Fermilab

DGaaS: GPU as a Service on Distributed Computing System

FERMILAB-TM-2889-STUDENT

DGaaS

GPU as a service on distributed computing system

FINAL REPORT

August-September ‘23

Sara Mazzucato

Marco Mambelli, Bruno Coimbra

Contents

ABSTRACT

DGaaS: GPU as a Service on Distributed Computing System

In the rapidly evolving landscape of scientific computing, Graphics Processing Units (GPUs) have become indispensable for their unparalleled ability to handle parallel tasks in complex calculations, simulations, and data analysis. Their utility is further magnified in machine learning and AI applications, where they significantly accelerate model training and predictive analytics. Within this context, the Triton Inference Server emerges as a pivotal open-source tool, specializing in AI inferencing and optimizing GPU utilization across various platforms and frameworks.

This paper presents an in-depth study on distributed High Throughput Computing (HTC), specifically focusing on the HTCondor framework and its resource provisioning tools, GlideinWMS and HEPCloud. These systems enable large-scale scientific experiments like CMS and DUNE to efficiently access and utilize vast computational resources. The paper explores the core architectural components of GlideinWMS, including jobs, user pools, and worker nodes, and discusses their integration with GPUs and the Triton server.

The primary aim of this research is to develop a solution that optimizes GPU utilization by leveraging Glideins and containers. This approach allows computational jobs, particularly those involving AI models, to use GPUs only when essential, thereby facilitating efficient sharing of limited GPU resources. To validate this architecture, the study conducted three key tests involving custom scripts, container-based servers, and Triton server deployments.

However, the study faces challenges, notably in locating the Triton server and ensuring secure remote access. To address these issues, future work will focus on developing a proxy mechanism and enhancing security protocols.

In conclusion, this study offers a comprehensive roadmap for effective and efficient GPU utilization in distributed High Throughput Computing. It aims to contribute significantly to the scientific community by solving pressing problems and implementing robust solutions in collaboration with the GlideinWMS and HEPCloud teams. The research sets the stage for a more efficient, scalable, and cost-effective paradigm in scientific computing.

INTRODUCTION

Computing resources as support to HEP experiments

Computing resources are vital for High Energy Physics (HEP) experiments, with grid computing emerging as a key solution. By connecting numerous geographically dispersed computers through networks, as seen in Fermilab's experiments like CMS and DUNE, grid computing enables collaborative task completion. This approach integrates global grid sites, including university and lab batch systems, into a unified infrastructure, fostering distributed high-throughput computing for HEP endeavors.

Worker nodes

A worker node, under the management of a resource manager, is distinguished by its resource allocation such as CPU, RAM, and Disk. Functioning as a logical resource abstraction, this node essentially serves as a computer tasked with specific jobs. The CMS Global Computing grid, exemplified with over 120 sites and 2 million CPU cores, showcases this concept. Adjacently, additional instances of worker nodes are illustrated on the left.

GlideinWMS

GlideinWMS serves as a scientific computing facilitator, designed to streamline workflows centered around GPU utilization and sharing. Built upon HTCondor, a workload management system acting as a batch system or local resource manager, GlideinWMS operates as a Glidein-driven workload management framework within the HTCondor framework. This system functions as a pilot-oriented resource provisioning tool for distributed High Throughput Computing. Comprising multiple services, GlideinWMS aims to simplify access to Grid resources by exposing pilots to potentially unreliable and diverse resources. Consequently, users can seamlessly submit standard HTCondor jobs while computational resources are seamlessly orchestrated in the background.

Components are visible in the bottom figure:

- Job: task, program that needs to be executed by the user running the experiment. E.g. data analysis' script working on clinical data
- User Pool: set of jobs that a user wants to be run.
- GlideinWMS Frontend & Factory: main services behind GlideinWMS.
- Worker nodes: machines part of the Grid where jobs will be actually executed.

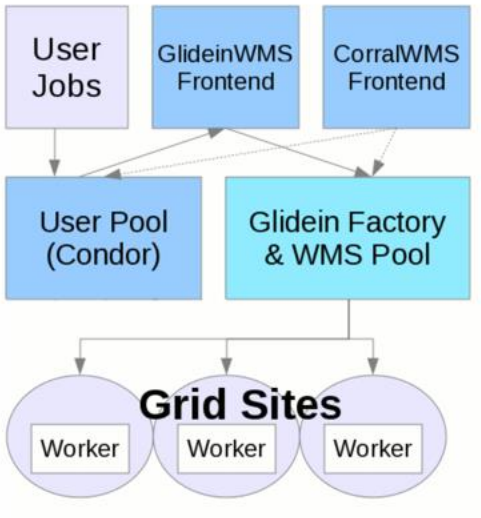

Figure 1- GlideinWMS structure

HTCondor

HTCondor functions as a workload management system, encompassing both batch system and local resource manager functionalities. As an open-source implementation, it offers fault tolerance, a sturdy feature set, and adaptability. It finds application in local high-performance computing centers such as UW Madison. For instance, when operating within a consistently active local cluster, specific software or file systems unique to the environment might be relied upon. Notably, it accommodates specialized job scenarios involving multiple nodes, like MPI or HTC resources. HTCondor provides essential components including a job queueing mechanism, scheduling policy, resource monitoring, and resource management to effectively manage and optimize job execution within computing clusters.

GPU's Role in Scientific Computing

In the realm of scientific computing, GPUs play a pivotal role due to their remarkable attributes. Their prowess in handling extensive parallel tasks is especially noteworthy. This capability proves invaluable for tackling intricate calculations, spanning simulations, data analysis, and optimization tasks. Notably, GPUs contribute to a significant boost in processing speed and efficiency, resulting in notable reductions in execution time. Their impact extends to machine learning as well. When training deep neural networks, GPUs' parallel processing capabilities offer accelerated model refinement. Moreover, they expedite data preprocessing and feature extraction in data analysis tasks. In the realm of AI applications, GPUs further enhance predictive modeling and decision-making processes through their computational potency.

