčiněný pádem nosiče tohoto kontejneru) - kontejner jednoduše spustí znovu na zdravém nosiči [8].



Obrázek 2.2: Architektura Dockeru

2.2.1 Kubernetes architektura

Kubernetes je tvořený *clusterem.* Cluster se sestává z jedné či více *master* jednotek a jedné či více *node* jednotek. Tyto jednotky mohou být jak fyzické servery, tak servery virtuální.

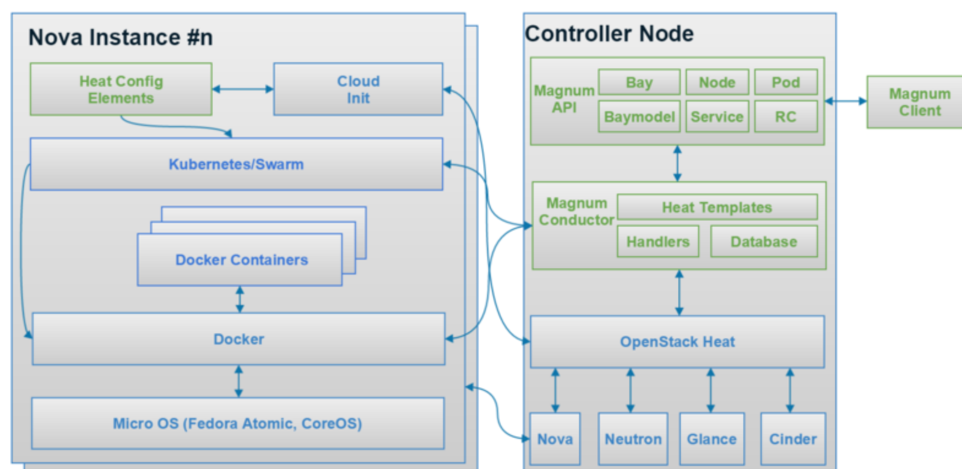
Master jednotky mají za úkol sestavit kýžený Kubernetes cluster a také se o tento cluster starat. Dále reagují na uživatelské příkazy pomocí API - starají se tedy o tvorbu či mazání kontejnerizovaných aplikací [9].

Node jednotky slouží jako nosiče kontejnerizovaných aplikací. V Kubernetes terminologii se jednomu či více kontejnerům, které musí běžet společně, aby korektně prováděly požadovaný úkon, říká `pod`. Jde o nejmenší možný objekt, se kterým lze v rámci Kubernetes clusteru pracovat. Pody jsou definovatelné pomocí šablonových souborů, které mimo jiné definují, z kterého obrazu se mají dané pody vytvořit [10].

2.3 Kubernetes v rámci Openstacku

Pro tvorbu Kubernetes clusterů v rámci Openstacku slouží projekt Magnum. Magnum umožňuje kontejnerovým orchestračním nástrojům, jako je například Kubernetes, být dostupnými v rámci Openstacku pro koncové uživatele jako zdroj.

Magnum využívá komponenty Heat k orchestraci Kubernetes clusterů pomocí Heat šablonových souborů. Výsledkem jsou tedy virtuální stroje běžící na platformě Openstack, které mají nainstalovaný a funkční Kubernetes cluster [11].



Obrázek 2.3: Architektura Openstack Magnum

2.3.1 Magnum architektura

Architektura komponenty Magnum je znázorněna na Obrázku 2.3 (zdroj: [11]). Magnum se sestává ze dvou služeb, a to `magnum-api`, což je API sloužící pro komunikaci s uživatelem, a `magnum-conductor`, což je služba obstarávající tvorbu, úpravu a mazání Kubernetes clusterů v rámci Openstacku. Magnum Conductor tedy komunikuje s Heat komponentou a předává jí Heat šablony, načež Heat vytvoří virtuální stroje s nainstalovaným orchestračním nástrojem kontejnerů.

Pomocí Magnum API lze snadno škálovat již vytvořený Kubernetes cluster, uživatel jednoduše navýší počet `master` či `node` jednotek a Magnum Conductor poté instruuje Heat k vytvoření další takové jednotky.

2.4 Srovnání výkonu kontejneru vs. výkonu VM

V této sekci porovnáme výkon kontejnerů běžících na Openstacku s výkonem virtuálních strojů také běžících na Openstacku. Testovací virtuální stroj má nainstalovaný systém Ubuntu 16.04 Server a má k dispozici 4 virtuální CPU jádra a 8 GB RAM paměti. Nosič tohoto virtuálního stroje obsahuje dva procesory Intel® Xeon® E5645. Jelikož je testovací Kubernetes cluster také vytvořen na Openstacku, jedná se i zde o virtuální stroje, mají stejné parametry jako náš testovací virtuální stroj. V Kubernetes clusteru byl pro testování vytvořen `pod`, ve kterém běží odlehčená verze systému Ubuntu 16.04 Server. Tento `pod` běží na jedné z *node* jednotek Kubernetes clusteru. Žádný další `pod` zde neběží, celý virtuální stroj by tedy měl být k dispozici našemu testovacímu `podu`.

Ke srovnání výkonu byl využit nástroj Unixbench, což je sada testů měřících výkon hardware a operačního systému. Obsahuje následující testy:

- *Dhrystone*: Tento test slouží k měření a porovnání výkonnosti počítačů. Zaměřuje se na práci s řetězci, tudíž zde nedochází k operacím s plovoucí řádovou čárkou. Je vysoce ovlivněn hardwarem ale i návrhem software, kompilátorem, optimalizací kódu aj.

- *Whetstone*: Tento test měří rychlost a efektivitu operací s plovoucí řádovou čárkou. Test obsahuje řadu modulů, které mají za cíl reprezentovat mix operací typicky prováděných ve vědeckých aplikacích. Test využívá řadu funkcní v jazyce C, mezi ně patří například `sin`, `cos`, `sqrt`, `exp` či `log`, ale i operace s celými a desetinnými čísly, přístupy k poli, podmíněných větví a volání funkcí.

- `execl` *Throughput*: Tento test měří počet vykonaných funkcí `execl` za vteřinu. Funkce `execl` je z rodiny `exec` funkcí v jazyce C, která nahrazuje nějaký proces procesem jiným, přičemž zachovává ID původního procesu.

- *File Copy*: Tento test měří tempo, jakým je systém schopen zkopírovat obsah jednoho souboru do souboru jiného, přičemž využívá různé velikosti *bufferu*.

- *Pipe Throughput*: Roura (*pipe*) je nejjednodušší forma komunikace mezi procesy. Tento test měří, kolikrát za sekundu dokáže proces zapsat 512 bajtů do roury a zase zpět.

- *Pipe-based Context Switching*: Tento test měří, kolikrát si jsou dva procesy schopné vyměnit zvyšující se celé číslo skrz rouru.

- *Process Creation*: Tento test měří, kolikrát se dokáže proces rozštěpit na sebe a dceřinný proces a tento proces následně zabít. Vytvoření procesu obnáší i alokaci paměti pro tento nový proces, tudíž tento test měří i propustnost paměti.

- *Shell Scripts*: Tento test měří, kolikrát za minutu je proces schopen spustit a zastavit množinu jednoho, dvou, čtyř a osmi souběžných shell skriptů, přičemž tyto skripty spouští sérii transformací do datového souboru.

- *System Call Overhead*: Tento test zjišťuje cenu vstupu a výstupu z jádra operačního systému, tedy zjišťuje režii spojenou se systémovým voláním. Test se skládá z jednoduchého programu volajícího funkci `getpid()`, která vrací ID volajícího procesu.

	Kontejner		Virtuální stroj	
Použitý test	**Výsledek**	**Index**	**Výsledek**	**Index**
Dhrystone	26846249,6	2300,4	27890155,5	2389,9
Whetstone	3388,1	616,0	3555,6	646,5
Execl Throughput	2897,5	673,8	3114,6	724,3
File Copy - velikost bufferu 1024, max 2000 bloků	421433,8	1064,2	476318,0	1202,8
File Copy - velikost bufferu 256, max 500 bloků	113812,3	687,7	124431,8	751,9
File Copy - velikost bufferu 4096, max 8000 bloků	1075609,7	1854,5	1128845,1	1946,3
Pipe Throughput	687596,6	552,7	721671,8	580,1
Pipe-based Context Switching	49511,5	123,8	92989,2	232,5
Process Creation	4445,5	352,8	6643,3	527,2
Shell scripts (1 souběžný skript)	5173,3	1220,1	7213,2	1701,2
Shell scripts (8 souběžných skriptů)	1324,8	2207,9	2171,5	3619,2
System Call Overhead	526289,4	350,9	559361,7	372,9

Tabulka 2.1: Unixbench měření - test na jednom CPU jádře

	Kontejner		Virtuální stroj	
Použitý test	**Výsledek**	**Index**	**Výsledek**	**Index**
Dhrystone	105041648,7	9001,0	102622097,8	8793,7
Whetstone	13232,8	2406,0	13208,8	2401,6
Execl Throughput	9556,9	2222,5	11168,5	2597,3
File Copy - velikost bufferu 1024, max 2000 bloků	549629,5	1388,0	807939,6	2040,3
File Copy - velikost bufferu 256, max 500 bloků	142431,6	860,6	220238,8	1330,7
File Copy - velikost bufferu 4096, max 8000 bloků	1641434,7	2830,1	2265287,0	3905,7
Pipe Throughput	2601914,4	2091,6	2762598,8	2220,7
Pipe-based Context Switching	215979,4	539,9	547634,9	1369,1
Process Creation	9989,2	792,8	18659,7	1480,9
Shell scripts (1 souběžný skript)	9945,3	2345,6	18459,1	4353,6
Shell scripts (8 souběžných skriptů)	1760,6	2934,3	2936,8	4894,7
System Call Overhead	1770118,5	1180,1	1945899,3	1297,3

Tabulka 2.2: Unixbench měření - test na čtyřech CPU jádrech

Tato sada testů byla spuštěna jak na virtuálním stroji, tak v kontejneru. Testy jsou prováděny nejdříve s jednou paralelní kopií testů. Měří tedy nejdříve výkon jednoho jádra CPU. Poté jsou provedeny testy paralelně tolikrát, kolik je v systému dostupných CPU jader. V našem případe čtyřikrát. Test vrací uživateli jak výsledek, tak index obdržený v daném testu. Index je většinou jednodušší na porovnání mezi dvěma hodnotami. Nakonec test vrátí celkový index, který daný stroj získává [12].

Tabulky 2.1, resp.2.2, zobrazují získané výsledky a indexy provedených testů na jednom, resp. na čtyřech CPU jádrech. Zeleně jsou zvýrazněny získané lepší indexy v daném testu. V drtivé většině případů získal lepší index virtuální stroj, což potvrzuje i celkový získaný index **909,8**, resp. **2540,8**, na jednom, resp. na čtyřech CPU jádrech. Kontejner obdržel výsledné indexy **745,7**, resp. **1803,4**, pro jedno, resp. pro čtyři CPU jádra.

Bylo celkem očekávatelné, že si kontejner v testu výkonnosti povede hůře oproti virtuálnímu stroji. Je to způsobeno tím, že kontejner je spuštěn na

virtuálním stroji, nikoliv na fyzickém hardware. V této kapitole šlo však o testování kontejnerů v rámci Openstacku. Pro další vývoj tedy byla vybrána možnost virtuálního stroje.

Kapitola 3

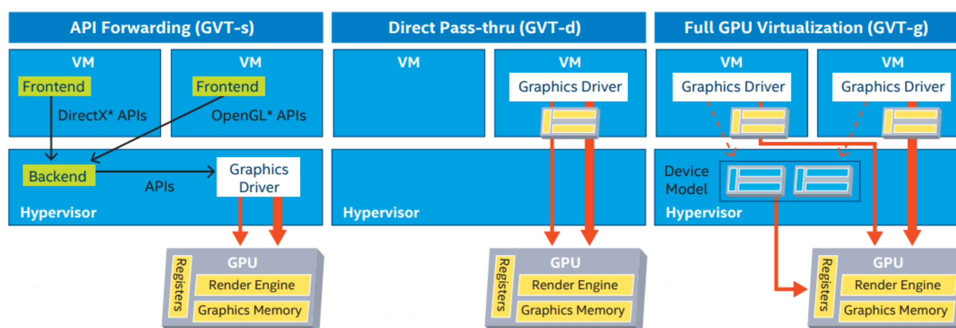
Analýza a návrh

3.1 Možnosti virtualizace grafických karet

Dle [13] existují tři možnosti, jak virtualizovat GPU v rámci Openstacku, všechny využívají technologie firmy Intel, pojmenované Intel® Graphics Virtualization Technology. Tyto možnosti jsou znázorněny na Obrázku 3.1 (zdroj: [13]). Jsou rozdělené podle míry virtualizace a míry sdílenosti GPU prostředků mezi virtuálními stroji.

