UG3.4: DGX A100 UserGuide

(updated: May 2022)

---

hostname: login.dgx.cineca.it

early availability: 17 Nov 2020

start of preproduction: 08 Jan 2021

start of production: 25 Jan 2021

Model : DGX A100

```
Architecture: Linux Infiniband Cluster
Nodes: 3
Processors: Dual AMD Rome 7742, 128 cores total, 2.25 GHz (base)
3.4 (max boost)
Accelerators: 8x NVIDIA A100 GPUs per node, NVlink 3.0,
              320GB GPU memory per node
RAM: 1TB/node
Internal Network: Mellanox IB EDR fully connected topology

Storage: 15 TB/node NVMe raid0
         95 TB Gluster storage

Peak performance single node:   5 petaFLOPS AI
                               10 petaOPS INT8
```

Access

The access on DGX can be done with **SSH (Secure Shell)** protocol using its hostname:

```
> login.dgx.cineca.it
```

The DGX login node is a virtual machine with 2 cpus and a x86_64 architecture **without GPUs**. The login node is only used for accessing the system, transferring data, and submitting jobs to the DGX nodes.

Accounting

For accounting information please consult our **dedicated section**.

The account_no (or project) is important for batch executions. You need to indicate an account_no to be accounted for in the scheduler, using the flag "-A"

```
#SBATCH -A <account_no>
```

With the "saldo -b" command, you can list all the account_no associated with your username.

```
> saldo -b   (reports projects defined on DGX)
```

Please note that **the accounting is in terms of consumed core hours, but it strongly depends also on the requested memory and number of GPUs.** Please refer to the dedicated section.

Disks and Filesystems

The storage organization conforms to the CINECA infrastructure (see Section "Data storage and Filesystems") as for the user's areas. In addition to the home directory (**$HOME**), for each user is defined a scratch area **$CINECA_SCRATCH**, a disk for storing run time data and files. On this DGX cluster there is no **$WORK** directory.

	Total dimension (TB)	Quota (GB)	Notes
$HOME	5	50	• permanent/backed up, user specific, local
$CINECA_SCRATCH	90	no quota	• permanent, user specific, local

It is also available temporary local storage on compute nodes generated when the job starts and accessible via environment variable **$TMPDIR**.

TMPDIR=/raid/scratch_local/slurm_job.$SLURM_JOB_ID

which can be used **exclusively** by the job's owner. During your jobs, you can access the area with the (local) variable $TMPDIR. In your sbatch script, we suggest moving the input data of your simulations to the $TMPDIR before the beginning of your run and also writing on $TMPDIR your results. The use of this area improves the I/O operations speed of your code by approximately 30%. However, the directory is **removed at the end of the job**, hence always remember to save the data stored in such area to a permanent directory in your sbatch script at the end of the run.

On DGX compute nodes the $TMPDIR local area has ~8 TB of available space. This quota is shared among all users with active jobs on the same compute nodes. For more details, please see the dedicated section of UG2.5: Data storage and FileSystems.

The $DRES filesystem is not mounted on the compute nodes and, at the moment, neither on the login node.

Since DGX filesystems are based on GlusterFS the usual unix command "quota" is not working. Use the local command **cindata** to query for disk usage and quota ("cindata -h" for help):

```
> cindata
```

For information about data transfer from other computers, please follow the instructions and caveats on the dedicated section **Data storage** or the document **Data Management**.

Important: users are enabled to use the local storage of compute nodes (/raid/scratch_local) beyond the $TMPDIR area to improve the I/O performances with respect to the poor ones of GlusterFS. Users can copy the needed dataset in a subdirectory of /raid/scratch_local (e.g. /raid/scratch_local /<username>) in advance through a dedicated job. These data will not be removed at the end of the job unless the compute node is running out of space. So users will find them in the following production jobs.

Please, note that /raid/scratch_local are local areas of compute nodes. So, if the dataset is present on dgx01, user needs to request dgx01 in his/her job script using the directive

```
#SBATCH --nodelist=dgx01
```

Data Management on DGX

DGX compute nodes can download data from a public website directly to the local storage (/raid/scratch_local) through *wget* or *curl* commands. If the dataset is public and could be useful for the community, we encourage you to write an email to superc@cineca.it so we can install the dataset on local storage to avoid multiple copies of the same dataset, given the limited space of the local storage.