Triton server

The Triton Inference Server stands as an open source software tailored for seamless AI inferencing. Within the Central Distributed System, the Triton server fulfills the role of an open source inference serving tool, with a focus on GPU utilization. This server facilitates GPU management by enabling its access from external programs, extending even to remote usage. Triton's functionality allows for the sharing of GPUs, allocating them to processes that demand their resources, whether local, remote, cloud-based, or across clusters. It empowers teams to

deploy AI models from various deep learning and machine learning frameworks, encompassing TensorRT, TensorFlow, PyTorch, ONNX, OpenVINO, Python, RAPIDS FIL, and more. Triton seamlessly supports inference across diverse platforms including NVIDIA GPUs, x86 and ARM CPUs, AWS Inferentia, and cloud environments, optimizing performance for a spectrum of query types such as real-time, batched, ensembles, and audio/video streaming.

Virtuam Machine and Containers

Two fundamental concepts are at the forefront: virtual machines (VMs) and containers. VMs act as self-contained software environments, mirroring physical computers. They facilitate the operation of multiple operating systems on a single hardware infrastructure. Conversely, containers are lightweight packages that enclose applications along with their dependencies, ensuring portability across various platforms.

Both VMs and containers significantly enhance operational efficiency by simplifying software deployment, optimizing resource utilization, and enabling seamless scalability in modern computing landscapes. This convergence is where Apptainer emerges. Apptainer serves as an orchestrator for container clusters, streamlining the scaling and management of applications. It facilitates the uniform distribution of applications across diverse environments, while maintaining simplicity in the process.

Given the inherent diversity of worker nodes, necessitating the utilization of both VMs and containers, Apptainer bridges the gap. VMs unify the disparate nodes by isolating services and encapsulating programs with their dependencies, ensuring seamless portability. While VMs are more widespread, containers, being computationally lighter, are often preferred for their efficiency in various applications.

Apptainer, in particular, stands out by providing robust support for running containers. A notable advantage is that these containers operate in an unprivileged state, enhancing security and maintaining system integrity. Through the synergy of VMs, containers, and the streamlined management offered by Apptainer, the complexities of modern computing environments are seamlessly addressed.

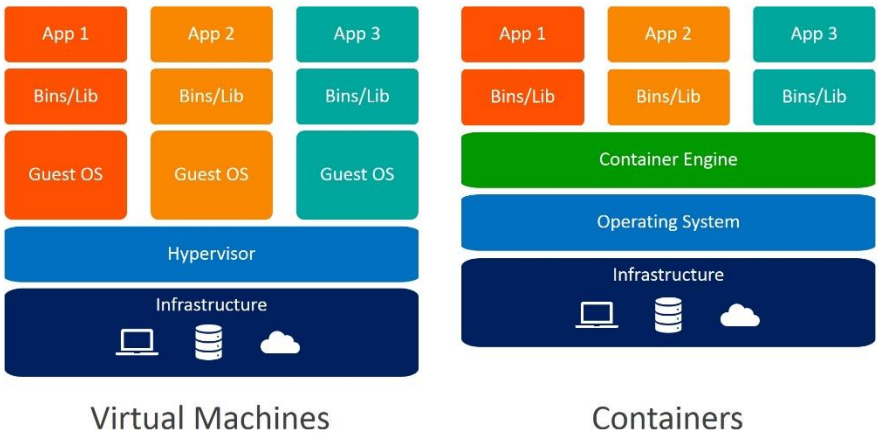

Figure 2-Difference between Virtual Machine (VM) and Containers

Glidein

A glidein refers to a pilot job initiated within a worker node to secure and configure the necessary resources for a job's execution. These glideins are solicited by the Frontend, activated through the Factory, and integrated into the virtual cluster. Essentially, a glidein functions as a pilot job, serving as the catalyst for commencing processes. It enables the initiation of simulations initiated by users, ensuring a smooth start to the computational tasks.

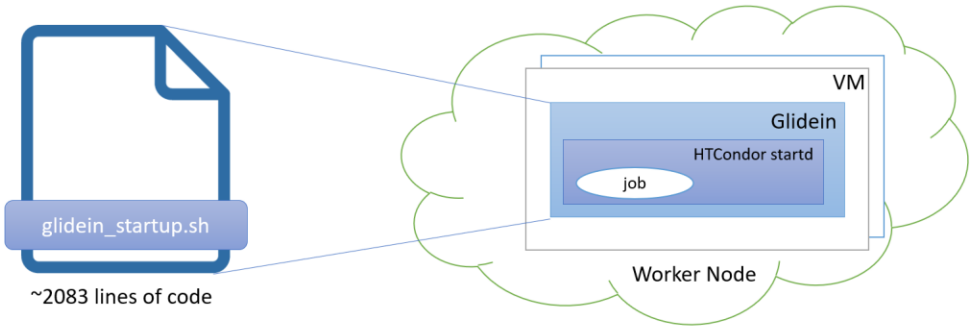

Figure 3-Glidein start up role inside the Worker Node

Life cycle phase execution

The life cycle of a glidein, encompassing the execution of custom scripts, unfolds through four distinct phases: startup, pre-job, post-job, and cleanup. The glidein's configuration component features a designated file section dedicated to specifying programs for both testing and node setup.

Our primary focus lies on the initial setup phase, where the Triton server is initiated. This is achieved using Apptainer, which employs a portable container for seamless execution.

