**PG-Strom让PostgreSQL查询飞起来**

**-----基于OpenShift-3.10的GPU加速**

**出自：**<https://blog.openshift.com/gpu-accelerated-sql-queries-with-postgresql-pg-strom-in-openshift-3-10/>

**译者：PostgreSQL开源社区朱君鹏,Qinghui.Guo**

## 介绍

在OpenShift 3.9 GPU博客中，我们利用OpenShift上的机器学习框架进行图像识别。在OpenShift 3.10博客中的如何使用带有DevicePlugin的GPU中，我们安装并配置了支持GPU的OpenShift集群。在本部分中，我们将在集群上创建更复杂的工作负载-使用GPU加速数据库查询。

任何机器学习算法的关键部分之一是数据（通常称为数据湖/仓库，存储为结构化，半结构化或非结构化数据）。

  机器学习管道的主要部分是准备，清理和探索这些数据。特别是删除NAs（缺失值），转换，规范化，子集化，排序和大量绘图。

  本博客将重点关注数据准备阶段，展示如何使用GPU加速PostgreSQL（数据仓库）中的查询。接着使用R绘制数据。

## 环境概述

* Red Hat Enterprise Linux 7.5，CentOS PostgreSQL 10镜像
* OpenShift Container Platform 3.10 Cluster running on AWS
* Container Runtime: crio-1.10.5
* 容器工具：podman-0.6.1，buildah-1.1，skopeo-0.1.30
* 主节点：m4.xlarge
* Infra节点：m4.xlarge
* 计算节点：p3.2xlarge（一个NVIDIA Tesla V100 GPU，8vCPU和61GB RAM）

本文中使用的yaml和配置文件在<https://github.com/redhat-performance/openshift-psap/tree/master/blog/gpu/pg-strom> 中可以找到。

我们创建了一个git仓库，其中包含要在整个博客中使用的工具。从现在开始，我们将它们称为pg-strom / <file>。

# git clone https://github.com/redhat-performance/openshift-psap

我们将使用centos / postgresql-10-centos7作为基础镜像，并使用PG-Strom启用GPU加速查询。 PG-Strom是PostgreSQL的扩展模块，可以加速SQL工作负载进行数据分析或批处理。

## 使用buildah构建自定义图像

新镜像的构建将使用buildah完成，这有助于构建OCI容器镜像，并且是“docker build”的替代方案。 Skopeo是一种便于检查，拉取容器镜像并将其推送到注册表的工具。现在安装buildah和skopeo。

# yum -y install buildah skopeo

Buildah使用简单的bash脚本进行构建，示例演示可在此处访问：Buildah演示。可以在pg-strom / buildah-pg.sh中找到此博客中使用的buildah脚本。该脚本安装容器，安装所需的软件并提交容器供以后使用。只需调用该脚本即可生成支持GPU的PostgreSQL容器。

# <dir>/pg-strom/buildah-pg.sh

让我们检查一下我们是否在本地提交了完整的镜像：

# podman images | grep pgstrom

localhost/pgstrom latest 7ca1854e4176 5 days ago 517MB

可以选择将容器推送到注册表。 Skopeo可以在不从注册表中提取镜像的情况下检查镜像：

# skopeo inspect  <registry>/<repo>/pgstrom

{

"Name": "<registry>/<repo>/pgstrom",

"Tag": "latest",

"Digest": "sha256:b71e9bca91e23a21e7579c01d38522b76dc05f83c59",

"RepoTags": [

"latest"

],

"Created": "2018-07-13T10:26:49.604378011Z",

...

## 准备PostgreSQL服务器

OpenShift 3.10的另一个新功能是支持HugePages。 HugePages是一种常用于数据库的内存访问性能优化技术（如PostgreSQL，C和Java应用程序）。我们将使用HugePages作为数据库内存。让我们安装一些工具以便于管理并创建所需的安装。

# yum -y install libhugetlbfs-utils

# hugeadm --create-global-mounts

让我们在节点上分配相当数量的大页面，并检查节点是否接受了更改。

为此，我们将使用tuned，它是用于监视和自适应调整系统设备的守护进程。我们可以创建一个配置文件，该配置文件是最简单的配置文件，包含几个部分来调整系统控件和参数。除此之外，还可以提供bash脚本来为系统实现更复杂的调整。

以下配置文件pg-strom / pgstrom.conf可以放在/ etc / tuned / pgstrom中，以便可以识别为要调整的配置文件。 Tuned具有配置文件继承的概念。在这里，我们将“include = openshift-node”设置为我们的自定义pgstrom（子）配置文件的父配置文件。