For private archive or dataset on a local machine that does not have a public IP reachable by compute nodes, users with an active job running can use *scp* with jump option or *rsync* with ssh tunnel and port forwarding to download them directly on /raid/scratch_local, avoiding an intermediate copy on GlusterFS:

```
> ssh -J <username>@login.dgx.cineca.it dgx02:/raid/scratch_local
```

or

```
> scp -o ProxyJump=<username>@login.dgx.cineca.it file.tar.gz dgx02:/raid/scratch_local/


> ssh -N -L 2222:dgx01:22 <username>@login.dgx.cineca.it &
> rsync -auve "ssh -p 2222" file.tar.gz <username>@localhost:/raid/scratch_local/
```

Modules environment

The DGX module environment is based on lmod and it is minimal, only a few modules being available on the cluster:

- HPC SDK: nvidia compilers and libraries
- OpenMPI: OpenMPI 4.0.3 (on compute nodes only)
- AI datasets: common AI datasets, such as ImageNet, available locally on each compute node.
- Singularity: container platform, needed to pull, and execute containers on DGX compute node.

To see all the installed module, you can type "module av" or "module spider".

We have installed a few modules to encourage users to run their simulations using container technology (via singularity) See below section How to use Singularity on DGX. You don't need to build your container, in most cases, but you can pull (download) it from Nvidia, Docker or Singularity repositories.

AI datasets modules

The AI datasets are organized into audio, images, and videos. The datasets are stored on a shared directory AND on the NVMe memories to provide significantly faster access inside jobs. Two sets of variables are hence defined by loading these modules, pointing to the location accessible on the login node and to the location available ONLY on the compute nodes (which we encourage to use inside jobs). For instance, the imagenet ILSVRC2012 dataset module defines the following variables:

- $DATA_ILSVRC2012_TRAIN : training dataset on the compute nodes only
- $DATA_ILSVRC2012_TRAIN_LOGIN: training dataset on login and compute nodes

- $DATA_ILSVRC2012_VAL : validation dataset on the compute nodes only
- $DATA_ILSVRC2012_VAL_LOGIN : validation dataset on login and compute nodes

With the command "module help <module_name>" you will find the list of variables defined for each data module.

Production environment

Since DGX is a general purpose system and several users use it at the same time, long production jobs must be submitted using a queuing system (scheduler). This guarantees that taccess to the resources is as fair as possible. On DGX the available scheduler is SLURM.

DGX cluster is based on a policy of node sharing among different jobs, and users can at most request one full node for their jobs. This means that, at a given time, one physical node can be allocated to multiple jobs of different users. Nevertheless, exclusivity at the level of the single core and GPU is guaranteed by low-level mechanisms.

Roughly speaking, there are two different modes to use an HPC system: Interactive and Batch. For a general discussion, see the section Production Environment.

Interactive

A serial program can be executed in the standard UNIX way:

```
> ./program
```

This is allowed only for very short runs on the login node, since the **interactive environment has a 10 minutes cpu-time limit.** Please do not execute parallel applications on the login node, and keep in mind that it is a virtual machine with 2 cpus and no GPUs.

A serial (or parallel) program, also using GPUs and needing more than 10 minutes can be executed interactively within an **"interactive" SLURM batch job.**

A request for resources allocation on the compute nodes is delivered to SLURM with salloc/srun commands, the request is queued and scheduled as any other batch job but, when granted, the standard input, output, and error streams of the interactive job connected to the terminal session from which the request was launched.

For example, to start an interactive session on one node and get the full node in exclusive way (including the eight GPUs) for one hour, launch the command:

```
> srun -N1 --exclusive --gres=gpu:8 -A <account_name> -p <partition_name> --time=01:00:00 --pty bash
```

Due to the slurm node configuration (defined with Sockets=2 CoresPerSocket=64 ThreadsPerCore=2 Procs=128 CPUs=256 Gres=gpu:8), the cores are always assigned with both threads (virtual cpus) so to provide a total number of virtual cpus equal to the number of requested tasks. Hence, if you request two tasks per node, two HTs of one physical core will be assigned to the job. If you want two physical cores you also have to specify that each task requests two (virtual) cpus (--cpus-per-tasks=2).

```
> srun -N1 --ntasks-per-node=2 --cpus-per-task=2 --gres=gpu:2 -A <account_name> -p <partition_name> --time=01:00:
00 --pty bash
```

SLURM automatically exports the environment variables you defined in the source shell on the login node. If you need a specific environment for your job ((i.e. specific library paths or options) you can "prepare" it on the login node before submitting your job.

A more specific description of the options used by salloc/srun to allocate resources in the “Batch” section, because they are the same of the sbatch command described there.

Batch

The production runs are executed in batch mode: the user writes a list of commands into a file (for example script.x) and then submits it to a scheduler (SLURM for DGX) that will search for the required resources in the system. As soon as the resources are available script.x is executed.