For illustrative purposes, let's consider the scenario of a single-node glidein. During the startup phase, a series of scripts come into play. These scripts undertake tests to ensure the worker node's functionality, followed by the setup process that readies the node for job execution. It's worth noting that multiple jobs can be involved, with the glidein capable of executing them either in parallel or sequentially.

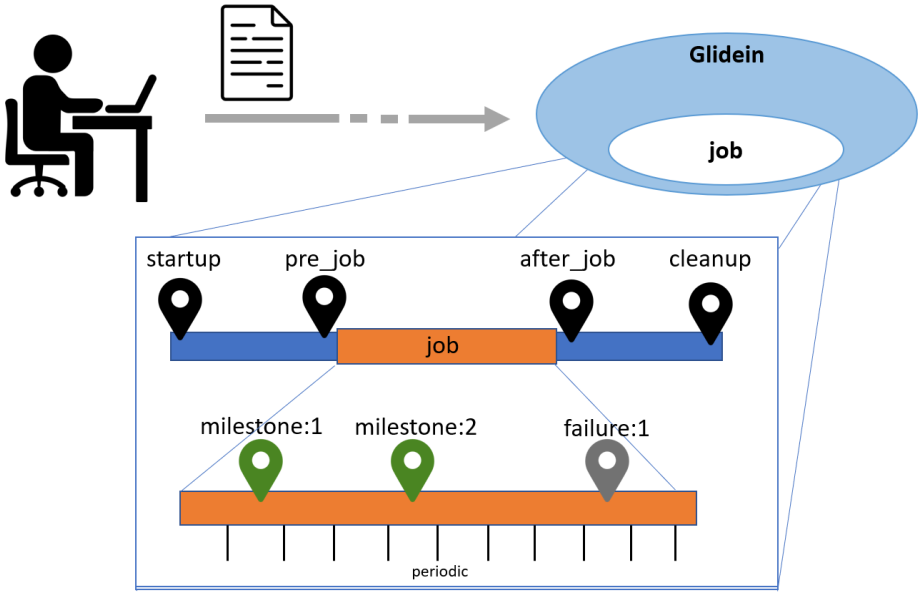

Figure 4-Workflow of Glidein during the submission of a user job

DELIVERABLES

The initial objective involves establishing a local system where a single node hosts both a glidein instance and multiple jobs executed by the same Triton server. Looking ahead, the aspiration is to expand this capability to remote clusters. This would empower jobs to leverage GPUs across different nodes, enabling glidein to access external jobs. This could potentially entail the creation of a proxy mechanism.

Notably, the envisioned setup envisions glidein and jobs interacting seamlessly. This harmonious interaction extends to utilizing glideins across diverse clusters, even those connected remotely.

However, this endeavor isn't without challenges. Two major difficulties include locating the Triton server and ensuring secure access from remote sources. These obstacles need to be navigated to achieve the envisioned efficient and collaborative computing setup.

METHODS

In this section are explained the first steps to set up a Virtual machine, a short introduction of Apptainer and then the different tests executed.

It is needed to install git, Docker, WSL, Alma9, as further explained in the following.

Creation of a VM

Figure 5-Symbol of Openstack

OpenStack is an open-source cloud computing platform that allows organizations to build and manage cloud infrastructure using a collection of integrated services. One of the key functionalities of OpenStack is its ability to provision and manage virtual machines (VMs) through its Compute service, commonly known as Nova.

Creating a VM in OpenStack:

1. Access Dashboard: Log in to the OpenStack Dashboard, also known as Horizon.
2. Navigate to Compute: Once logged in, go to the "Compute" section and then click on "Instances."
3. Launch Instance: Click on the "Launch Instance" button to initiate the VM creation process.
4. Configure Instance: In the launch instance dialog, you'll be prompted to configure various settings such as:
   - Name: A name for the VM.
   - Flavor: The type of instance (CPU, memory, storage).
   - Image: The operating system image to use.
   - Key Pair: SSH key pair for secure access.
   - Network: The network to which the VM will be connected.
   - Security Groups: Firewall rules for the VM.
   - Launch: After configuring the settings, click on the "Launch" button to create the VM.
5. Monitor Status: You can monitor the status of the VM from the "Instances" page. Once the status changes to "Active," the VM is up and running.
6. Access VM: You can then access the VM using SSH or through the console available in the dashboard.

Available VM address:

- root@fermicloud862.fnal.gov
- root@fermicloud762.fnal.gov (has 30+70=100GB)

Addresses of Frontend and Factory:

- fermicloud819.fnal.gov FRONTEND
- fermicloud798.fnal.gov FACTORY
- fermicloud762

Windows Subsystem for Linux

First of all it is needed to do the 'Windows Setup' of the three containers: factory, front end and compute engine (https://github.com/glideinWMS/glideinwms/wiki/Windows-Setup).

The following steps serve as a guide on setting up a development environment on Windows using WSL (Windows Subsystem for Linux), Docker, and VSCode.

- **Prerequisites**: Enable Virtual Machine Platform and WSL in Windows Features.
- **Installation**: Install a Red Hat-based Linux distro like Rocky, Alma, or CentOS from the Windows Store. Follow the guide on Microsoft's website for WSL installation. In our case the distro used is Alma9.
- **Basic Commands**: Use wsl to enter the default Linux distro, and `wsl -d <Distro>` for non-default ones. List all installed distros with `wsl -l`.
- **Setup**: Connect to the Fermilab network directly or via VPN. Install Kerberos for authentication.
- **VSCode and Extensions**: Install VSCode and various extensions for WSL, Docker, Python, and SSH management.
- **WSL Configuration**: Set the default Linux distro to Alma9/CentOS and create an `ssh.bat` file to handle SSH configurations between Windows and Linux.
- **Remote** Connection: Use VSCode's Remote Explorer to connect to remote VMs and containers. Make sure to authenticate Fermilab's network if needed.
- **Compute Engine and Factory Setup**: Instructions are provided for setting up compute engines and factories, checking their statuses, and running startup scripts.
- **Token Management**: Update tokens every few hours for proper functioning.
- **Dev Containers**: Install `podman-docker` to use development containers within VSCode. In this case the following version was used:

```
docker pull nvcr.io/nvidia/tritonserver:23.08-py3
```

Apptainer/Singularity

Reference: https://mambelli.github.io/hsf-training-apptainer/index.html

Apptainer (formerly Singularity), a free and open-source container platform designed for the scientific and High-Performance Computing (HPC) communities. It allows for the creation and running of isolated application environments, known as containers.