3.1.1 Intel® GVT-s

První metoda virtualizace grafických karet využívá technologie tzv. *API forwarding*, tedy přeposílání API požadavků. Každý virtuální stroj požadující přístup ke grafické kartě musí mít nainstalován virtuální grafický ovladač (například *DirectX* nebo *OpenGL*) a pomocí tohoto ovladače přeposílá API požadavky grafické kartě. Skutečný hardware (grafická karta) se tedy k aplikacím (uloženým na virtuálních strojích) chová abstraktně [14].



Obrázek 3.1: Tři možnosti virtualizace grafických karet

Výhodou tohoto řešení je možnost sdílení jediné grafické karty mezi více virtuálními stroji, nicméně dle [13] je nevýhodou nedostatečná kompatibilita.

3.1.2 Intel® GVT-d

Druhá možnost virtualizace grafických karet umožňuje virtuálnímu stroji přímý přístup ke grafické kartě. Tato technologie předává virtuálnímu stroji nativní možnosti ovladače grafické karty, je tedy předpokladem mít nainstalovaný ovladač dané grafické karty na virtuálním stroji. S grafickou kartou poté uživatel operuje stejným způsobem, jako kdyby se nejednalo o virtualizaci. Této možnosti virtualizace je dosaženo pomocí základních hardwarových virtualizačních metod technologie Intel VT-d [14]. Výhodou tohoto řešení je výkon - uživatel nepozná rozdíl virtualizované grafické karty metotou Intel® GVT-d a karty nevirtualizované. Jeho nevýhodou je však nemožnost sdílení takové grafické karty mezi více virtuálními stroji [13].

3.1.3 Intel® GVT-g

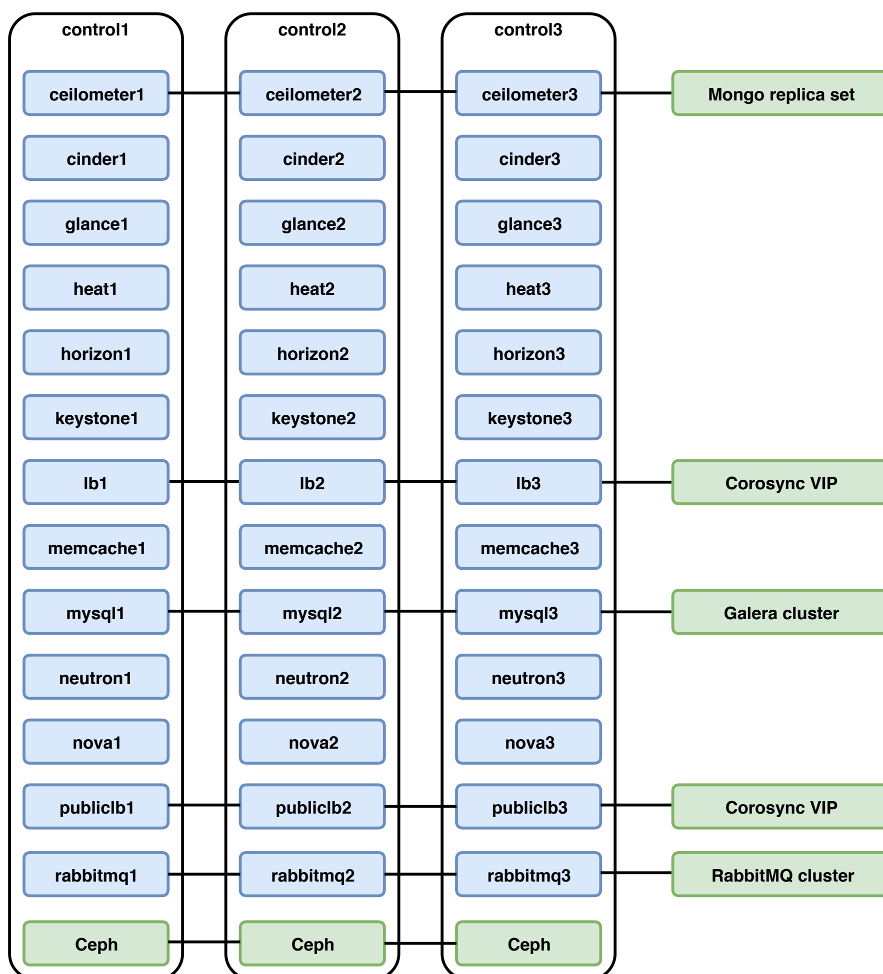
Třetí možnost virtualizace grafických karet je nejnovější metodou. Každý virtuální stroj si udržuje kopii nativních ovladačů grafické karty. Agent, který je uložen na nosiči grafické karty, poté v časových okéncích předává danému virtuálnímu stroji grafickou kartu k plnému využití. V daném okamžiku, kdy má jednu grafickou kartu k dispozici jediný virtuální stroj, se z celkového systémového hlediska může jevit jako sdílený přístup ke grafické kartě několika virtuálními stroji. Ač se toto řešení může jevit jako nejlepší vhodné, Intel tuto možnost uzavřel pouze vybraným partnerům [14].

3.2 Návrh cloudu Openstack

Pro vývoj tohoto projektu je třeba mít k dispozici funkční cloud Openstack. Jelikož tento cloud bude dále sloužit jako GPU cloud pro výzkumníky v oblasti AI a strojového učení, bude dbáno na vysokou dostupnost (High Availability) tohoto cloudu.

Jelikož Openstack je rozsáhlý ekosystém se spoustou komponent, manuální instalace tohoto systému je velice náročný úkon, je tedy vhodné využít automatizačního nástroje. K automatizované instalaci systému Openstack bude využit software Puppet, což je nástroj ke spravování a nastavování konfigurace mnohých infrastruktur, mezi něž patří i Openstack. Tento nástroj obsahuje i deklarativní stejnojmenný jazyk, který slouží k automatizovanému provádění požadované akce, nastavování konfigurace souborů a mnohé jiné [15].

V následujících podsekcích jsou detailněji popsány jednotlivé servery, jak fyzické, tak virtuální, potřebné pro bezproblémovou funkčnost cloudu Openstack.



Obrázek 3.2: Návrh cloudu Openstack

3.2.1 Nosiče obsluhující Openstack

K dispozici budou tři stroje (pojmenujme je `control1` až `control3`), které se nainstalují jako nosiče virtuálních strojů pomocí virtualizační technologie KVM. Na každém virtuálním stroji bude jedna komponenta Openstacku, tyto virtuální stroje jsou znázorněny modrými obdélníčky na Obrázku 3.2.

Základem Openstacku je databáze. Aby byla databáze skutečně vysoce dostupná, bude použita technologie MySQL Galera Cluster, která zaručuje replikaci mezi jednotlivými MySQL uzly. Toho Galera dosahuje pomocí tzv. `master - master` replikace, to znamená, že se všechny MySQL uzly chovají jako hlavní uzel, každý z nich si udržuje kopii všech dat v databázi a výpadek jakéhokoliv z těchto uzlů neohrozí integritu ani dostupnost dat [16].

Na nosičích `control1` až `control3` bude nainstalován software Ceph. Tento

software bude sloužit jako backend pro Cinder a Glance. Jedná se o distribuované moderní úložiště, často používané v datacentrech a hojně využívané jako backend právě pro Openstack. Jeho možnost tzv. *RBD* blokového úložiště umožňuje vysokou dostupnost jak obrazů, tak všech svazků připojených k virtuálním serverům. Ceph se při každém zápisu sám postará o replikaci dat napříč celým clusterem. Je zároveň snadno rozšiřitelný. Umožňuje navíc jednoduché vytváření snapshotů a import/export dat z/do jiného clusteru (například kvůli zálohování dat) [17].

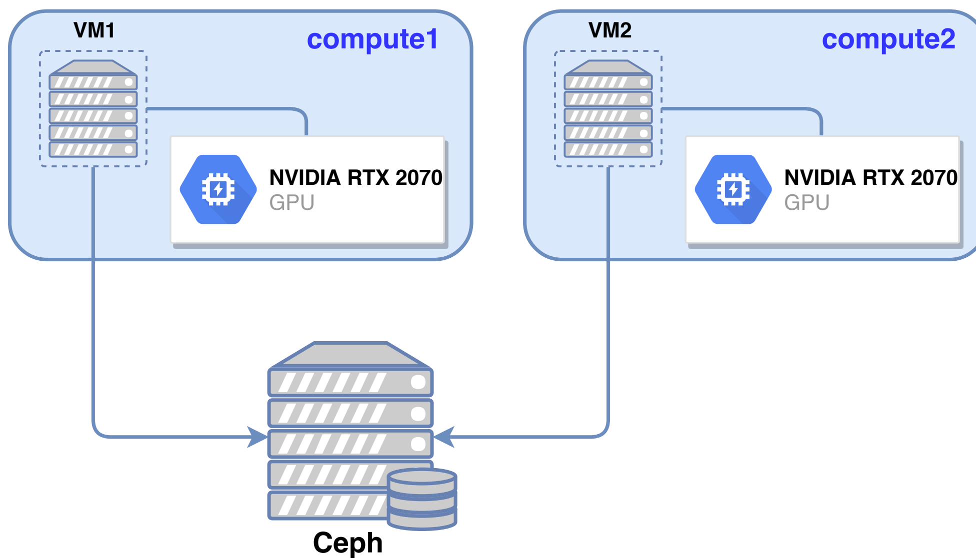
Virtuální stroje `lb1` až `lb3` budou sloužit k dodržení vysoké dostupnosti pro ostatní komponenty Openstacku, půjde o tzv. *load balancery*. Bude na nich nainstalován software HAProxy, což je proxy server, který ale zároveň dodržuje zásady *load balancingu* a vysoké dostupnosti. Umožňuje fungovat jak na HTTP, tak TCP protokolu [18]. Samotná HAProxy ale nestačí k zajištění fungování služeb, pokud jeden ze strojů `lb1` až `lb3` přestane fungovat. K HAProxy bude na těchto strojích nainstalován software Corosync, který díky virtuální IP adrese, tzv. `VIP`, umožní dostupnost balancerů i v případě, že jeden ze strojů přestane fungovat. Corosync si sám hlídá, zda jsou všechny tři servery dostupné a drží `VIP` adresu na jednom z nich. Pokud daný server, na kterém je `VIP` adresa, přestane fungovat, automaticky přesune tuto adresu na jiný stroj, který je v danou chvíli k dispozici. Stroje `lb1` až `lb3` nebudou sloužit jako balancery jen pro komponenty Openstacku, ale i pro databázi. Tím je zaručena i vysoká dostupnost Galera clusteru zmíněného výše.

Virtuální stroje `publiclb1` až `publiclb3` budou mít podobnou funkcionalitu jako stroje `lb1` až `lb3` zmíněné výše, avšak s tím rozdílem, že jde o veřejné balancery určené pro klienty. Tyto balancery budou zároveň dodávat klientovi uživatelské rozhraní Horizon, toto rozhraní tak bude pro klienta také vysoce dostupné.

K uživatelskému prostředí Horizon je třeba mít nakonfigurovanou cache, která bude udržovat danou relaci. K tomu je možné využít lokální cache, avšak toto není dopoučené, pokud chceme mít vysoce dostupný Openstack s více stroji, na kterých je nainstalován Horizon. Proto je lepší využít jeden či více Memcached serverů, které zajistí, že cache bude synchronizována mezi všemi třemi Horizon servery [19]. Memcached server bude nainstalován na virtuálních strojích `memcache1` až `memcache3`.

Nedílnou součástí kvalitního Openstack cloudu je mít robustní systém pro AMQP protokol. K tomu budou sloužit virtuální stroje `rabbitmq1` až `rabbitmq3`, na kterých bude nainstalován software RabbitMQ, což je zprostředkovatel zpráv, k čemuž mu slouží fronta zpráv. Pomocí této fronty komunikují všechny komponenty Openstacku mezi sebou, je tedy nesmírně důležité, aby tento software byl správně nakonfigurován a byl vysoce dostupný. RabbitMQ umí také provozovat cluster a díky tomu, že jsou všechny zprávy zrcadleny mezi všechny RabbitMQ servery, je zajištěna vysoká dostupnost a integrita těchto zpráv [20].

Ceilometer služba používá jako databázi pro uchovávání statistik Mon-



Obrázek 3.3: Znázornění nosičů grafických karet

goDB, což je multiplatformní dokumentová databáze. Tato databáze umožňuje také replikaci, tzv. *replica set*, tudíž je snadné díky této replikaci zajistit vysokou dostupnost těchto dat.

Všechny ostatní komponenty již nevyžadují žádnou další speciální úpravu pro zajištění vysoce dostupného cloudu. Je třeba mít v konfiguračních souborech nastavené virtuální IP adresy (například cesta k databázi, cesta ke Keystone službě, atd.), tudíž všechen provoz bude procházet skrze servery `lb1` až `lb3`.