This is an example of a script file:

```
#!/bin/bash
#SBATCH -A <account_name>
#SBATCH -p dgx_usr_prod
#SBATCH --time 00:10:00     # format: HH:MM:SS
#SBATCH -N 1                # 1 node
#SBATCH --ntasks-per-node=8 # 8 tasks
#SBATCH --gres=gpu:1        # 1 gpus per node out of 8
#SBATCH --mem=7100          # memory per node out of 980000 MB
#SBATCH --job-name=my_batch_job
#SBATCH --mail-type=ALL
#SBATCH --mail-user=<user_email>


<your commands>
```

As stated in the previous section, by requesting --ntasks-per-node=8 your job will be assigned 8 logical cpus (hence, the first 4 cpus with their 2 HTs). You can write your script file (for example script.x) using any editor, then you submit it using the command:

```
> sbatch script.x
```

The script file must contain both directives to SLURM and commands to be executed, as better described in the section Batch Scheduler SLURM.

On DGX there are two overlapped partitions: dgx_usr_prod (the default partition) and dgx_usr_preempt. dgx_usr_prod is the high priority partition, on this partition you can only have only one job running at the same time. dgx_usr_preempt is the low priority partition, it is free of charge, there is no limit on the number of job running at the same time, but your jobs may be killed if a job on the high priority partition (dgx_usr_prod) need resources.

One of the commands will be probably the launch of a singularity container application (see Section How to use Singularity on DGX).

Summary

In the following table, you can find all the main features and limits imposed on the SLURM partitions and QOS.

SLURM partition	**QOS**	**#cores/#GPUs per job**	**max walltime**	**max running jobs per user/ max n. of cores /GPUs/nodes per user**	**Priority**	**notes**
dgx_usr_prod	dgx_qos_sprod	max = 32 cores (64 cpus) / 2 GPUs max mem=245000MB	48 h	1 job per user 32 cores (64 cpus) / 2 GPUs	30	
	normal (noQOS)	max = 128 cores (256 cpus) / 8 GPUs max mem=980000MB	4 h	1 job per user 8 GPUs / 1 node per user	40	
dgx_usr_preempt	dgx_qos_sprod	max = 32 cores (64 cpus) / 2 GPUs max mem=245000MB	48 h	(no limit)	1	free of charge / your jobs may be killed in any moment if a high priority job requests for resources in dgx_usr_prod partition
	normal (noQOS)	max = 128 cores (256 cpus) / 8 GPUs max mem=980000MB	24 h	(no limit)	1	free of charge / your jobs may be killed in any moments if a high priority job requests for resources in dgx_usr_prod partition

How to use Singularity on DGX

On each node of DGX cluster singularity is installed in the default path. You don't need to load singularity module in order to use it, but we have created it to provide some examples to users via the command "*module help tools/singularity*". If you need more info you can type *singularity --help* or visit Singularity documentation web site.

Pull a container

To pull a container from a repository you need to execute the command *singularity pull <container location>*. For example if you want to download pytorch container from docker repository you can use this command:

pull a container

```
singularity pull docker://nvcr.io/nvidia/pytorch:20.12-py3
```

While pulling a container form a repository you may encounter some issues:

1. **`no space left on device`**. This error happens often because singularity uses /tmp directory to store temporary files. Since /tmp size is quite small, when it is full you obtain this error and you cannot perform container pull. We suggest to set the variable SINGULARITY_TMPDIR in order to set a different temporary directory for Singularity, e.g.
   `$ export SINGULARITY_TMPDIR=$CINECA_SCRATCH/tmp`
2. **`CPU time limit exceeded`**. This error happens when the container you want to pull is quite large and the last stage (creating SIF image) needs to much CPU time. On the login node each process has 10 minutes of CPU time limit. In order to avoid this issue we suggest to request an interactive job to pull it, or pull it on your laptop and copy it on DGX cluster.

Run a container

Once you have pulled a container, running it is quite easy. Here an interactive example about pytorch container:

Run a container in interactive mode

```
fcola000@dgxslurm:~$ git clone https://github.com/pytorch/examples.git
fcola000@dgxslurm:~$ srun -p dgx_usr_prod -N1 -n32 --gres=gpu:2 -t 01:00:00 --pty /bin/bash
fcola000@dgx01:~$ cd examples/mnist
fcola000@dgx01:~examples/mnist$ singularity exec --nv $HOME/pytorch_20.12-py3 python main.py
```

You can launch the same computation using a batch script:

Run a container in batch mode

```
fcola000@dgxslurm:~$ cat submit.sh
#!/bin/bash

#SBATCH -N1
#SBATCH -n 32
#SBATCH -p dgx_usr_prod
#SBATCH -t 01:00:00
#SBATCH --gres=gpu:2
#SBATCH -o job_%J.out
#SBATCH -e job_%J.err

git clone https://github.com/pytorch/examples.git
cd $HOME/examples/mnist
singularity exec --nv $HOME/pytorch_20.12-py3.sif python main.py

fcola000@dgxslurm:~$ sbatch submit.sh
```

Here you can find more examples about pytorch.