Prerequisites:

- Basic Unix Shell knowledge is required.
- Access to a Linux-based system with Apptainer/Singularity installed or the ability to enable user namespaces and access CVMFS.

Software Setup:

Apptainer and Singularity are often pre-installed on institutional computing resources. You can check their availability and versions with `apptainer --version` or `singularity --version`.

If not installed or if the version is outdated, you can potentially use an updated version via CVMFS if user namespaces are enabled.

A short guide on using Apptainer's Command Line Interface (CLI) for container management is reported below.

Key Features:

- Check versions with `--version` and get help with `--help`.
- Apptainer allows you to search, build, and run containers easily.
- Compatible with both its own and Docker images.

Commands:

- `apptainer --version` or `singularity --version` to check the version.
- apptainer --help to see available options.
- `apptainer search` to find containers.
- `apptainer pull` to download images.

Running Containers:

- Use `apptainer shell` to start an interactive shell inside the container.
- Use `apptainer exec` to execute a specific command inside the container.

Storage and Binding:

- Images are stored as .`sif` files.
- Directories can be bound between the host and the container using the `--` option.

Compatibility:

Apptainer is compatible with Docker images and can pull from Docker Hub.

Important Notes: Recommended versions are Apptainer >= 1.0 or Singularity >= 3.5.

In the following it is explained how to create and modify containers using Apptainer's build command, described as the "*Swiss army knife of container creation*."

Workflow:

- Develop and test containers in a build environment like a laptop.
- Deploy the container into a production environment like an institutional cluster.
- For reproducibility and transparency, it's recommended to build containers from a definition file.

Building Containers:

- Use `--sandbox` flag with `apptainer build` to create a writable directory for building and testing.
- Example: `apptainer build --sandbox myCentOS7 docker://centos:centos7`

Interactive Sessions:

Use `--writable` and `--fakeroot` options for writing files and installing new components.

Key Points:

- `build` is essential for container creation.
- Sandboxes are writable directories for interactive container building.
- Superuser permissions are needed for certain build tasks.
- Use interactive builds for testing and definition files for production or distribution.

To build and deploy containers using Apptainer definition files, which offer a streamlined way to create containers, it is possible to follow the example below.

- Create a definition file (e.g., `hello-world.def`) with the necessary script.
- Build the image with `apptainer build hello-world.sif hello-world.def`.
- Run the image with `./hello-world.sif.`

Deployment:

Containers can be executed immediately and anywhere once built.

Libraries like Sylabs Cloud Library and organizations like OSG facilitate image distribution.

Finally, we can use Apptainer for running background services, particularly useful for deploying web applications like Jupyter notebooks. Unlike commands like `run` and `shell` which run containers in the foreground, Apptainer offers the concept of "instances" to run containers in the background.

Basic Usage:

- To start an instance, use `apptainer instance start <image> <instance_name>.`
- To list running instances, use `apptainer instance list.`
- To interact with an instance, use `apptainer shell instance://<instance_name>`.
- To stop an instance, use `apptainer instance stop <instance_name>`.

Advanced Features:

- Instances can have bind paths for directory mounting using `--bind` option.
- You can deploy web services, demonstrated with a basic HTML server using Python's `http.server`.
- You can deploy a Jupyter notebook with a custom environment, including libraries like ROOT.

SSH Tunneling: For remote deployments, SSH tunneling can be used to access services.

Reproducibility: Apptainer allows you to package the environment needed for Jupyter notebooks, ensuring that your code will work consistently over time.

Test1: script execution

The aim of the first test is to create a custom script (*hello_script_factory.sh*) in the factory and call it from the frontend.

```
kinit saram@FNAL.GOV #saWed08022356#

klist
ssh root@fermicloud862.fnal.gov
```

```
# ON THE FACTORY
podman exec -it factory-workspace.fnal.gov /bin/bash #exec continue after
shutting down

ls /etc/gwms-factory/
nano hello_script_factory.sh
find / -name " hello_script_factory " 2>/dev/null #find path
vi /etc/gwms-factory/glideinWMS.xml  #to modify the xml file

#Enter this line at the end of the xml file
<file absfname="/hello_script.sh" after_entry="False" const="True"
executable="True"/>
#on the cmd
systemctl stop gwms-factory
systemctl reload gwms-factory  OR  gwms-factory reconfig
systemctl start gwms-factory

# ON FRONTEND
podman exec -it frontend-workspace.fnal.gov /bin/bash #exec continue after
shutting down
# renew il token scitoken-create.sh
su - testuser
condor_submit submitFile
```

Test 2: use a Glidein script of a container to launch a server

Typically, containers are initiated in an interactive mode using Podman, but the goal here is to run them in the background as a server.

To achieve this, a server can be launched from a container using either Apptainer or Docker, utilizing a web server Docker image.

In the Glidein configuration file, make sure to update the 'type' attribute to include 'exe' along with modifiers such as 'exe:s', where 's' represents Singularity (or Apptainer).