3.2.2 Nosiče s GPU pro virtuální stroje Openstacku

Pro vývoj tohoto projektu budou k dispozici dva servery, které budou sloužit jako nosiče grafických karet a zároveň jako nosiče virtuálních strojů, na které budou tyto grafické karty připojeny. Grafické karty jsou NVIDIA GeForce RTX 2070. Každý takový server pojme právě jednu grafickou kartu a bude mít k dispozici dva procesory Intel® Xeon® E5-2660 a 16 GB RAM. Jak znázorňuje Obrázek 3.3, jako backend pro svazky bude sloužit Ceph, který bude nainstalovaný na serverech `control1` až `control3`, není tudíž nutné mít na serverech `compute1` a `compute2` nikterak velké úložiště, na těchto strojích bude nainstalovaný pouze systém Ubuntu 16.04 Server a na něm Openstack, konkrétně služba `nova-compute`.

3.3 Návrh funkcionality

Cílem tohoto projektu je umožnit zadávat výpočetní úlohy tak, aby byly efektivně využity výpočetní prostředky (grafické karty). Projekt byl pojmenován jako **Flare** od anglického slova flare, což znamená vzplanout či plápolat. Doba běhu takových úloh není předpokládána jako nekonečná, odtud analogie s plamínkem, který musí někdy zhasnout.

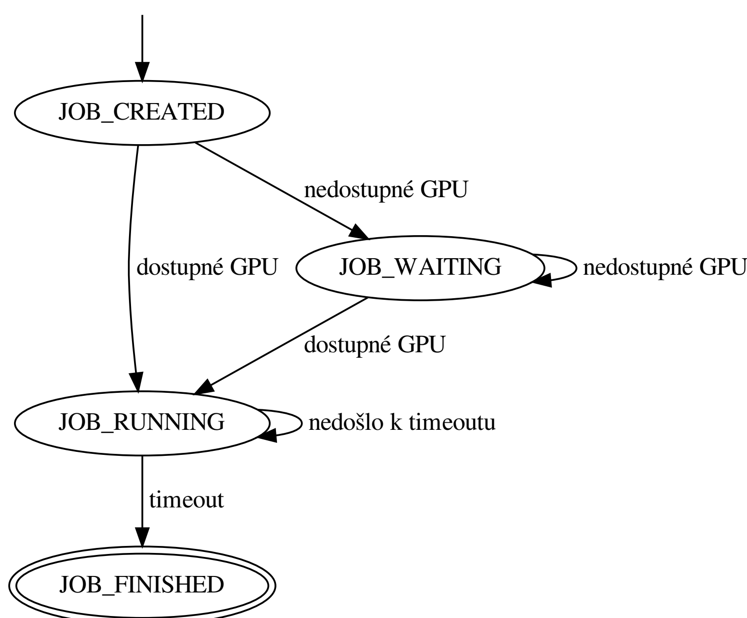
Úloha musí být vytvořena ze šablony, pojmenované jako *Job Template*. Šablona může sloužit například k opakovanému spuštění stejné úlohy. Tato šablona obsahuje následující parametry:

- `flavor`: flavor ID, které bude předáno Nově na vytvoření serveru; tento flavor obsahuje nejen informace o počtu virtuálních CPU jader a velikostí RAM paměti, ale zároveň i informace o použité grafické kartě,
- `image`: ID obrazu, ze kterého má být server vytvořen,
- `network`: ID sítě, která má být k serveru připojena,
- `public`: příznak, který indikuje, zda je tato šablona veřejná, tedy dostupná pro všechny projekty v rámci celého Openstacku, nebo zda je šablona dedikována pro daný projekt, ve kterém je vytvářena,
- `volume_size`: udává v gigabytech velikost svazku, který má být k serveru připojen a sloužit jako hlavní úložiště,
- `keypair`: název ssh klíče, díky kterému bude možné se na server přihlásit,
- `node_count`: udává počet totožných serverů, které se vytvoří a budou vykonávat danou úlohu,
- `name`: jméno této šablony.

Při vytváření úlohy uživatel zadá tyto parametry:

- `job_template`: jméno šablony, ze které má být úloha vytvořena,
- `keypair`: použití tohoto parametru může přepsat ssh klíč obsažený v šabloně (například, když jednu úlohu zadává více lidí),
- `live_timeout`: doba v minutách, po kterou má být úloha vykonávána.

3.3.1 Životní cyklus úlohy

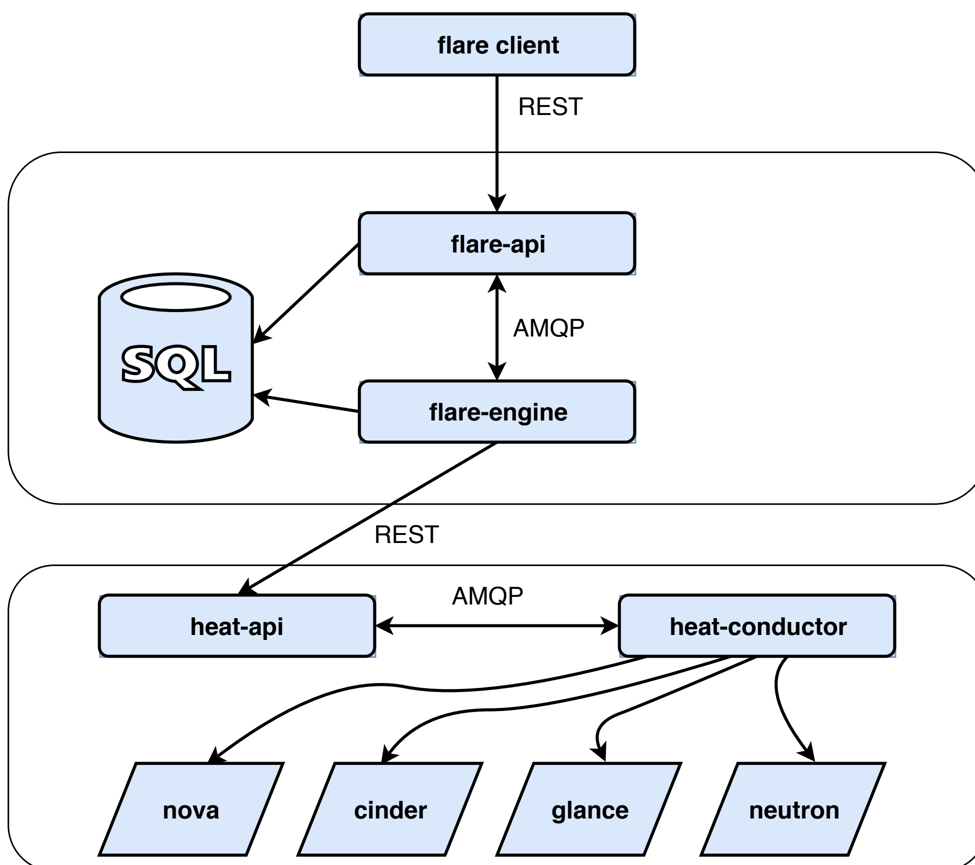


Obrázek 3.4: Diagram životního cyklu jedné úlohy

Životní cyklus těchto úloh je znázorněn na Obrázku 3.4. Pokud není k dispozici žádná grafická karta, tak úloha po vytvoření přejde do stavu `JOB_WAITING`. V opačném případě přejde úloha do stavu `JOB_RUNNING` a virtuální server s připojenou grafickou kartou se vytvoří a začne vykonávat zadanou úlohu. Jakmile dojde k vypršení životnosti úlohy zadané parametrem `live_timeout`, tak přejde úloha do stavu `JOB_FINISHED`, virtuální server se smaže a grafická karta je uvolněna pro další úlohy. Úlohy jsou zadávány do fronty podle stáří, tedy nejdéle čekající úloha dostává k dispozici grafickou kartu jako první.

3.4 Návrh architektury

Projekt by se měl sestávat z několika komponent, jak je znázorněno na Obrázku 3.5. Jednou by mělo být **API**, které bude sloužit jako vstupní bod pro uživatele, který má v plánu spustit požadovanou úlohu. API nadále bude komunikovat s komponentou pojmenovanou jako **Engine**, která bude provádět skutečnou



Obrázek 3.5: Návrh architektury projektu

práci, tedy vytvářet a mazat zadané úlohy. Nedílnou součástí projektu by měl být klient, který bude interagovat s uživatelem a zadávat požadované příkazy API programu Openstack Flare.

3.4.1 Flare API

Jako je tomu i v ostatních komponentách Openstacku, i Flare bude mít *REST API*, které bude sloužit uživateli k interakci s touto komponentou. Toto API tedy bude obsluhovat `GET`, `POST`, `DELETE` a `PATCH` požadavky. API vytvořené v této práci bude mít jednu verzi pojmenovanou jako *v1*. Základní API příkazy (které nebudou vyžadovat autentikaci):

`GET` /
vrátí dostupné verze API

`GET` /v1
vrátí všechny prostředky, které poskytuje daná verze API

Všechna následující volání Flare API už vyžadují autentikaci, která prochází skrze Keystone:

- správa šablon úloh:

  `POST` /v1/jobtemplates
  - vytvoření nové šablony úloh

  `GET` /v1/jobtemplates
  - vrátí všechny šablony úloh

  `GET` /v1/jobtemplates/{`jobtemplate_identificator`}
  - vrátí detaily o dané šabloně

  `DELETE` /v1/jobtemplates/{`jobtemplate_identificator`}
  - smaže danou šablonu úloh

  `PATCH` /v1/jobtemplates/{`jobtemplate_identificator`}
  - upraví danou šablonu úloh

- správa úloh:

  `POST` /v1/jobs
  - vytvoření nové úlohy

  `GET` /v1/jobs
  - vrátí všechny úlohy

  `GET` /v1/jobs/{`job_identificator`}
  - vrátí detaily o dané úloze

  `DELETE` /v1/jobs/{`job_identificator`}
  - smaže danou úlohu

  `PATCH` /v1/jobs/{`job_identificator`}
  - upraví danou úlohu

- správa Flare služeb (může být využito pro hlídání dostupnosti Flare služeb):

  `GET` /v1/fservices
  - vrátí informace o běžících službách Flare Engine

- správa statistik:

  `GET` /v1/stats?project_id=
  - vrátí statistiky o úlohách v daném projektu

  `GET` /v1/stats
  - vrátí celkové statistiky napříč všemi projekty

- správa kvót:

  `POST` /v1/quotas
  - vytvoření nové kvóty

  `GET` /v1/quotas
  - vrátí všechny kvóty

  `GET` /v1/quotas/{project_id}/{resource}
  - vrátí detaily o dané kvótě v daném projektu

  `DELETE` /v1/quotas/{project_id}/{resource}
  - smaže danou kvótu v daném projektu

  `PATCH` /v1/quota/{project_id}/{resource}
  - upraví danou kvótu v daném projektu

- správa dostupných PCI zařízení (grafických karet) v cloudu:

  `GET` /v1/pci_device
  - vrátí počet aktuálně dostupných PCI zařízení (grafických karet)

3.4.2 Flare Engine

Jak již naznačuje Obrázek 3.5, Flare API bude komunikovat s komponentou Flare Engine pomocí protokolu AMQP. To znamená, že každá zpráva bude ukládána do fronty zpráv, ze které si ji poté protistrana vyzvedne a případně na ni odpoví stejným způsobem. Veškerý požadavek, který obdrží Flare API, přejde následně na Flare Engine, který poté provede požadovanou činnost.

Jak je z obrázku zřejmé, Flare Engine bude při snaze klienta o vytvoření úlohy komunikovat pomocí Heat klienta s Heat API. Engine Heatu předá šablonu, kterou Heat zpracuje a postará se o vytvoření serveru. Toho docílí Heat API pomocí komponenty Heat Conductor, která komunikací se službami Cinder, Glance, Nova a Neutron zajistí vytvoření svazku z obrazu a následné připojení tohoto svazku na vytvořený server, ke kterému se připojí požadovaná síť.

Veškeré informace o úlohách, šablonách úloh, kvótách či statistikách bude Flare Engine ukládat do SQL databáze. V této databázi budou také uloženy aktuální stavy jednotlivých úloh.

Nedílnou součástí služby Flare Engine budou periodické funkce, které se postarají o synchronizaci stavu vytvořených úloh, hlídání dostupnosti grafických karet a také o dodržování nepřekročení doby životnosti všech běžících úloh. Jakmile úloha překročí danou životnost, Engine se postará o to, aby *stack*, který byl Heatem při zadání úlohy vytvořen, byl korektně smazán a úloha přešla do stavu `JOB_FINISHED`.

3.4.3 Flare klient

Flare CLI klient bude prostředníkem mezi uživatelem a Flare API. Pomocí klienta bude možné vytvářet šablony úloh, upravovat je, vytvářet samotné úlohy a tyto úlohy samozřejmě i mazat. Další funkcionalitou bude práce s kvótami či zjišťování stavu služby Flare Engine a celkových statistik.