# tuned

[main]

summary=Configuration for pgstrom

include=openshift-node

[vm]

transparent\_hugepages=never

[sysctl]

vm.nr\_hugepages=16384

现在，根据openshift-node配置文件加载新的调优配置文件。默认情况下，每个OpenShift节点（计算，控制平面......）都有自己的调整配置文件，具体取决于它的职责。

# tuned-adm profile pgstrom

检查节点是否有HugePages。

# oc describe node | grep Capacity -A13

Capacity:

cpu: 8

hugepages-1Gi: 0

hugepages-2Mi: 32Gi

memory: 62710868Ki

nvidia.com/gpu: 1

pods: 250

Allocatable:

cpu: 8

hugepages-1Gi: 0

hugepages-2Mi: 32Gi

memory: 29054036Ki

nvidia.com/gpu: 1

pods: 250

## 持久化存储

我们希望数据被持久保存（而不是存储在ephemerally，在容器中，这将不会持久）。为此，我们将在OpenShift 3.10中使用一个名为Local Volumes的新功能。这里的关键部分是新的本地存储配置器，它监视主机上的特定路径，并为此路径中找到的每个安装点创建PV（持久卷）。

这些PV可以分组为命名组PV，如SSD或HDD，并且可以在命名空间中声明。然后，该声明可以被pod用作卷。

如何设置具有本地卷的PV可在此处阅读：配置本地卷以及Pod如何使用PersistentVolumeClaim消耗此PV可在此处阅读：使用持久卷。

对于这个博客，我创建了一个HDD PV组，其中包含一个保存PostgreSQL数据的磁盘。

# ls /mnt/local-storage/hdd/

disk1

OpenShift将为disk1创建一个PV，我们将声明这个PV并将其安装到pod中。查看pg-strom / pgstrom.yml以了解如何安装声明。

设置数据存储后，我们可以使用以下pg-strom / pgstrom.yml部署pgstrom pod。启动pod并检查日志，容器将初始化PostgreSQL数据存储（initdb）。

# oc create -f pgstrom.yml

# oc logs pgstrom

---------------------------- snip ---------------------------------------

Starting server...

UTC [1] LOG: listening on IPv4 address "0.0.0.0", port 5432

UTC [1] LOG: listening on IPv6 address "::", port 5432

UTC [1] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"

UTC [1] LOG: listening on Unix socket "/tmp/.s.PGSQL.5432"

UTC [1] LOG: redirecting log output to logging collector process

UTC [1] HINT: Future log output will appear in directory "log".

要启用扩展，我们必须更改postgresql.conf（存储在安装到pod中的PV中）并添加一些用于GPU加速的调整。在pod中编辑/var/lib/pgsql/data/userdata/postgresql.conf并设置：

# oc rsh pgstrom /bin/bash

$ vi /var/lib/pgsql/data/userdata/postgresql.conf

------------------------- snip -----------------------------------------

## postgresql.conf

huge\_pages = on

# Initial buffers and mem too small, increase it to work in mem

# and not in storage

shared\_buffers = 30GB

work\_mem = 30GB

# PG-Strom internally uses several background workers,

# Default of 8 is too small, increase it

max\_worker\_processes = 100

max\_parallel\_workers = 100

# PG-Strom module must be loaded on startup

shared\_preload\_libraries = '/usr/pgsql-10/lib/pg\_strom.so,pg\_prewarm'

------------------------- snip --------------------------------

编辑配置文件后，重新启动pod以启用新功能。

# oc replace --force -f pgstrom.yml

在pod中，我们可以检查服务器是否在监听

# oc exec pgstrom pg\_isready

/var/run/postgresql:5432 - accepting connections

buildah脚本还安装了postgresql-odbc，用于从我们的工作站连接到数据库服务器。但首先让我们创建一个测试数据库，我们将在其上进行一些初始测试查询。

# oc exec -it pgstrom /bin/bash

$ cd /var/lib/pgsql/pg-strom/test

$ make init\_regression\_testdb

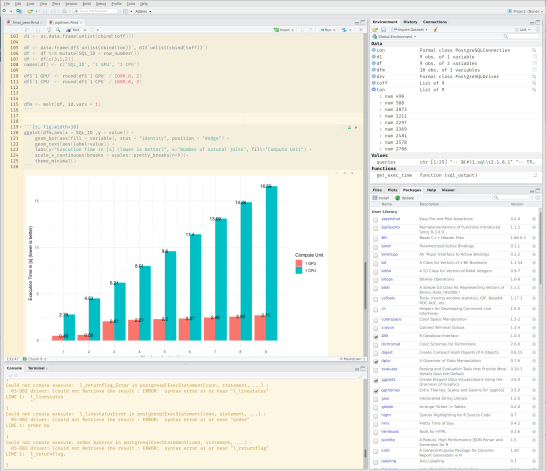
# Grab some coffee, this takes some time