Finally, run the Docker application within the container to start the server.

Reference to deploy a model using Triton inference Server:

- https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/getting_started/quickstart.html
- https://github.com/triton-inference-server/server/blob/main/docs/getting_started/quickstart.md#run-on-cpu-only-system

```
# Step 1: Create the example model repository
git clone -b r23.08 https://github.com/triton-inference-server/server.git
cd server/docs/examples
./fetch_models.sh
```

```
# Step 2: Launch triton from the NGC Triton container
docker run --gpus=1 --rm --net=host -v ${PWD}/model_repository:/models
nvcr.io/nvidia/tritonserver:23.08-py3 tritonserver --model-repository=/models

# Step 3: Sending an Inference Request
# In a separate console, launch the image_client example from the NGC Triton SDK
container
docker run -it --rm --net=host nvcr.io/nvidia/tritonserver:23.08-py3-sdk
/workspace/install/bin/image_client -m densenet_onnx -c 3 -s INCEPTION
/workspace/images/mug.jpg

# Inference should return the following
Image '/workspace/images/mug.jpg':
    15.346230 (504) = COFFEE MUG
    13.224326 (968) = CUP
    10.422965 (505) = COFFEEPOT
```

Other triton tutorials to familiarize users with Triton's features and provide guides and examples to ease migration: https://github.com/triton-inference-server/tutorials/tree/main

Wilson Cluster – Institutional Cluster: SLURM job scheduler

SLURM (Simple Linux Utility For Resource Management) is an open-source resource manager and job scheduling system developed by SchedMD. It is widely used in high-performance computing environments, including most of the Top 500 supercomputers.

SLURM provides various commands for job control, monitoring, and resource allocation, such as scontrol, squeue, sbatch, salloc, srun, sinfo, and sacct.

Users can check their default SLURM account and associated accounts using the 'sacctmgr' command. If an account name is not specified during job submission, the default account will be used.

Reference: https://computing.fnal.gov/wilsoncluster/slurm-job-scheduler/

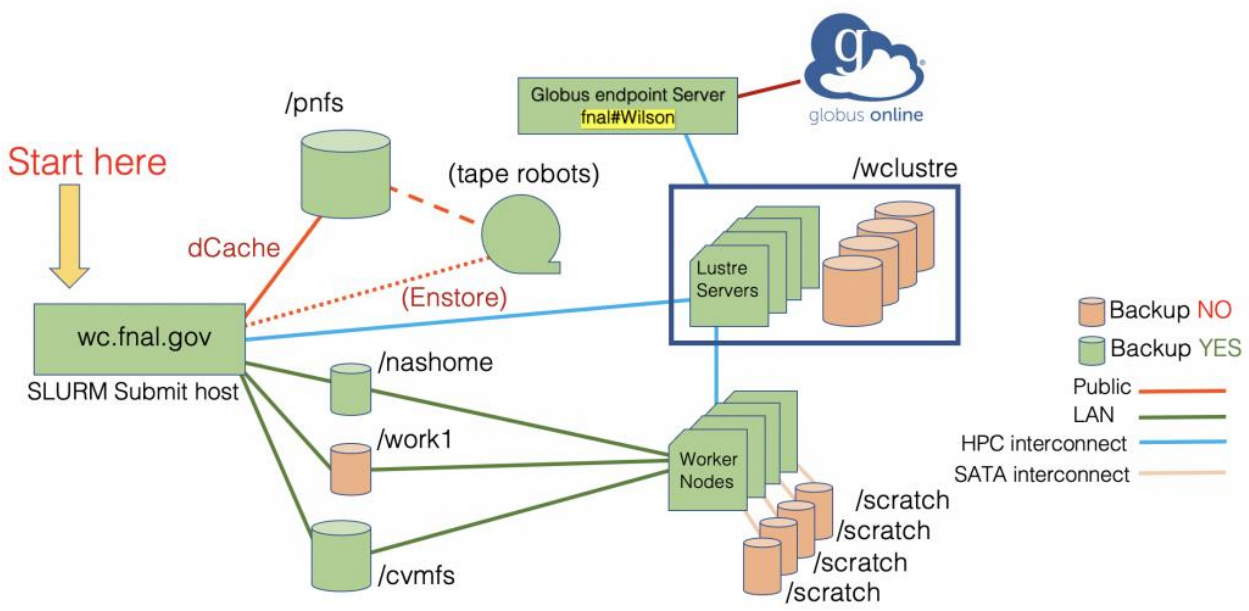

Figure 6-SLURM job scheduler

```
#To connect to the SLURM Submit host:
ssh wc.fnal.gov
#to check your “default” SLURM account
sacctmgr list user name=saram
      User   Def Acct       Admin
---------  ---------- ----------
   saram    HEPCloud       None
```

Slurm Resource types

SLURM Type	Resource Type	Description	GPU Type	Number of resources	Number of tasks per resource	SLURM Partition	Nodenames	Shared Resource **
–constraint				–nodes	–ntasks-per-node	–partition	—nodelist	–exclusive
intel2650	CPU	2.6 GHz Intel E5-2650v2 "*Ivy Bridge*" Eight Core	None	90	16	cpu_gce	wcwn[001-090]	No
intel2650	CPU	Same as above	None	10	16	cpu_gce_test	wcwn[091-100]	No
p100nvlink	GPU	2.4GHz Dual CPU Fourteen Core Intel	2x NVIDIA P100 with NVLINK	1	56	gpu_gce	wcgpu01	Yes
p100	GPU	1.7GHz Dual CPU Eight Core Intel	8x NVIDIA P100 with **NO** NVLINK	1	16	gpu_gce	wcgpu02	Yes
v100	GPU	2.5GHz Dual CPU Twenty Core Intel	2x NVIDIA V100 with **NO** NVLINK	4	40	gpu_gce	wcgpu[03-06]	Yes
a100	GPU	2.8GHz Dual CPU Thirty-Two Core EPYC 7543	4x NVIDIA A100 with **NO** NVLINK	1	64	gpu_gce	wcgpua01	Yes
v100nvlinkppc64	GPU	3.8GHz Dual CPU Sixteen Core IBM Power9	4x NVIDIA V100 with NVLINK	1	128	gpu_gce_ppc	wcibmpower01	Yes

Figure 7-SLURM Resource Types

It is better to use cpu_test to send a test job since it is faster, then we should use the SLURM partitions that have GPU (and are into the red rectangle).