Konkrétně bude mít Flare klient následující možné příkazy:

- `job-create`: se zadanými parametry zašle příkaz k vytvoření úlohy; možné parametry jsou popsány v podsekci 3.3,
- `job-delete`: smaže zadanou úlohu,
- `job-list`: zobrazí seznam a stav všech existujících úloh,
- `job-show`: zobrazí detaily o zadané úloze,
- `job-update`: aktualizuje zadaný parametr dané úlohy,
- `job-template-create`: se zadanými parametry zašle příkaz k vytvoření šablony úloh; možné parametry jsou popsány v podsekci 3.3,
- `job-template-delete`: smaže zadanou šablonu úloh,
- `job-template-list`: zobrazí seznam všech existujících šablon úloh,
- `job-template-show`: zobrazí detaily o zadané šabloně,
- `job-template-update`: aktualizuje zadaný parametr dané šablony,
- `service-list`: zobrazí seznam a stav běžících služeb Flare Engine,
- `available-pci-devices-list`: vrátí počet aktuálně dostupných PCI zařízení (grafických karet),
- `stats-list`: zobrazí statistiky; pokud je zadaný projekt, vypíše statistiky z daného projektu,
- `quotas-create`: vytvoří kvótu se zadanými parametry; možné parametry jsou:
  - `--project-id`: specifikuje, do jakého projektu má kvóta spadat,
  - `--resource`: specifikuje daný prostředek (například úlohu, či šablonu),
  - `--hard-limit`: limit, který nesmí být překročen u zadaného prostředku,
- `quotas-delete`: smaže zadanou kvótu,
- `quotas-list`: zobrazí seznam všech kvót,
- `quotas-show`: zobrazí detaily o dané kvótě,
- `quotas-update`: aktualizuje zadanou kvótu.

Kapitola **4**

Realizace

V této kapitole je popsána realizace projektu. Realizaci samotného projektu Flare předcházelo zprovoznění cloudu Openstack a zároveň možnost virtualizovat grafickou kartu na serverech `compute1` a `compute2`. Po realizaci komponenty Flare se seznámíme s vytvářením vlastních obrazů, které mohou obsahovat kromě čistého systému i ovladače na grafickou kartu, případně vykonávat požadovaný početní úkon.

4.1 Instalace cloudu Openstack

Jak již bylo zmíněno v sekci 3.2, k instalaci systému Openstack byl využit automatizační nástroj Puppet. Díky jeho deklarativnímu jazyku se daly jednoduše nainstalovat nejen všechny komponenty Openstacku, ale zároveň i podpůrné služby jako MySQL, HAProxy, Corosync i RabbitMQ.

Nejprve bylo třeba zprovoznit databázi, která se nainstalovala na stroje `mysql1` až `mysql3`. Do této databáze se nástrojem Puppet vytvořily všechny potřebné tabulky, uživatelé a oprávnění pro tyto uživatele.

Následovalo nainstalování komponenty Keystone, jejíž funkčnost podmiňuje chod dalších komponent. Keystone byl nainstalován Puppetem na servery `keystone1` až `keystone3`. Puppet se postaral i o nainstalování tzv. *endpointů*, což jsou cesty k ostatním službám. Tyto endpointy definují, přes jakou adresu je daná služba (například Cinder, jak můžeme vidět na ukázce ve Výstupu příkazové řádky 4.1) dostupná a na jakém rozhraní (interface) [21]. Například uživatelé mohou přistupovat k Cinderu pouze přes `public` rozhraní, tudíž v našem případě přes URL `http://185.120.69.165:8776/v1/%(tenant_id)s`.

Poté již bylo možné instalovat další komponenty, jako jsou například Nova, Cinder, Glance a další. Díky Puppetu bylo snadné nastavit konfigurační parametry (například cesty k databázi, apod.), tudíž všechny služby byly již nastavené dle našich požadavků.

```
$ openstack endpoint list --service cinder -c "Service Name" -c Interface -c
    URL
+--------------+-----------+-------------------------------------------+
| Service Name | Interface | URL                                       |
+--------------+-----------+-------------------------------------------+
| cinder       | admin     | http://172.27.113.5:8776/v1/%(tenant_id)s   |
| cinder       | internal  | http://172.27.113.5:8776/v1/%(tenant_id)s   |
| cinder       | public    | http://185.120.69.165:8776/v1/%(tenant_id)s |
+--------------+-----------+-------------------------------------------+
```

Výstup příkazové řádky 4.1: Ukázka Keystone endpointů služby Cinder

Instalace softwaru HAProxy a Corosync na servery `lb1` až `lb3` a `publiclb1` až `publiclb3` opět zajistil Puppet, instalace tedy byla snadná. Nastavená vnitřní virtuální IP adresa je `172.27.113.5` a je aktivní na stroji `lb1`. Pokud stroj spadne, adresa se automaticky přehodí díky programu Corosync na další aktivní balancer. Veřejná virtuální IP adresa je `185.120.68.165` a podobným způsobem je aktivní na stroji `publiclb1`. Tyto adresy mohou být k vidění právě na Výstupu příkazové řádky 4.1.

Poté, co byl Openstack úspešně nainstalován, bylo zapotřebí nastavit několik základních věcí, aby byl Openstack připraven k vytvoření virtuálního stroje. Zaprvé bylo třeba vytvořit projekt, uživatele a síť, kterou budeme přistupovat k virtuálním strojům. Následně bylo třeba pomocí Glance do Openstacku nahrát základní systémový obraz, ze kterého se budou dále vytvářet svazky pro virtuální stroje. Zvoleným systémem byl Ubuntu 16.04 Server.

Nakonec byly nainstalovány servery `compute1` a `compute2`. V danou chvíli na nich nebyla prováděna žádna úprava s ohledem na virtualizaci grafických karet, pouze byla otestována funkčnost vytvořením základního virtuálního serveru. V následující sekci se budeme věnovat potřebným úpravám pro chod virtuálních strojů s připojenou grafickou kartou.

4.2 Zprovoznění virtualizace GPU na compute serveru

Námi používaná virtualizační technologie KVM podporuje QEMU, což je open-source virtualizér a emulátor. Tento emulátor umožňuje od Linuxové verze jádra 3.9, pomocí virtualizační technologie Intel® GVT-d, popsané v sekci 3.1.2, předat grafickou kartou až k virtuálnímu stroji. Tomuto mechanismu se říká *PCI passthrough* [22]. Právě technologii Intel® GVT-d jsme vybrali pro tento projekt. Tato technologie by měla docílit stejného výkonu virtualizované grafické karty oproti kartě nevirtualizované - to budeme podrobněji testovat v kapitole 5.

4.2.1 Konfigurace systému a systémových komponent

Nejprve bylo zapotřebí nastavit IOMMU, což je pomocná jednotka správy paměti propojující operační paměť se vstupně-výstupní sběrnicí podporující přímý přístup do paměti [23]. Použití IOMMU umožňuje využít technologii *PCI passthrough*, ale i ochranu paměti proti poškozeným či podezřelým zařízením [22]. Pro správnou funkčnost IOMMU bylo potřeba povolit v nastavení BIOS serveru technologii Intel® VT-d. Samotné IOMMU se povoluje předáním parametru `intel_iommu=on` do jádra systému.

K předávání PCI zařízení virtuálním strojům používá jádro hostitelského systému takzvané IOMMU skupiny. IOMMU skupina je nejmenší možná množina fyzických zařízení, které mohou být předány virtuálnímu stroji [22]. K tomu, abychom mohli předat grafickou kartu modulu `vfio-pci`, který se stará o požadované *PCI passthrough* virtuálnímu stroji, bylo zapotřebí zjistit hodnoty `vendor_id` a `product_id` pro všechna zařízení, která chceme virtuálnímu stroji předat. To je v našem případě grafická karta NVIDIA GeForce RTX 2070, ovšem ta se v systému jeví jako více zařízení, jak můžeme vidět ve Výstupu příkazové řádky 4.2. Hodnoty `vendor_id` a `product_id` jsou uvedeny v hranatých závorkách ve výstupu příkazu `lspci -nnk`. Hexadecimální číslo `10de` označuje `vendor_id`, v našem případě NVIDIA. Pro *VGA controller* je `product_id` číslo `1f02` a pro zvukové zařízení (kterým grafická karta samozřejmě disponuje) je to číslo `10f9`.

```
# lspci -nnk | grep -i nvidia
04:00.0 VGA compatible controller [0300]: NVIDIA Corporation Device [10de:1f02
    ] (rev a1)
        Kernel modules: nvidiafb, nouveau
04:00.1 Audio device [0403]: NVIDIA Corporation Device [10de:10f9] (rev a1)
04:00.2 USB controller [0c03]: NVIDIA Corporation Device [10de:1ada] (rev a1)
04:00.3 Serial bus controller [0c80]: NVIDIA Corporation Device [10de:1adb] (
    rev a1)
```

Výstup příkazové řádky 4.2: Výstup příkazu `lspci`

Hodnoty jak pro *VGA controller*, tak pro zvukové zařízení, bylo třeba předat modulu `vfio-pci` pomocí souboru, který je znázorněn na Výstupu příkazové řádky 4.3.

```
# cat /etc/modprobe.d/vfio.conf
options vfio-pci ids=10de:1f02,10de:10f9
```

Výstup příkazové řádky 4.3: Předané hodnoty modulu `vfio-pci`

Aby však modul `vfio-pci` mohl převzít kontrolu nad grafickou kartou, bylo zapotřebí zakázat některým modulům přístup k této grafické kartě. Tyto moduly mají totiž tendenci převzít kontrolu nad grafickou kartou ještě dříve než modul `vfio-pci`. Konkrétně šlo o moduly `snd_hda_intel` (ten může mít

tendenci převzít kontrolu nad zvukovým zařízením grafické karty) a modul `nouveau`, což je nativní ovladač grafických karet NVIDIA v linuxových systémech.

Samotný název modulu `vfio-pci` bylo poté třeba přidat do souboru `/etc/modules-load.d/modules.conf`, aby mohl být tento modul načten při spuštění systému.

V našem případě byla třeba ještě poslední systémová úprava, neboť grafická karta NVIDIA GeForce RTX 2070 obsahuje port typu USB-C, který se v systému jeví jako USB zařízení, jak znázorňuje Výstup příkazové řádky 4.2. Všechna USB zařízení si ale v linuxovém systému zabírá modul `xhci_hcd`, který si ovšem nemůžeme dovolit zakázat. V takovém případě by totiž nefungoval ani jediný USB port. Zde tedy muselo dojít k jiné úpravě, a to k přidání `vendor_id`, `product_id` a cesty k zařízení získané z příkazu `lspci` do souboru `/etc/rc.local`, způsobem, který je znázorněn ve Výstupu příkazové řádky 4.4.

```
# cat /etc/rc.local
#!/bin/sh -e
#
# rc.local
#

echo "10de 1ada" > /sys/bus/pci/drivers/vfio-pci/new_id
echo "0000:04:00.2" > /sys/bus/pci/devices/0000:04:00.2/driver/unbind
echo "0000:04:00.2" > /sys/bus/pci/drivers/vfio-pci/bind

exit 0
```

Výstup příkazové řádky 4.4: Předání USB zařízení modulu `vfio-pci`

Všechny výše uvedené změny byly zadány do konfiguračních souborů jazyka Puppet, kterým se říká manifesty. Díky tomu jsme schopni snadno tyto změny aplikovat pro další případné `compute` servery automaticky. Jediné, co je potřeba, aby byly změny aplikovány, je server restartovat. Poté se nahrají požadované moduly a nechtěné moduly se v systému zakážou.