## R连接数据库

对于下一部分，我们将使用R和RStudio创建一个类似于Jupyter的笔记本来访问数据库并从我们的查询中绘制一些数字。

 根据您在工作站上的版本安装R，RStudio和R软件包RPostgreSQL。

 我添加了一个简单的笔记本pg-strom / pgstrom.Rmd作为参考。这是RStudio中运行的笔记本运行针对PostgreSQL服务器的SQL查询。



第一步是通过db驱动程序包连接到数据库。

require(RPostgreSQL)

drv <- dbDriver("PostgreSQL")

con <- dbConnect(drv,

user="postgres",

password="postgres",

host=".com",

# This is the database we created in the step before

dbname="contrib\_regression\_pg\_strom")

现在我们可以对数据库发出SQL查询。第一步是启用PG-Strom扩展。

dbGetQuery(con, "SET pg\_strom.enabled=on")

我们现在可以对数据库运行加速查询，让我们从两个表上的一个简单的NATURAL JOIN开始，让数据库EXPLAIN和ANALYZE在这个特定的查询中做了什么。

dbGetQuery(con, "EXPLAIN ANALYZE SELECT cat, count(\*), avg(ax) FROM t0 NATURAL JOIN t1 GROUP BY CAT")

                                                           QUERY PLAN

------------------------------------------------------------------------------------------------------------------------------------

GroupAggregate  (cost=18097.45..18130.73 rows=1024 width=24) (actual time=299.065..300.648 rows=1025 loops=1)

  Group Key: t0.cat

  -> Sort  (cost=18097.45..18100.01 rows=1024 width=48) (actual time=299.053..299.480 rows=1025 loops=1)

        Sort Key: t0.cat

        Sort Method: quicksort  Memory: 193kB

        -> Custom Scan (GpuPreAgg)  (cost=18025.77..18046.25 rows=1024 width=48) (actual time=295.002..295.546 rows=1025 loops=1)

              Reduction: Local

              Combined GpuJoin: enabled

              -> Custom Scan (GpuJoin) on t0  (cost=19089.39..26894.61 rows=995000 width=16) (never executed)

                    Outer Scan: t0 (cost=0.00..20310.00 rows=1000000 width=12) (actual time=62.944..106.492 rows=1000000 loops=1)

                    Depth 1: GpuHashJoin (plan nrows: 1000000...995000, actual nrows: 1000000...994991)

                             HashKeys: t0.aid

                             JoinQuals: (t0.aid = t1.aid)

                             KDS-Hash (size plan: 11.54MB, exec: 7125.12KB)

                    -> Seq Scan on t1 (cost=0.00..2031.00 rows=100000 width=12) (actual time=0.012..49.225 rows=100000 loops=1)

Planning time: 0.424 ms

Execution time: 476.807 ms

PostgreSQL构建计划节点的树结构，表示所采取的不同操作。我们可以看到GPU（GpuPreAgg，GpuJon，GpuHashJoin）正在加速执行树的各个部分，因此我们得到了计划和执行时间。

 让我们关闭扩展并重新运行SQL语句。我们的笔记本再次执行以下内容。

dbGetQuery(con, "SET pg\_strom.enabled=off")

dbGetQuery(con, "EXPLAIN ANALYZE SELECT cat, count(\*), avg(ax) FROM t0 NATURAL JOIN t1 GROUP BY CAT")

QUERY PLAN

----------------------------------------------------------------------------------------------------------------------------------------------------

Finalize GroupAggregate  (cost=27455.10..27488.38 rows=1024 width=24) (actual time=2819.862..2823.624 rows=1025 loops=1)

  Group Key: t0.cat

  -> Sort  (cost=27455.10..27460.22 rows=2048 width=48) (actual time=2819.847..2821.153 rows=3075 loops=1)

        Sort Key: t0.cat

        Sort Method: quicksort  Memory: 529kB

        -> Gather  (cost=27127.42..27342.46 rows=2048 width=48) (actual time=2806.990..2809.907 rows=3075 loops=1)

              Workers Planned: 2

              Workers Launched: 2

              -> Partial HashAggregate  (cost=26127.42..26137.66 rows=1024 width=48) (actual time=2804.084..2804.723 rows=1025 loops=3)

                    Group Key: t0.cat

                    -> Hash Join (cost=3281.00..23018.05 rows=414583 width=16) (actual time=306.307..2281.754 rows=331664 loops=3)

                          Hash Cond: (t0.aid = t1.aid)

                          -> Parallel Seq Scan on t0  (cost=0.00..14476.67 rows=416667 width=12) (actual time=0.027..622.406 rows=333333 loops=3)