We used SLURM to submit an interactive job. Remember to exif at the end. In this case srun is the command equivalent to condor submit).

Example:

```
[@wc]$ srun --pty --nodes=1 --ntasks-per-node=16 --partition cpu_gce bash
[user@wcwn001]$ env | grep NTASKS
SLURM_NTASKS_PER_NODE=16
SLURM_NTASKS=192
[user@wcwn001]$ exit
```

In my case:

```
docker images
REPOSITORY                    TAG                 IMAGE ID       CREATED        SIZE
nvcr.io/nvidia/tritonserver   23.08-py3           3e96065a3dcc   4 weeks ago    12.4GB
nvidia/cuda                   12.2.0-base-centos7 0789f3090339   2 months ago   516MB

df -h #to look how much disk is free
```

Inference test running in a Triton Server using python function:

```
# Install Triton Client in python
pip install 'tritonclient[all]'
```

Define in a .py file:

```
import tritonclient.http as httpclient
from tritonclient.utils import InferenceServerException
triton_client = httpclient.InferenceServerClient(
                url='localhost:8000')
def test_infer(model_name,
               input0_data,
               input1_data):
    inputs = []
    outputs = []
    inputs.append(httpclient.InferInput('INPUT_LAYER_1_NAME',
list(input0_data.shape), "INT64"))
    inputs.append(httpclient.InferInput('INPUT_LAYER_2_NAME',
list(input1_data.shape), "INT64"))
# Initialize the data
    inputs[0].set_data_from_numpy(input0_data)
    inputs[1].set_data_from_numpy(input1_data)
outputs.append(httpclient.InferRequestedOutput('model_output',
                                               binary_data=False))
    results = triton_client.infer(
        model_name,
        inputs,
        outputs=outputs
)
return results

input0_data = np.arange(start=0, stop=16, dtype=np.int64)
input0_data = np.expand_dims(input0_data, axis=0)
input1_data = np.full(shape=(1, 16), fill_value=-1, dtype=np.int64)

test_infer('simple_identity', input0_data, input1_data)
```

Create a /tmp folder to contain the model and run the script

```
mkdir /tmp/models
[saram@SlimProSara:~/models/simple_identity]$ cp -r ../simple_identity
/tmp/models/
[saram@SlimProSara:~/models/simple_identity]$ docker run --rm -p 8000:8000 -p
8001:8001 -p 8002:8002 -v /tmp/models:/models 3e96065a3dcc tritonserver --model-
repository=/models
[saram@SlimProSara:~/models/simple_identity] python3 test.py
```

Alternative way to use cmd and python to call function inline from file:

```
PYTHONPATH="/~/:$PYTHONPATH"  #update python path, insert the ‘‘directory list”
separated by :
/~ #is equivalent to $home

#Import the files/ functions
 import ‘name file.py’ or ‘directory’
from ‘module/folder’ import * #it will import all
```

If all the scripts are located in a single folder, it's advisable to create a package by adding an __init__.py file. This file can be used to specify which components are public or private, as well as to initialize variables.

Test 3: use Glidein script from a container to launch Triton

When Glidein is activated, it locates the image and launches Triton via Apptainer. After that, jobs can be submitted. You can also place the client image and run tests again.

1. **Create two images (server and client): client.sif and server.sif**

```
nano client.def
```

Inside paste the image configuration:

```
#start from img di Triton, 2 img in /cvmfs/local

tritonserver_23.08-py3-sdk.sif #client
tritonserver_23.08-py3.sif #serverls

BootStrap: docker
From: ubuntu:20.04

%runscript
  echo "Client"
```

Create the .sif file:

```
apptainer build client.sif client.def
```

Repeat for `server.sif`

2. **Save the images in the directory /opt and make it publicly visible:**

```
# Move the SIF File to /opt: Use the sudo mv command to move the SIF file to the
/opt directory.
sudo mv server.sif /opt/

# Change Ownership
sudo chown root:root /opt/client.sif
sudo chown root:root /opt/server.sif
#Set Public Visibility: To make the file publicly visible,  use chmod command.
The following command sets read and execute permissions for everyone
sudo chmod 755 /opt/client.sif
sudo chmod 755 /opt/server.sif
```

3. **Manual test: create a script (.sh) to invocate two things:**
    1. **to launch Triton server as daemon**
    2. **that launch the client and execute on this client the inference of the test**

Include the `docker run` command for sending the inference request using the `image_client` example.

```
# Open the Script in Nano

    nano launch_triton_and_client.sh

# Modify the Script:
    #!/bin/bash

    # Launch Triton server as a daemon
    docker run --gpus=1 --rm --net=host -v ${PWD}/model_repository:/models
nvcr.io/nvidia/tritonserver:23.08-py3 tritonserver --model-repository=/models &

    # Wait for the Triton server to initialize (adjust time as needed)
    sleep 10

    # Launch the client and execute inference
    docker run -it --rm --net=host nvcr.io/nvidia/tritonserver:23.08-py3-sdk
/workspace/install/bin/image_client -m densenet_onnx -c 3 -s INCEPTION
/workspace/images/mug.jpg


# Make the Script Executable**: If you haven't already, make the script
executable:
    chmod +x launch_triton_and_client.sh
```

Now, when you run the script using `./launch_triton_and_client.sh`, it will:

- Launch the Triton server as a daemon.