4.2.2 Konfigurace v Openstacku

Openstack Nova umožňuje předání PCI zařízení virtuálním strojům pomocí tzv. aliasů. Alias je struktura popisující dané PCI zařízení, které chceme virtuálnímu stroji předat. Tento alias se zadává do konfiguračního souboru na Nova serverech `nova1` až `nova3`. Aliasů je samozřejmě možné zadat více, pokud máme v úmyslu mít více různých PCI zařízení, která chceme virtualizavat [24]. Zde je příklad aliasu, který umožní virtualizaci naší grafické karty NVIDIA GeForce RTX 2070 (`vendor_id` a `product_id` jsou údaje *VGA controlleru* získaného z Výstupu příkazové řádky 4.2):

```
alias = {
  "vendor_id": "10de",
  "product_id": "1f02",
  "device_type": "type-PCI",
  "name": "rtx2070"
}
```

Na samotném `compute` serveru, který je nosičem grafické karty, bylo zapotřebí nastavit tzv. `passthrough_whitelist`, což je také struktura, zde obsahující pouze `vendor_id` a `product_id`, která umožní přístup Nově ke grafické kartě. V našem případě tedy hodnota `passthrough_whitelist` vypadá následovně:

```
passthrough_whitelist = {
  "vendor_id": "10de",
  "product_id": "1f02"
}
```

Posledním krokem bylo vytvoření tzv. `flavor`, blíže popsaného v sekci 1.5. Tento `flavor` totiž nemusí obsahovat jen informace o počtu virtuálních CPU jader a velikost RAM paměti, kterou má virtuální stroj obdržet, ale zároveň i informace o PCI zařízení, které má virtuální stroj dostat k dispozici [24]. Toho lze docílit nastavením vlastnosti (*property*) a předáním správného aliasu a počtu takových zařízení.

```
$ openstack flavor show 16CPU_12GB_1GPU-2070
+----------------------------+-------------------------------------+
| Field                      | Value                               |
+----------------------------+-------------------------------------+
| OS-FLV-DISABLED:disabled   | False                               |
| OS-FLV-EXT-DATA:ephemeral  | 0                                   |
| access_project_ids         | None                                |
| disk                       | 0                                   |
| id                         | aaba2d64-8a82-45fe-bcbd-ccea2c4c1d94 |
| name                       | 16CPU_12GB_1GPU-2070                |
| os-flavor-access:is_public | True                                |
| properties                 | pci_passthrough:alias='rtx2070:1'   |
| ram                        | 12288                               |
| rxtx_factor                | 1.0                                 |
| swap                       |                                     |
| vcpus                      | 16                                  |
+----------------------------+-------------------------------------+
```

Výstup příkazové řádky 4.5: `flavor` obsahující informace o grafické kartě

Jak vypadá takový `flavor`, můžeme vidět na Výstupu příkazové řádky 4.5, kde vidíme `flavor` nastavený na 16 virtuálních CPU jader, 12 GB RAM a jednu grafickou kartu NVIDIA GeForce RTX 2070. Číslo za dvojtečkou v parametru `properties` u daného `flavor` značí právě počet těchto zařízení. Jelikož máme k dispozici servery, které pojmou pouze jednu takovou grafickou kartu, bude toto číslo vždy nastaveno na 1.

Po těchto úpravách již bylo možné vytvořit virtuální server s připojenou grafickou kartou. Karta se korektně zobrazovala v systému virtuálního serveru, jak znázorňuje Výstup příkazové řádky 4.6 a po nainstalování CUDA ovladačů (obsahující i ovladače grafické karty) následně i korektně fungoval příkaz `nvidia-smi`, zobrazující informace o grafické kartě, jak znázorňuje Výstup příkazové řádky 4.7. Z tohoto výstupu můžeme vyčíst jak běžící procesy využívající grafickou kartu, tak aktuální spotřebu, využití paměti a další. Virtuální stroj tedy převzal plnou kontrolu nad grafickou kartou. Její nosič naopak už kontrolu nemá žádnou.

```
# lspci -nnk | grep -i nvidia
00:06.0 VGA compatible controller [0300]: NVIDIA Corporation Device [10de:1f02
    ] (rev a1)
        Kernel driver in use: nvidia
        Kernel modules: nvidia_410_drm, nvidia_410
```

Výstup příkazové řádky 4.6: Výstup příkazu `lspci` z virtuálního stroje

```
# nvidia-smi
Wed Jan  2 21:45:48 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 410.79       Driver Version: 410.79       CUDA Version: 10.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce RTX 2070    Off  | 00000000:00:06.0 Off |                  N/A |
| 22%   37C    P0     1W / 175W |      0MiB /  7952MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Výstup příkazové řádky 4.7: Výstup příkazu `nvidia-smi` z virtuálního stroje

4.3 Flare

Jako i ostatní komponenty Openstacku, i Flare byl napsán v programovacím jazyce Python. Bylo využito i mnoho stejných *frameworků*, aby byla dodržena soudružnost s ostatními projekty. Projekt byl testován na virtuálním stroji `flare1`, bežícím na nosiči `control1`. V následujících sekcích se podrobněji seznámíme s jednotlivými částmi projektu.

4.3.1 Konfigurační soubor

V analogii s ostaními projekty Openstacku i Flare používá svůj konfigurační soubor `/etc/flare/flare.conf`. Je možné konfigurovat například cestu k databázi, ke Keystone serverům, či například číslo portu, na kterém má poslouchat služba `flare-api`. Pro využití konfiguračního souboru byla využita Python knihovna `oslo.config`, což je Openstack knihovna sloužící k zjišťování konfiguračních hodnot z konfiguračního souboru. Knihovna umožňuje zadat i defaultní hodnoty, aby nebylo nutné do konfiguračního souboru vypisovat hodnoty, které jsou často neměnné. Pro přehlednost konfiguračního souboru jsou jednotlivé proměnné shlukovány do skupin dle užití, například konfigurační proměnné Flare API jsou pod skupinou `[api]`.

4.3.2 Objekty a databáze

Každý element v programu Flare je reprezentován jako objekt. Objekty tedy jsou `Job`, `JobTemplate`, `FlareService`, `PciDevice`, `Quota` a `Stats`. Každý objekt obsahuje prvky (*fields*), tyto prvky reprezentují jednotlivé parametry daného objektu, které jsou zároveň stejnojmenně uloženy i v databázi. Prvky jednotlivých objektů jsou následující (vedle názvu prvku je popsán datový typ prvku):

- `Job`:

```
'id': fields.IntegerField(),
'uuid': fields.UUIDField(nullable=True),
'name': fields.StringField(nullable=True),
'project_id': fields.StringField(nullable=True),
'user_id': fields.StringField(nullable=True),
'job_template_id': fields.StringField(nullable=True),
'keypair': fields.StringField(nullable=True),
'stack_id': fields.StringField(nullable=True),
'status': f_fields.JobStatusField(nullable=True),
'status_reason': fields.StringField(nullable=True),
'live_timeout': fields.IntegerField(nullable=True),
'job_template': fields.ObjectField('JobTemplate'),
'auth_token_info': fields.StringField(nullable=True),
'auth_token': fields.StringField(nullable=True)
```

  – většina těchto prvků již byla popsána v sekci 3.3, významem prvků `auth_token` a `auth_token_info` se budeme podrobněji zabývat v podsekci 4.3.4, význam ostatních je zjevný z jejich názvu,

- `JobTemplate`:

```
'id': fields.IntegerField(),
'uuid': fields.UUIDField(nullable=True),
'project_id': fields.StringField(nullable=True),
'name': fields.StringField(nullable=True),
'image_id': fields.StringField(nullable=True),
'flavor_id': fields.StringField(nullable=True),
'keypair_id': fields.StringField(nullable=True),
'network_id': fields.StringField(nullable=True),
'volume_size': fields.IntegerField(nullable=True),
'node_count': fields.IntegerField(nullable=True)
```

  – u tohoto objektu jsou všechny prvky popsány v sekci 3.3,

- `FlareService`:

```
'id': fields.IntegerField(),
'host': fields.StringField(nullable=True),
'binary': fields.StringField(nullable=True),
'disabled': fields.BooleanField(),
'disabled_reason': fields.StringField(nullable=True),
'last_seen_up': fields.DateTimeField(nullable=True),
'forced_down': fields.BooleanField(),
'report_count': fields.IntegerField(),
```

  – tento objekt slouží ke zjišťování stavu služeb `flare-engine`, jsou zde tedy například informace o serveru, na kterém je daná služba spuštěna či informace o tom, kdy naposledy se služba hlásila jako v pořádku,

- `Quota`:

```
'id': fields.IntegerField(),
'project_id': fields.StringField(nullable=False),
'resource': fields.StringField(nullable=False),
'hard_limit': fields.IntegerField(nullable=False),
```

  – prvek `resource` může být typu `Job` nebo `JobTemplate` - zamezujeme tedy touto kvótou uživatelům k vytvoření příliš mnoha úloh či šablon úloh; ostatní prvky jsou zřejmé,

- `Stats`:

```
'jobs': fields.IntegerField()
```

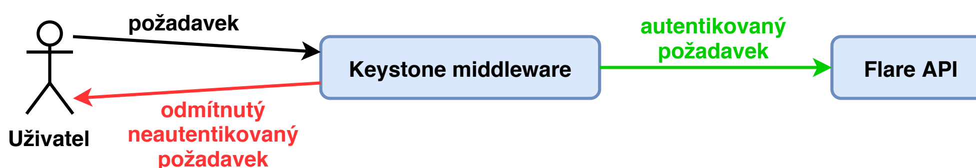
  – zde máme pouze jeden prvek a to počet aktuálních úloh v systému.

Objekty samotné mají implementovány metody pro práci s databází. S databází komunikují pomocí databázového API, které je implementováno díky SQLAlchemy, což je soubor nástrojů pro Python pro práci s SQL databází [25]. SQLAlchemy se stará o veškerou komunikaci s MySQL databází, provozovanou na serverech `mysql1` až `mysql3`. SQLAlchemy pracuje s modely, které reprezentují daný objekt. Na tyto modely poté vykonává požadované SQL příkazy. Samotné Flare API a Flare Engine už poté pracují pouze s objekty a jejich prvky, ty se postarají o veškerou databázovou činnost.

4.3.3 API

Flare API využívá Python webového frameworku Pecan, který umožňuje zpracování HTTP požadavků. API bylo implementováno dle návrhu v podsekci 3.4.1. K implementaci všech částí API bylo využito tzv. *controllerů*. Pecan využívá k mapování HTTP požadavku ke správnému controlleru směrovací strategii [26]. Každý prvek popsaný v podsekci 3.4.1 tedy obsahuje vlastní controller a v něm jsou implementované jednotlivé `GET`, `POST`, `DELETE` a `PATCH` metody.

Autentikace bylo dosaženo pomocí využití Openstack knihovny `keystonemiddleware`, díky které jsme schopni autentikovat všechny neveřejné požadavky, pouze cesty `/` a `/v1` jsou veřejné. U všech ostatních se musí uživatel autentikovat pomocí Keystone. Je například ověřeno, že se dotazuje API na informace o projektu, ke kterému má přístup. Keystone middleware tedy slouží jako vstupní brána k našemu programu, jak znázorňuje Obrázek 4.1.



Obrázek 4.1: Znázornění autentikace požadavků

Každý API controller pracuje s objekty popsanými v předchozí části. Jako první úkol při zpracovávání každého požadavku je zkontrolování tzv. *policy* souboru. To je soubor ve formátu JSON umístěný ve složce `/etc/flare`. Tento soubor obsahuje informace o tom, kdo může provádět požadovaný `GET`, `POST`, `DELETE` či `PATCH` příkaz. Jde o soubor využívaný i v ostatních komponentách Openstacku [27]. Na následujícím příkladu, jak takový soubor může vypadat pro operaci s úlohami (operace se šablonami, statistikami, kvótami a další je analogická), můžeme vidět zadefinování role `admin_or_owner`, která definuje uživatele, který je buď administrátor, nebo má přístup do daného projektu:

```
{
    "admin_or_owner":  "is_admin:True or project_id:%(project_id)s",

    "job:create": "rule:admin_or_owner",
    "job:delete": "rule:admin_or_owner",
    "job:detail": "rule:admin_or_owner",
    "job:get": "rule:admin_or_owner",
    "job:get_all": "rule:admin_or_owner",
    "job:update": "rule:admin_or_owner"
}
```

Po zkontrolování restrikcí zadaných v souboru `policy.json` již dojde k provedení požadované akce. Například v případě požadavku o vytvoření úlohy dojde k validaci zadaných parametrů a následné volání RPC API, což je

API, které slouží jako AMQP klient ke komunikaci se službou Flare Engine. Této službě právě API předá informace o vytvářené úloze a zadá jí příkaz ke skutečnému vytvoření úlohy.

Naproti tomu při vytváření šablony úloh je malý rozdíl v tom, kdo skutečně vytvoří objekt. Jelikož šablona úloh je akce nevyžadující žádnou další akci (vytvoření Heat stacku či zařazení úlohy do fronty), není třeba, aby objekt vytvářel Flare Engine, ale postará se o to samotné API.

4.3.4 Engine

Engine má za úkol vytváření úloh, zařazování těchto úloh do fronty, pokud není k dispozici grafická karta, případně vytvoření Heat stacku, který bude obsahovat virtuální stroj s připojenou grafickou kartou.

Engine využívá AMQP protokolu pro komunikaci s Flare API. Dostane od API zprávu s požadavkem na zpracování, např. na vytvoření úlohy. Flare Engine běží jako samostatná služba (`flare-engine`) a má tři hlavní úkoly: vytvoření úlohy, úprava úlohy a smazání úlohy.

Vytvoření úlohy spočívá nejprve ve zjištění, zda je dostupný požadovaný počet grafických karet (`node_count` parametr obsažený v šabloně úlohy). Aktuální stav grafických karet evidovaný komponentou Nova můžeme zjistit z databáze. V `nova` databázi je totiž tabulka `pci_devices`, která obsahuje informace o všech PCI zařízeních v systému Openstack, ať už zabraných či dostupných, viz. Výstup příkazové řádky 4.8. V našem případě se samozřejmě jedná o grafické karty.