                          -> Hash (cost=2031.00..2031.00 rows=100000 width=12) (actual time=305.785.  -> Seq Scan on t1 (cost=0.00..2031.00 rows=100000 width=12) (actual time=0.012..182.335 rows=100000 loops=3)

Planning time: 0.201 ms

Execution time: 2824.216 ms

我们可以看到没有GPU方法出现，查询在CPU上运行。执行时间从~477 ms增加到2824 ms -几乎是6倍的加速。

 让我们以此为起点，为10个表（t0-t9）创建自然连接的基准，在CPU和GPU上一次添加一个表。我已经实现了一个小函数来从输出中提取执行时间并将结果保存在R数据框中，以便于处理和绘图（有关详细信息，请参阅笔记本）。

 现在针对CPU和GPU运行查询。

dbGetQuery(con, "SET pg\_strom.enabled=on")

ton <- list()

ton <- c(ton, get\_exec\_time(dbGetQuery(con, "EXPLAIN ANALYZE SELECT cat, count(\*), avg(ax) FROM t0 NATURAL JOIN t1 GROUP BY CAT")))

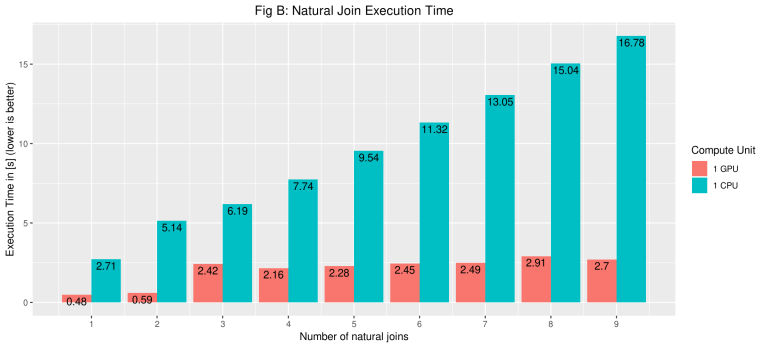
ton <- c(ton, get\_exec\_time(dbGetQuery(con, "EXPLAIN ANALYZE SELECT cat, count(\*), avg(ax) FROM t0 NATURAL JOIN t1 NATURAL JOIN t2 GROUP BY CAT")))

ton <- c(ton, get\_exec\_time(dbGetQuery(con, "EXPLAIN ANALYZE SELECT cat, count(\*), avg(ax) FROM t0 NATURAL JOIN t1 NATURAL JOIN t2 NATURAL JOIN t3 GROUP BY CAT")))

ton <- c(ton, get\_exec\_time(dbGetQuery(con, "EXPLAIN ANALYZE SELECT cat, count(\*), avg(ax) FROM t0 NATURAL JOIN t1 NATURAL JOIN t2 NATURAL JOIN t3 NATURAL JOIN t4 GROUP BY CAT")))

...

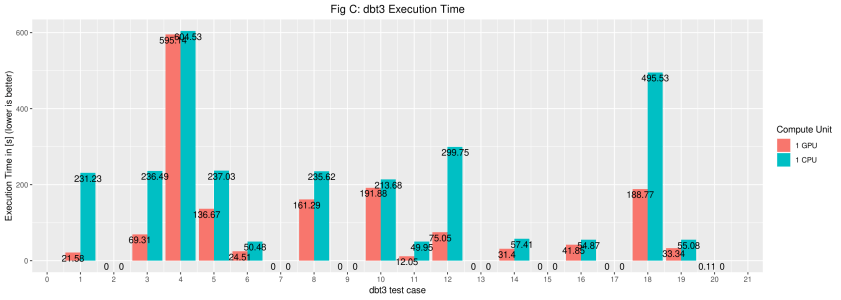
在提取执行时间之后，我们现在能够在R中绘制数据，我们在这里使用ggplot2，这是R中最常用的图形包。



GPU加速查询在~2.5秒内保持平稳，其中CPU查询因添加的每个新表而增加。我们可以进一步添加更多表来显示GPU执行时间何时会增加，但这超出了此博客的范围。

## PostgreSQL 上使用PG-Strom运行DBT-3基准测试

DBT-3基准测试是决策支持基准测试的开源实现，其中数据和工作负载基于特定的数据库规范。基准测试将在PG-Strom启用和禁用的情况下运行，类似于上面的测试。



大多数查询都会在没有任何SQL查询更改的情况下加速（某些查询未在CPU或GPU上运行，因此报告为0）。

## 结论

PG-Strom是一个在数据库SQL级别即时加速的优秀插件。查询（avg，cnt，sqrt，...）中的数字运算次数越多，GPU的利益就越大。

此外，还有一些努力正在加速用GPU加速R.在不久的将来，人们可能拥有从数据库到分析，绘图和统计计算的完整GPU加速管道。