- Wait for 10 seconds to allow the server to initialize.
- Run the `image_client` example to send an inference request for the `densenet_onnx` model using the specified image.

NB: docker stop 123456789abc remember to stop it!

Put model repository in the right place `sudo mv /model_repository /root/`

With Apptainer

```
# Launch Triton server as a daemon
#docker run --gpus=1 --rm --net=host -v ${PWD}/model_repository:/models
nvcr.io/nvidia/tritonserver:23.08-py3 tritonserver --model>
apptainer instance start --bind ${PWD}/model_repository:/models
/cvmfs/local/tritonserver_23.08-py3.sif tritonserver --model-repos>

# Wait for the Triton server to initialize (adjust time as needed)
sleep 10

# Launch the client and execute inference
#docker run -it --rm --net=host nvcr.io/nvidia/tritonserver:23.08-py3-sdk
/workspace/install/bin/image_client -m densenet_onnx -c >
apptainer run /cvmfs/local/tritonserver_23.08-py3-sdk.sif
/workspace/install/bin/image_client -m densenet_onnx -c 3 -s INCEPTION />
```

4. **Create a similar script to launch the Triton server and execute multiple time the client using different images simulating in this way the multiple jobs (if possible, otherwise the same img, depending on model densenet constraints)**

A shell script was used to launches the Triton server and then runs the client multiple times with different images to simulate multiple jobs.

```
# Launch the client and execute inference
#docker run -it --rm --net=host nvcr.io/nvidia/tritonserver:23.08-py3-sdk
/workspace/install/bin/image_client -m densenet_onnx -c >
apptainer run /cvmfs/local/tritonserver_23.08-py3-sdk.sif
/workspace/install/bin/image_client -m densenet_onnx -c 3 -s INCEPTION />

#!/bin/bash

# Launch Triton server as a daemon
docker run --gpus=1 --rm --net=host -v ${PWD}/model_repository:/models -v
/root/image_repository:/workspace/images
nvcr.io/nvidia/>

# Wait for the Triton server to initialize (adjust time as needed)
```

```
sleep 10

# List of images to use for inference
images=("mug1.jpg" "mug2.jpg" "mug3.jpg")

# Loop to execute the client multiple times with different images
for img in "${images[@]}"; do
    echo "Running inference on $img"
    docker run -it --rm --net=host -v /root/image_repository:/workspace/images
nvcr.io/nvidia/tritonserver:23.08-py3-sdk /workspac>done

    done


# **Make the Script Executable**: Back in the terminal, make the script
executable by running:
    chmod +x launch_triton_and_multiple_clients.sh

#execute
./launch_triton_and_multiple_clients.sh
```

Note: Make sure that the paths to the model repository and images are correct and accessible from the containers. Also, adjust the `sleep` time as needed based on your Triton server's initialization time.

To mount the images

cd /mnt/c/Users/saram/Pictures/images/

scp ./* root@fermicloud762.fnal.gov:/root/image_repository

With the apptainter:

```
# Launch Triton server as a daemon
apptainer instance start --bind ${PWD}/model_repository:/models
/cvmfs/local/tritonserver_23.08-py3.sif tritonserver --model-repos
#docker run --gpus=1 --rm --net=host -v ${PWD}/model_repository:/models
nvcr.io/nvidia/tritonserver:23.08-py3 tritonserver --model-repository=/models &

# Wait for the Triton server to initialize (adjust time as needed)
sleep 10

# List of images to use for inference
images=("mug1.jpg" "mug2.jpg" "mug3.jpg")

# Loop to execute the client multiple times with different images
for img in "${images[@]}"; do
```

```
        echo "Running inference on $img"
        # docker run -it --rm --net=host nvcr.io/nvidia/tritonserver:23.08-py3-
sdk /workspace/install/bin/image_client -m densenet_onnx -c 3 -s INCEPTION
/workspace/images/$img
        #apptainer run /cvmfs/local/tritonserver_23.08-py3-sdk.sif
/workspace/install/bin/image_client -m densenet_onnx -c 3 -s INCEPTION
/workspace/images/mug.jpg
        apptainer run --bind "${PWD}/model_repository:/models" --bind
"/root/image_repository:/workspace/images" /cvmfs/local/tritonserver_23.08-py3-
sdk.sif /workspace/install/bin/image_client -m densenet_onnx -c 3 -s INCEPTION
/workspace/images/$img
done
```

5. **Create a custom script in the factory to call the Triton server (modify xml on the factory inserting a path to a .sh script that invokes the triton, as script 3 or 4). Configurate the file vi /etc/gwms-factory/glideinWMS.xml**

Move to file section, in the general location (at the bottom).

```
apptainer instance start --bind ${PWD}/model_repository:/models
/cvmfs/local/tritonserver_23.08-py3.sif tritonserver --model-repos>
```

ITB_CE_Big, fermicloud762

6. **Send an user job (specifying the container to use, Marco will share the command) that use the client container, transfer the image, mount the dir with the image and execute the classification**

In the job submit file: (when there is condor submit)

```
+REQUIRED_OS = “titonclient” OR +SingularityImage = "/cvmfs/local/client” (as
long as we work with 1 site)
```

And add variable in the frontend configuration:

```
SINGULARITY_IMAGES_DICT param with value
"titonclient:/cvmfs/local/client,tritonserver:/cvmfs/local/server"
```

NEXT STEPS

The next main aim is to modify factory code and front end code to automate the startup of the Triton server. By following the steps here reported, we will not only make it easier for users to use the Triton server but also ensure that the system is robust, scalable, and well-documented.