```
MariaDB [nova]> select compute_node_id, address, product_id, vendor_id, status
     from pci_devices;
+-----------------+-------------+------------+-----------+-----------+
| compute_node_id | address     | product_id | vendor_id | status    |
+-----------------+-------------+------------+-----------+-----------+
|              13 | 0000:04:00.0 | 1f02       | 10de      | allocated |
|              18 | 0000:04:00.0 | 1f02       | 10de      | allocated |
+-----------------+-------------+------------+-----------+-----------+
```

Výstup příkazové řádky 4.8: Databázové informace o grafických kartách

Ke zjištění počtu dostupných grafických karet je využito také SQLAlchemy, zde je ovšem vytvořena speciální *session* (spojení k databázi), neboť toto je jediný případ operace s jinou databází než s databází `flare`. Zde je nutné připojení k databázi `nova`, cestu k této databázi lze definovat parametrem `nova_connection` ve skupině `[database]` v konfiguračním souboru `/etc/flare/flare.conf`. Objekt `PciDevice` obsahje metodu `get_available_pci_devices()`, která se zeptá databáze na počet PCI zařízení, která jsou ve stavu *available* a vrátí tuto informaci. Pokud není dostupný požadovaný počet grafických karet, úloha přejde do stavu `JOB_WAITING`. V opa-

čném případě je úloha uvedena do stavu `JOB_RUNNING` a je zavolána třída `HeatDriver`, která obstarává práci se *stackem*. Ta se postará díky knihovně `heatclient` o vytvoření stacku se zadanými parametry. Zároveň je nastaven prvek `stack_id` u dané úlohy, abychom měli informaci o tom, jaký *stack* je součástí dané úlohy. Toho můžeme využít například u mazání úlohy, kdy zjišťujeme, zda se skutečně smazal daný *stack* a můžeme opravdu úlohu označit jako smazanou.

Nedílnou součástí Flare Engine jsou periodické úkony. To jsou metody, které jsou pravidelně spouštěné. V našem případě využíváme čtyři pravidelné periodické metody:

- `sync_deleted_jobs()`: Tato metoda se stará o úlohy, které máme v plánu smazat. Poté, co klient zadá příkaz ke smazání úlohy, přejde úloha do stavu `DELETE_IN_PROGRESS`. Metoda vyhledává úlohy v tomto stavu a zjišťuje, zda je Heat *stack* korespondující s danou úlohou již smazán. Pokud ano, úlohu můžeme úspěšně označit za smazanou a odstranit informace o této úloze z databáze. Tato metoda běží každých 30 sekund.

- `sync_time_exceeded_jobs()`: Tato metoda se stará o úlohy, které již běží, tudíž zabírají jednu či více grafických karet, a hlídá, zda nepřekročily svoji životnost, tedy prvek `live_timeout` obsažený v objektu `Job`. Toho je docíleno pomocí Openstack knihovny `oslo.utils`, která obsahuje metodu `timeutils.is_older_than()`, a zároveň prvku `updated_at` obsaženého v objektu `Job` [28]. Tento prvek představuje okamžik, kdy byl změněn status dané úlohy - k tomu mohlo dojít pouze změnou ze stavu `JOB_WAITING` do stavu `JOB_RUNNING`, tedy uvolněním jedné nebo více grafických karet a skutečné spuštění dané úlohy. Metodě `timeutils.is_older_than()` tedy předáme tento údaj a `live_timeout` a jsme schopni zjistit, zda došlo k překročení doby životnosti. Pokud se tak stalo, úloha přejde do stavu `JOB_FINISHED`, dojde ke smazání *stacku* (aby došlo ke skutečnému uvolnění jedné či více grafických karet) a čeká se na uživatele, aby úlohu smazal.

- `sync_job_status()`: Tato metoda hlídá úlohy, které jsou ve stavu `JOB_WAITING`, tedy úlohy, které neměly dostupné požadované prostředky v době svého vytváření. Jakmile se uvolní požadovaný počet grafických karet (zjištujeme pomocí metody `get_available_pci_devices()`), začne proces podobný procesu vytváření úlohy jako takové. Tedy dojde k vytvoření *stacku* s požadovanými parametry a ke spuštění požadované úlohy. Úloha tedy přejde do stavu `JOB_RUNNING`, současně je změněn prvek `updated_at`, tudíž od tohoto okamžiku je počítána doba životnosti dané úlohy. Co se fronty úloh ve stavu `JOB_WAITING` týče, jako první je vybrána nejstarší taková úloha. Prioritou je zde čas.

- `sync_available_pci_devices()`: Tato metoda slouží pouze jako pomocná. Každých 10 vteřin totiž zapisuje do logu (výstup příkazové řádky nebo log soubor) aktuální počet dostupných grafických karet. K logování dochází pouze, pokud je logovací úroveň nastavená na úroveň `DEBUG`.

4.3.5 Klient

Implemetace Flare CLI klienta byla inspirována jinými projekty Openstacku kvůli zachování soudržnosti. Byla implementována verze `v1` korespondující s verzí Flare API. Základním pomocným příkazem pro uživatele je příkaz `flare help`, který vypíše mimo jiné na příkazovou řádku možné argumenty, jak je vidět na Výstupu příkazové řádky 4.9.

```
Positional arguments:
  <subcommand>
    job-create          Create a job.
    job-delete          Delete specified job.
    job-list            Print a list of available jobs.
    job-show            Show details about the given job.
    job-update          Update information about the given job.
    job-template-create
                        Create a job template.
    job-template-delete
                        Delete specified job template.
    job-template-list   Print a list of job templates.
    job-template-show   Show details about the given job template.
    job-template-update
                        Updates one or more job template attributes.
    available-pci-devices-list    Show number of available PCI devices
    service-list        Print a list of flare services.
    stats-list          Show stats for the given project_id
    quotas-create       Create a quota.
    quotas-delete       Delete specified resource quota.
    quotas-list         Print a list of available quotas.
    quotas-show         Show details about the given project resource quota.
    quotas-update       Update information about the given project resource
                        quota.
    bash-completion     Prints arguments for bash-completion. Prints all of
                        the commands and options to stdout so that the
                        flare.bash_completion script doesn't have to hard code
                        them.
    help                Display help about this program or one of its
                        subcommands.
```

Výstup příkazové řádky 4.9: Část výstupu příkazu `flare help`

Použitím každého z těchto argumentů za příkazem `flare help` se vypíší potřebné informace k provedení takového příkazu. Například použití příkazu `flare help job-create` vypíše informace znázorněné na Výstupu příkazové řádky 4.10. Nepovinné parametry jsou uvedeny v hranatých závorkách, v případě jejich neuvedení se použijí defaultní hodnoty.

```
$ flare help job-create
usage: flare job-create --job-template <job_template>
                        [--keypair <keypair>] [--live-timeout <live-timeout>]
                        [<name>]

Create a job.

Positional arguments:
  <name>                Name of the job to create.

Optional arguments:
  --job-template <job_template>
                        ID or name of the job template.
  --keypair <keypair>   Name of the keypair to use for this job.
  --live-timeout <live-timeout>
                        The timeout for job liveness in minutes. The default
                        is 60 minutes.
```

Výstup příkazové řádky 4.10: Výstup příkazu `flare help job-create`

```
$ flare --os-project-id 389e8d2d5b434eb4943601599a7c42a6 job-create --job-
    template testtemplate1 --keypair cilamich --live-timeout 10 testjob1
Request to create job 5a2dc14e-bfc4-43f6-967b-88cc8176c655 has been accepted.
$ flare job-show 5a2dc14e-bfc4-43f6-967b-88cc8176c655
+-----------------+------------------------------------+
| Property        | Value                              |
+-----------------+------------------------------------+
| status          | JOB_WAITING                        |
| uuid            | 5a2dc14e-bfc4-43f6-967b-88cc8176c655 |
| stack_id        | -                                  |
| created_at      | 2019-01-04T00:21:51+00:00          |
| updated_at      | 2019-01-04T00:21:51+00:00          |
| live_timeout    | 10                                 |
| job_template_id | a850fbb0-2ffa-4b67-81e3-c9ea8c08e81c |
| keypair         | cilamich                           |
| status_reason   | -                                  |
| name            | testjob1                           |
+-----------------+------------------------------------+
```

Výstup příkazové řádky 4.11: Vytváření úlohy v daném projektu

Při vytváření úlohy je možné zadat argument `--os-project-id` a k němu ID projektu, ve kterém chceme danou úlohu vytvářet. Ve Výstupu příkazové řádky 4.11 si můžeme všimnout použití takového příkazu a následné zobrazení informace o vytvořené úloze. V tomto případě je úloha ve stavu `JOB_WAITING`, nebyly totiž v danou chvíli dostupné prostředky. Je tedy zřejmé, že atribut `stack_id` neobsahuje žádnou hodnotu, neboť požadovaný *stack* ještě nemohl být vytvořen.

4.4 Vytváření vlastních obrazů

Je zřejmé, že základní obraz se systémem Ubuntu 16.04 Server nám nebude pro naše účely dostačující. Používat takový obraz pro každý virtuální server s připojenou grafickou kartou by například znamenalo nutnost instalovat ovladače grafické karty pokaždé, když se nám virtuální stroj vytvoří, což je časově náročná činnost. Nabídla se tedy myšlenka vytvoření vlastního obrazu, který již bude obsahovat požadované ovladače. K tomuto byl využit nástroj Packer, což je nástroj pro tvorbu vlastních obrazů napříč různými platformami, mezi které se řadí i Openstack [29].

Packer funguje na principu šablon. Do šablony zadáme požadovanou akci (například pomocí `shell` příkazů) a z této šablony se vytvoří v Openstacku virtuální server z původního obrazu (v našem případě obraz se systémem Ubuntu 16.04 Server), provede příkazy uvedené v šabloně a následně ze svazku připojeného na tento server vytvoří obraz. Tento obraz můžeme tedy chápat jako *snapshot* z daného okamžiku uložený do obrazu. Obraz nám následně poslouží jako vzor pro další virtuální servery.

Šablona je ve formátu JSON a sestává se ze dvou částí:

- `builders`: tato část obsahuje informace o daném prostředí, tedy o našem cloudu; konkrétně je třeba zadat tyto parametry:
  - `type`: název platformy, pro kterou obraz vytváříme - v našem případě `openstack`,
  - `ssh_username`: uživatel, ke kterému budeme pomocí SSH klíče přistupovat - pro Ubuntu systém jde o uživatele ubuntu,
  - `image_name`: název nově vytvořeného obrazu,
  - `source_image`: název nebo ID zdrojového obrazu, ze kterého chceme nový obraz vytvořit,
  - `flavor`: Nova *flavor*, který použijeme pro vytvoření dočasného serveru - zde je důležité použít flavor s nastaveným parametrem `disk`, neboť Packer vytváří tzv. *ephemeral* servery (popsané v sekci 1.5), které nemají připojený persistentní svazek,
  - `domain_name`: název domény, ve které chceme obraz vytvořit,

  - `networks`: síť, která má být použita při vytváření dočasného serveru,
  - `security_groups`: bezpečnostní skupiny použité při vytváření dočasného serveru,
  - `image_visibility`: příznak, zda má být nově vytvořený obraz dostupný napříč všemi projekty.

- `provisioners`: tato část obsahuje informace o tom, co se skutečně stane, jakmile je dočasný virtuální stroj ze zdrojového obrazu vytvořen; může být dvojího typu: `file` a `shell`:
  - `file`: umožňuje po vytvoření dočasného virtuální serveru spustit na tomto serveru skript ze souboru; je třeba zadat tyto parametry:
    * `type`: typ prostředku - v tomto případě `file`,
    * `destination`: složka, do které má být soubor umístěn,
    * `source`: cesta k souboru - tato cesta musí být dostupná ze složky, ve které je šablona pro Packer,
  - `shell`: tento typ umožňuje tzv. *inline* zadat příkazy, které se mají provést po vytvoření dočasného virtuálního serveru; je třeba zadat tyto parametry:
    * `type`: typ prostředku - v tomto případě `shell`,
    * `inline`: zde zadáváme množinu příkazů, které se mají provádět po vytvoření virtuálního serveru.

Vytvořením obrazu obsahujícího platformu CUDA (obsahující i ovladače grafické karty NVIDIA GeForce RTX 2070) jsme docílili použitím následující Packer šablony (cesty k souborům jsou pro přehlednost zkráceny znaky `...`):

```
{
  "builders": [{
      "type": "openstack",
      "ssh_username": "ubuntu",
      "image_name": "ubuntu-xenial-cuda",
      "source_image": "ubuntu-xenial-cloud",
      "flavor": "PACKER_2CPU_2GB_10DISK",
      "domain_name": "projects",
      "networks": "93f79c5f-54ad-4e21-b45f-3f302845aa55",
      "security_groups": ["default"],
      "image_visibility": "public"
  }],
  "provisioners": [
    {
      "type" : "shell",
      "inline": [
        "sudo wget http://...nvidia.com/.../cuda-repo-...-.deb",
        "sudo dpkg -i cuda-repo-...-.deb",
        "sudo apt-key adv --fetch-keys http://...nvidia.com/.../7fa2af80.pub",
        "sudo apt-get update",
        "sudo apt-get install -y cuda"
      ]
    }
  ]
}
```

Kapitola **5**

Testování a diskuse

5.1 Srovnání výkonnosti GPU

Pro srovnání výkonnosti GPU připojené na virtuální stroj versus GPU připojené přímo na server byly využity výkonnostní testy obsažené v knihovně CUDA [30]. Bylo použito několik testů, konkrétně:

- *UnifiedMemoryPerf*: tento test demonstruje výkonnostní srovnání sjednocené paměti a jiných typů paměti jako například tzv. *zero-copy* bufferů, stránkovatelné (*pageable*) či tzv. *page-locked* paměti na jedné grafické kartě,
- *MatrixMul_nvrtc*: tento test demonstuje násobení dvou matic za použití knihovny `libNVRTC`,
- *Bandwidth Test*: tento test měří propustnost kopírování paměti; měří rychlost kopírování paměti z hosta (serveru) do zařízení (grafické karty), ze zařízení do hosta a ze zařízení do zařízení.

5.1.1 Test UnifiedMemoryPerf

Tento test byl spuštěn na souvislé kousky paměti o velikosti 1024, 4096, 16384 a 65536 kilobajtů. Test obsahuje tyto položky:

- *UMhint* - pracuje s řízenou pamětí a s nápovědami (tyto nápovědy slouží aplikaci k detekci skutečné použitelnosti dat [31]),
- *UMhntAs* - podobný přístup k paměti jako v předchozím případě, s tím rozdílem, že zde se k paměti přistupuje asynchronně,
- *UMeasy* - opět podobný přístup k paměti jako v předchozích dvou případech, zde však nedochází k nápovědám ohledně použitelnosti dat,

- *0Copy* - tento přístup k paměti pracuje s technologií *zero-copy* - tedy přístup, kde jsou data alokována na procesoru a grafická karta k nim přistupuje při každé operaci; nedochází tedy k žádnému přenosu paměti,
- *MemCopy* - v tomto případě už dochází k přenosu stránkovatelné paměti od paměťového prostoru hosta do paměťového prostoru zařízení (grafické karty),
- *CpAsync* - podobný přístup jako v předchozím případě, zde však dochází k asynchronnímu přístupu k paměti,
- *CpHpglk* - oproti *MemCopy* přístupu je zde paměť tzv. *page-locked* - tato paměť může urychlit samotné kopírování z/do hosta/zařízení [32],
- *CpPglAs* - oproti předchozímu přístupu zde k dochází k asynchronnímu přístupu k paměti.

Výsledky testu jsou znázorněny v tabulce 5.1. Tabulka obsahuje výsledky měření prováděných na serveru, ke kterému byla přímo připojena grafická karta NVIDIA GeForce RTX 2070, a výsledky měření prováděných na virtuálním stroji, přičemž grafická karta NVIDIA GeForce RTX 2070 byla virtualizována pomocí metody Intel® GVT-d (více o této metodě v podsekci 3.1.2). Z výsledků je patrné, že virtualizace pomocí metody Intel® GVT-d skutečně přináší požadovaný výkon. Nevirtualizovaná grafická karta má sice ve většině výsledků testů lepší čas, avšak rozdíly jsou ve většině případů mírné.

5.1.2 Test MatrixMul_nvrtc

V tomto testu bylo provedeno násobení dvou čtvercových matic stejných rozměrů. Jednalo se o matice velikosti 128, 256, 512, 1024, 2048, 4096 a 8192. Doby těchto výpočtů jsou zanesené v tabulce 5.2. Ani zde nevidíme žádný výrazný pokles výkonu virtualizované grafické karty oproti kartě nevirtualizované.

5.1.3 Bandwidth test

Tento test byl proveden na různé velikosti přenášené paměti. Konkrétně byly testovány velikosti 128, 256, 512, 1024 a 2048 MB. Zjištěné propustnosti jsou znázorněny v Tabulce 5.3 pro nevirtualizovanou grafickou kartu a v Tabulce 5.4 pro kartu virtualizovanou. Zkratky **H** a **D** v popiscích sloupců reprezentují hosta (**H**ost) a zařízení (**D**evice).

Zde už vidíme rozdíl ve výkonech karty virtualizované oproti kartě nevirtualizované. Virtualizovaná karta vykazuje horší výsledky v přenášení paměti z hosta do zařízení i naopak. Toto je způsobeno režií spojenou s virtualizací RAM paměti. Propustnost přenášené paměti ze zařízení do zařízení je pochopitelně stejná, zde nedochází k žádnému kontaktu s hostem (serverem).

Velikost [kB]	Virtuální stroj	Fyzický stroj	Virtuální stroj	Fyzický stroj
	UMhint [ms]		UMhntAs [ms]	
1024	**3,632**	3,833	**3,097**	3,49
4096	14,164	**14,033**	**13,251**	13,26
16384	**62,15**	62,629	59,068	**55,158**
65536	300,402	**279,876**	287,766	**271,421**
	UMeasy [ms]		0Copy [ms]	
1024	4,423	**3,762**	3,652	**3,535**
4096	17,238	**16,746**	45,403	**25,749**
16384	83,432	**74,783**	235,814	**222,867**
65536	381,044	**358,656**	1966,606	**1811,66**
	MemCopy [ms]		CpAsync [ms]	
1024	**3,038**	3,049	**2,923**	2,982
4096	13,455	**12,019**	14,856	**12,277**
16384	67,309	**62,436**	69,431	**65,253**
65536	318,845	**311,036**	312,766	**309,623**
	CpHpglk [ms]		CpPglAs [ms]	
1024	2,011	**1,852**	2,047	**2,002**
4096	11,449	**10,016**	**10,8**	11,017
16384	50,686	**51,234**	50,086	**47,025**
65536	261,573	**251,974**	261,224	**251,109**

Tabulka 5.1: UnifiedMemoryPerf test

Rozměr matice	Doba výpočtu - virtuální stroj [s]	Doba výpočtu - fyzický stroj [s]
128	1,865	1,506
256	1,577	1,598
512	2,205	1,620
1024	2,391	1,920
2048	6,274	5,740
4096	39,093	38,316
8192	299,91	298,921

Tabulka 5.2: `MatrixMul_nvrtc` test

Velikost paměti [MB]	H → D [MB/s]	D → H [MB/s]	D → D [MB/s]
128	10005,2	9720	378871,4
256	9949,7	9350,8	379335,4
512	10222,5	8709,8	378740,6
1024	9975,7	9459,2	379860,6
2048	10221,4	9736,5	379669,7

Tabulka 5.3: Bandwidth test - GPU na serveru

Velikost paměti [MB]	H → D [MB/s]	D → H [MB/s]	D → D [MB/s]
128	8720,8	6391,1	378514
256	8635,6	5317,1	379356,7
512	8933,1	5389,1	381550,6
1024	8867,6	7591,8	380545,3
2048	8875,2	7531,9	380731,3

Tabulka 5.4: Bandwidth test - GPU ve virtuálním stroji

5.1.4 Shrnutí

Ve většině případů je očekávána zhruba stejná vykonnost virtualizované grafické karty jako u karty nevirtualizované. Je to dáno tím, že technologie Intel® GVT-d, popsaná v sekci 3.1, umožňuje virtuálnímu stroji přímý přístup ke grafické kartě. Hostitelský stroj má ke kartě přístup zakázán, plnou kontrolu nad ní má jen stroj virtuální, nedochází tedy k žádné režii spojené s virtualizací grafické karty.

K režii může docházet při kopírování či přenášení paměti z virtuálního stroje ke grafické kartě i naopak. To je však způsobeno režií virtualizace RAM paměti, o kterou se stará nosič virtuálního serveru.

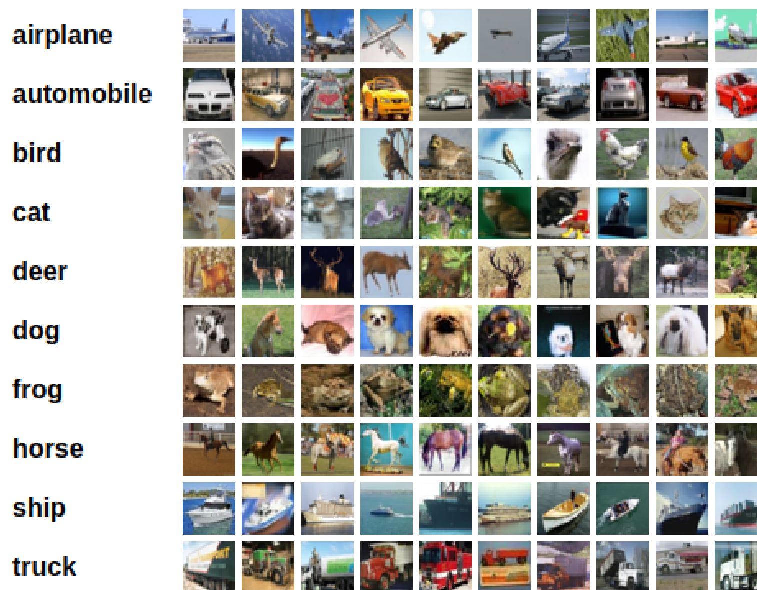
5.2 Ukázkové trénování neuronové sítě

Pro demonstraci funkčnosti a výkonnosti grafické karty v našem GPU cloudu bylo využito trénování neuronové sítě na datasetu `CIFAR-10`. Kód neuronové sítě společně s tímto datasetem je volně dostupným prostředkem pro testování výkonu grafické karty [33].

`CIFAR-10` dataset obsahuje 60 tisíc barevných obrázků o rozměrech 32 x 32 pixelů. Tyto obrázky jsou rozděleny do deseti tříd podle objektu, který zobrazují. Každá taková třída obsahuje 6 tisíc obrázků, přičemž 5 tisíc je určeno pro trénovací účely a 1 tisíc je určen pro testovací účely. Třídy jsou následujících typů: letadla, automobily, ptáci, kočky, srnky, psi, žáby, koně, lodě a nákladní vozidla. Tyto třídy se navzájem neprolínají [34]. Náhodný výběr deseti obrázků z každé třídy je znázorněn na Obrázku 5.1 (zdroj: [34]).

Trénování neuronové sítě bylo spuštěno nejen na virtuálním stroji, ale i na stroji fyzickém, abychom mohli porovnat výkonnost na stejné grafické kartě. Trénovací obrázky jsou grafické kartě předávány po dávkách. Jedna dávka obsahuje 128 obrázků. Program je spouštěn v krocích (defaultně je nastaveno 80 tisíc kroků). Každým krokem se neuronová síť vylepšuje. Program pravidelně po deseti krocích vypisuje, kolik vzorků grafická karta zpracovává za vteřinu. Po 100 krocích vypisuje informace o tom, v jakém stavu se natrénovaná síť nachází, tedy jak je v danou chvíli vytrénovaná.

Naším měřítkem výkonnosti je počet zpracovaných vzorků za vteřinu. Grafická karta na fyzickém stroji zvládla zpracovávat průměrně **2008** vzorků za



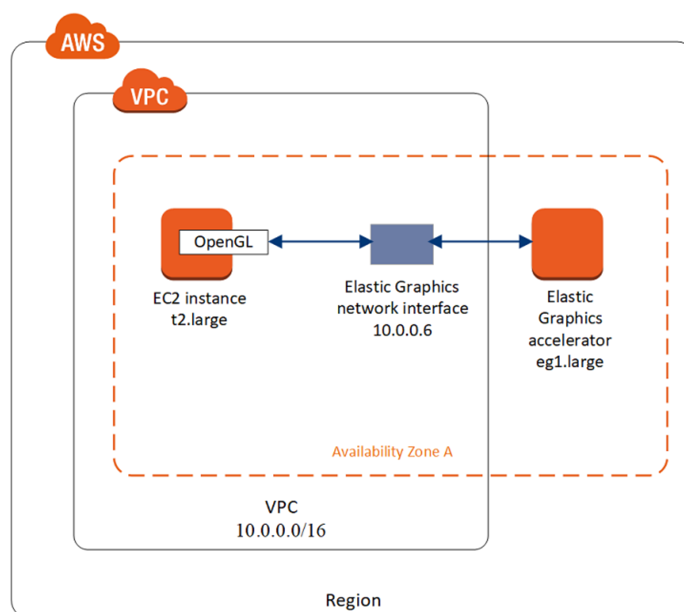
Obrázek 5.1: Náhodný výběr obrázků z 10 tříd datasetu `CIFAR-10`

vteřinu, přičemž karta byla v době výpočtu vytížena průměrně na **57** procent. Na virtuálním stroji zvládla grafická karta zpracovávat průměrně **1780** vzorků za vteřinu, což je zhruba o deset procent méně oproti nevirtualizované grafické kartě. Nicméně grafická karta na virtuálním stroji byla v době výpočtu vytížena průměrně na **48** procent, což je také rozdíl zhruba deseti procent. Pokleslý výkon si můžeme vysvětlit režií spojenou s přenášením dat ze zařízení do grafické karty. Náš virtuální stroj používá jako úložiště svých dat a operačního systému backend Ceph, což je síťové úložiště, které mohlo způsobit zpomalení přenosu dat do grafické karty.

5.3 Existující řešení

Korporátní společnosti, jako jsou například Google nebo Amazon, již nabízí své vlastní GPU cloudy, ani v jednom případě se však nejedná o open-source řešení, které bychom si mohli vyzkoušet ve vlastním prostředí. Od těchto korporací pouze dostaneme virtuální stroj s jednou či více grafickými kartami.

Google nabízí uživatelům virtuální servery s možností připojení jedné až osmi grafických karet. Jedná se o grafické karty NVIDIA typu Tesla K80, P100, P4 a V100. Spotřebitelům pak Google účtuje využitý čas po hodinách. Google, podobně jako náš GPU cloud, virtualizuje grafické karty pro uživatele pomocí *PCI passthrough*, cílí tedy na uživatele požadující plný výkon grafických karet [35]. Virtuální servery s grafickými kartami od Google využívá například mo-