- Conduct a Comprehensive Test: Run multiple jobs in the glidein to ensure stability. Also, perform tests on various resources such as GPU and Wilson cluster to validate compatibility and performance.
- Create a Default Server Image: Generate a default image for the server on CVMFS by creating and building a Dockerfile. Review the existing image options for Triton on the official website and select the most commonly used versions for the default image.
- Simplify Triton Startup in Factory Code: Modify the factory code to enable Triton server startup with a single switch command. Users should be able to start Triton with just one line of code or optionally specify a custom image version.
- Define a Default Client Image: Create a default client image using a Dockerfile. Existing images are large, so investigate ways to reduce the image size (see point 5). A separate client is needed for each job.
- Identify Minimum Requirements for a Lighter Client Image: Determine the minimum requirements needed to offer a runtime or a lighter image for the client. This should be in contrast to loading the full image used in the first test.
- Enhance Frontend Usability for Triton: Modify the frontend code to simplify Triton usage for end-users. Add a switch in the frontend commands to request Triton. Users should also be able to select which site to send the glidein to, making it a bit more complex. In addition to activating configurations for starting the Triton client, users should also be able to choose a site that supports Triton.

In parallel, check use cases:

- Find a Simple Example and Test at Scale: Locate a straightforward example (such as the one involving mugs) and conduct tests on a large scale, involving multiple glide-ins and running numerous jobs simultaneously.
- Identify a More Complex Use Case: Adapt your existing research or classification models to utilize the Triton server. This will serve as a more significant, real-world test case.
- Document the Adaptation Process: Create comprehensive documentation outlining the steps taken to adapt your models for the Triton server. This will serve as a guide for others who wish to do the same.
- Generalize and Educate: Provide a generalized guide on how to use the system, possibly starting with a PyTorch example. Explain how to transition from a Python script (.py) to a program that leverages the Triton server.

By following these steps, you'll be able to test the system's capabilities and also make it easier for others to adapt their projects to use the Triton server.

Utilities

```
#update tokens
 ~/scripts/create-scitoken.sh
create-scitoken.sh #in the frontend!
```

```
#to connect to ce
Condor_ssh_to_job 7 #(number)
# check glidein log files
/var/log/gwms-factory/client/.../entry_ce../job.#.out[err]
```

Locate (yam install) → it is useful to search

CONCLUSIONS

In the realm of scientific computing, the need for substantial computational resources is both pressing and ever-growing. This paper has delved into the intricacies of distributed High Throughput Computing, focusing on the HTCondor framework and its resource provisioning tools, GlideinWMS and HEPCloud. GlideinWMS, functioning as a pilot-oriented provisioning tool, creates consistent virtual clusters by deploying Glideins to diverse and often unreliable resources. This approach has proven to be invaluable for large-scale scientific experiments like CMS and DUNE, which require efficient access to vast computational resources.

The paper also emphasized the critical role of Graphics Processing Units (GPUs) in accelerating various computational tasks, particularly in AI and machine learning workflows. Despite their significance, the availability of GPUs is often limited and not uniformly distributed among worker nodes. To address this bottleneck, the study introduced the NVidia Triton server within containers to enhance GPU utilization. This innovative approach allows for the sharing of limited GPU resources across all jobs managed by a Glidein within a worker node.

Looking ahead, the study explores the possibility of extending this GPU sharing approach to various resources, potentially through proxying mechanisms within the Glidein to access remote Triton servers. This holistic strategy aims to optimize resource utilization and streamline the computational process, thereby addressing some of the most pressing challenges in scientific computing today.

In summary, this research contributes significantly to the field by offering a comprehensive solution for optimizing GPU utilization in a distributed computing environment. It sets the stage for future work that will focus on overcoming current limitations, such as the secure location of Triton servers and the development of proxy mechanisms for more efficient resource sharing. By doing so, the study aims to have a concrete impact on the scientific community, offering a more efficient, scalable, and cost-effective paradigm for scientific computing.

TRAININGS AND REFERENCES

- GlideinWMS Glidein description and Basic Concepts - A GlideinWMS Glossary (Drive repository)
- HT Condor documentation
- VM and containers
- Training Git and Github
    - https://indico.cern.ch/event/1269936/contributions/5354355/attachments/2662158/4612177/git_cms%20(1).pdf
    - https://mambelli.github.io/git-novice/
    - https://learngitbranching.js.org/
- Training shell
    - https://indico.cern.ch/event/1269936/contributions/5348268/attachments/2660880/4610121/Shell_Bash%20notes.pdf
    - https://swcarpentry.github.io/shell-novice/
- Training Apptainer
- Additional material
    - Esercizi: https://www.hackerrank.com/domains/shell
    - > Classic: http://www.tldp.org/LDP/abs/html/
    - > Break it down: http://explainshell.com
    - > Great guide: http://wiki.bash-hackers.org/
    - > Good to know: https://mywiki.wooledge.org/BashPitfalls
- > Unit testing: https://bats-core.readthedocs.io/en/stable/Glidein:
    - https://github.com/glideinWMS/glideinwms
    - https://github.com/glideinWMS/glideinwms/wiki/Development-Workflow
    - Instructions: https://glideinwms.fnal.gov/doc.prd/index.html
    - Custom scripts: https://glideinwms.fnal.gov/doc.prd/factory/custom_scripts.html
- Container and apptainer
    - https://www.docker.com/resources/what-container/
    - https://circleci.com/blog/docker-image-vs-container/
    - https://hsf-training.github.io/hsf-training-singularity-webpage/index.html
    - https://mambelli.github.io/hsf-training-apptainer/index.html