Obrázek 5.2: Schéma komunikace VM a grafického akcelerátoru v cloudu Amazon

bilní aplikace Shazam, která slouží k rozeznávání názvů skladeb, které uživatel zaslechne například v rádiu [36].

Amazon svým uživatelům poskytuje grafické karty pomocí tzv. *Elastic Network Interface*, což je síťové rozhraní určené pro komunikaci mezi grafickým akcelerátorem a virtuálním strojem v Amazon cloudu [37]. Schéma komunikace je znázorněno na Obrázku 5.2 (zdroj: [37]).

5.4 Možné rozšíření práce

V této práci byl úspěšně implementován GPU cloud na platformě Openstack. Ke GPU cloudu byla navrhnuta a implementována komponenta Openstack Flare, která umožňuje vytvářet úlohy vyžadující grafickou kartu a poté těmto úlohám alokuje jednu či více volných grafických karet po omezenou dobu. V případě nedostupnosti grafické karty jsou úlohy zařazeny do fronty, kde čekají na uvolnění GPU prostředku.

Jedním z možných rozšíření je implementace dalších faktorů určujících, zda úloha může uvolnit alokovanou grafickou kartu, například zjišťováním, zda byla činnost obsazující grafickou kartu úspěšně dokončena.

Z časových důvodů nebyl implementován plugin do grafického uživatelského rozhraní Horizon, který by komunikoval s komponentou Openstack Flare. V této práci jako klient slouží pouze CLI klient. Dalším rozšířením by tedy mohlo být zakomponování Flare klienta do Horizonu.

V současnosti umí Openstack Flare pracovat pouze s jedním typem grafické karty v rámci celého cloudu. Možným rozšířením je implementace možnosti vybírat si mezi různými typy grafických karet.

Nakonec by bylo možné rozšířit funkcionalitu Openstack Flare i pro kontejnery v rámci projektu Openstack Magnum, který se stará o orchestrační nástroje kontejnerů.

Závěr

Cílem této práce bylo seznámit se s open-source cloudovou platformou Openstack a prozkoumat možnosti virtualizace grafických karet na této platformě. Byly představeny tři možnosti virtualizace grafických karet, přičemž byla vybrána ta, která by neměla degradovat výkon grafické karty v případě její virtualizace.

V této práci byl implementován GPU cloud na platformě Openstack, který umožňuje předat grafickou kartu virtuálnímu stroji. Grafickou kartu má v danou chvíli virtuání stroj plně pod kontrolou, uživatel tedy nepozná rozdíl oproti kartě nevirtualizované.

Do cloudu Openstack byla navržena a implementována kompomenta Openstack Flare, která umožňuje dynamicky plánovat běh virtuálních strojů s přidělenými GPU prostředky. Dané virtuální stroje mají grafické karty přiděleny jen po omezenou dobu, po uplynutí této doby dostane grafickou kartu k dispozici další virtuální stroj v pořadí.

Byla představena možnost vytvoření vlastního obrazu do Openstacku. Díky této možnosti je například možné připravit virtuální stroj s nainstalovanými ovladači grafické karty. Do vlastního obrazu ale můžeme nahrát jakýkoliv skript. Můžeme si tak například připravit obraz, který bude trénovat neuronovou síť, či pracovat s velkými daty.

Nakonec byla srovnána výkonnost virtualizované grafické karty oproti kartě nevirtualizované a bylo dokázáno, že ve většině případů nepozná uživatel rozdíl. Byla také uvedena ukázka využití GPU cloudu v aplikaci trénování neuronové sítě.

Literatura

[1] KHEDHER, Omar: *Mastering OpenStack - Second Edition*. Packt Publishing, 2017, ISBN 978-1-78646-398-2.

[2] BOUŠKA, Petr: Adresářové služby a LDAP [online]. 2007, [cit. 2019-01-04]. Dostupné z: https://www.samuraj-cz.com/clanek/adresarove-sluzby-a-ldap/

[3] BOBČÍK, Boleslav: PAM - správa autentizačních mechanismů [online]. 2000, [cit. 2019-01-04]. Dostupné z: https://www.root.cz/clanky/pam-sprava-autentizacnich-mechanismu/

[4] Openstack Foundation: Securing OpenStack networking services [online]. 2018, [cit. 2019-01-04]. Dostupné z: https://docs.openstack.org/security-guide/networking/services-security-best-practices.html

[5] Openstack Foundation: Create OpenStack client environment scripts [online]. 2018, [cit. 2019-01-04]. Dostupné z: https://docs.openstack.org/mitaka/install-guide-obs/keystone-openrc.html

[6] Docker, Inc.: What is a Container [online]. 2018, [cit. 2019-01-04]. Dostupné z: https://www.docker.com/resources/what-container

[7] Docker, Inc.: Docker overview [online]. 2018, [cit. 2019-01-04]. Dostupné z: https://docs.docker.com/engine/docker-overview

[8] What is Kubernetes? [online]. 2018, [cit. 2019-01-04]. Dostupné z: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes

[9] Kubernetes Architecture [online]. 2018, [cit. 2019-01-04]. Dostupné z: https://kubernetes.io/docs/concepts/architecture

[10] Pod Overview [online]. 2018, [cit. 2019-01-04]. Dostupné z: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview

[11] Magnum [online]. 2018, [cit. 2019-01-04]. Dostupné z: https://wiki.openstack.org/wiki/Magnum

[12] byte-unixbench [online]. 2016, [cit. 2019-01-05]. Dostupné z: https://github.com/kdlucas/byte-unixbench

[13] Intel Corporation: Bringing New Use Cases and Workloads to the Cloud with Intel® Graphics Virtualization Technology (Intel® GVT-g) [online]. 2016, [cit. 2018-12-27]. Dostupné z: https://01.org/sites/default/files/downloads/igvt-g/gvtflyer.pdf

[14] JAIN, Sunil: Intel® Graphics Virtualization Update [online]. 2014, [cit. 2018-12-27]. Dostupné z: https://software.intel.com/en-us/blogs/2014/05/02/intel-graphics-virtualization-update

[15] Puppet: How Puppet works [online]. 2018, [cit. 2018-12-31]. Dostupné z: https://puppet.com/products/how-puppet-works

[16] Codership Oy.: GALERA CLUSTER FOR MYSQL AND MARIADB - THE TRUE MULTI-MASTER [online]. 2017, [cit. 2018-12-30]. Dostupné z: http://galeracluster.com/products/

[17] Red Hat, Inc.: BLOCK STORAGE [online]. 2017, [cit. 2018-12-30]. Dostupné z: https://ceph.com/ceph-storage/block-storage/

[18] HAProxy Technologies, LLC: HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer [online]. 2018, [cit. 2018-12-30]. Dostupné z: https://www.haproxy.org/

[19] Openstack Foundation: Set up session storage for the Dashboard [online]. 2018, [cit. 2018-12-30]. Dostupné z: https://docs.openstack.org/horizon/latest/admin/sessions.html

[20] Pivotal Software, Inc.: Highly Available (Mirrored) Queues [online]. 2018, [cit. 2018-12-30]. Dostupné z: https://www.rabbitmq.com/ha.html

[21] SMITH, Stephen: Understanding Keystone Endpoints [online]. 2014, [cit. 2018-12-31]. Dostupné z: https://linuxacademy.com/blog/linux/understanding-keystone-endpoints/

[22] Archlinux: PCI passthrough via OVMF [online]. 2018, [cit. 2018-12-31]. Dostupné z: https://wiki.archlinux.org/index.php/PCI_passthrough_via_OVMF

[23] IOMMU [online]. poslední aktualizace 11. srpna 2018 14:23, [cit. 2019-01-02]. Dostupné z: https://cs.wikipedia.org/wiki/IOMMU

[24] Openstack Foundation: Attaching physical PCI devices to guests [online]. 2018, [cit. 2019-01-02]. Dostupné z: https://docs.openstack.org/nova/pike/admin/pci-passthrough.html

[25] SQLAlchemy - The Database Toolkit for Python [online]. 2018, [cit. 2019-01-03]. Dostupné z: https://www.sqlalchemy.org/

[26] LACOUR, Jonathan: Controllers and Routing [online]. 2010, [cit. 2019-01-03]. Dostupné z: https://pecan.readthedocs.io/en/latest/routing.html

[27] Openstack Foundation: Policies [online]. 2018, [cit. 2019-01-03]. Dostupné z: https://docs.openstack.org/security-guide/identity/policies.html

[28] Openstack Foundation: Timeutils [online]. 2018, [cit. 2019-01-03]. Dostupné z: https://docs.openstack.org/oslo.utils/latest/reference/timeutils.html

[29] HashiCorp: Introduction to Packer [online]. 2018, [cit. 2019-01-04]. Dostupné z: https://www.packer.io/intro/index.html

[30] NVIDIA Corporation: Cuda Samples [online]. 2018, [cit. 2018-12-28]. Dostupné z: https://docs.nvidia.com/cuda/cuda-samples/index.html

[31] NVIDIA Corporation: CUDA C Programming Guide [online]. 2018, [cit. 2018-12-28]. Dostupné z: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[32] ROOT.cz: CUDA: compute capability, profiler a page-locked paměť [online]. 2009, [cit. 2018-12-28]. Dostupné z: https://www.root.cz/clanky/cuda-compute-capability-profiler-a-page-locked-pamet/

[33] CIFAR-10 [online]. 2018, [cit. 2019-01-07]. Dostupné z: https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10_estimator

[34] The CIFAR-10 dataset [online]. 2018, [cit. 2019-01-07]. Dostupné z: https://www.cs.toronto.edu/~kriz/cifar.html

[35] GRAPHICS PROCESSING UNIT (GPU) [online]. 2019, [cit. 2019-01-07]. Dostupné z: https://cloud.google.com/gpu/

[36] RONCERO, Jesús: Moving GPUs to Google Cloud [online]. 2017, [cit. 2019-01-07]. Dostupné z: https://blog.shazam.com/moving-gpus-to-google-cloud-36edb4983ce5

[37] Amazon Elastic Graphics [online]. 2019, [cit. 2019-01-07]. Dostupné z: https://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/elastic-graphics.html

Příloha A

Seznam použitých zkratek

AI Artificial Inteligence

API Application Programming Interface

AMQP Advanced Message Queuing Protocol

CLI Command Line Interface

CPU Central Processing Unit

IOMMU Input-output Memory Management Unit

IP Internet Protocol

JSON JavaScript Object Notation

KVM Kernel Virtual Machine

LDAP Lightweight Directory Access Protocol

MAC Media Access Control

NFS Network File System

PAM Pluggable Authentication Module

PCI Peripheral Component Interconnect

QEMU Quick EMUlator

RAM Random Access Memory

RBD Rados Block Device

RC Run Commands

TCP Transmission Control Protocol

UDP User Datagram Protocol

VM Virtual Machine

YAML Yet Another Markup Language

Příloha B

Obsah přiloženého CD

```
readme.txt ............................ stručný popis obsahu CD
src
  impl ................................ zdrojové kódy implementace
  thesis .............................. zdrojová forma práce ve formátu LaTeX
text .................................. text práce
  DP_Cila_Michal_2019.pdf ............. text práce ve formátu PDF
  DP_Cila_Michal_2019_zadani.pdf ...... zadání práce ve formátu PDF
```