Generic Resource (GRES) Scheduling

Contents §

Overview §

Slurm supports the ability to define and schedule arbitrary Generic RESources (GRES). Additional built-in features are enabled for specific GRES types, including Graphics Processing Units (GPUs), CUDA Multi-Process Service (MPS) devices, and Sharding through an extensible plugin mechanism.

Configuration §

By default, no GRES are enabled in the cluster's configuration. You must explicitly specify which GRES are to be managed in the *slurm.conf* configuration file. The configuration parameters of interest are **GresTypes** and **Gres**.

For more details, see GresTypes and Gres in the *slurm.conf* man page.

Note that the GRES specification for each node works in the same fashion as the other resources managed. Nodes which are found to have fewer resources than configured will be placed in a DRAIN state.

Snippet from an example *slurm.conf* file:

```
# Configure four GPUs (with MPS), plus bandwidth
GresTypes=gpu,mps,bandwidth
NodeName=tux[0-7] Gres=gpu:tesla:2,gpu:kepler:2,mps:400,bandwidth:lustre:no
```

Each compute node with generic resources typically contain a *gres.conf* file describing which resources are available on the node, their count, associated device files and cores which should be used with those resources.

There are cases where you may want to define a Generic Resource on a node without specifying a quantity of that GRES. For example, the filesystem type of a node doesn't decrease in value as jobs run on that node. You can use the **no_consume** flag to allow users to request a GRES without having a defined count that gets used as it is requested.

To view available *gres.conf* configuration parameters, see the gres.conf man page.

Running Jobs §

Jobs will not be allocated any generic resources unless specifically requested at job submit time using the options:

--gres
  Generic resources required per node
--gpus
  GPUs required per job
--gpus-per-node

GPUs required per node. Equivalent to the *--gres* option for GPUs.

--gpus-per-socket
: GPUs required per socket. Requires the job to specify a task socket.

--gpus-per-task
: GPUs required per task. Requires the job to specify a task count.

All of these options are supported by the *salloc*, *sbatch* and *srun* commands. Note that all of the *--gpu** options are only supported by Slurm's select/cons_tres plugin. Jobs requesting these options when the select/cons_tres plugin is <u>not</u> configured will be rejected. The *--gres* option requires an argument specifying which generic resources are required and how many resources using the form *name[:type:count]* while all of the *--gpu** options require an argument of the form *[type]:count*. The *name* is the same name as specified by the *GresTypes* and *Gres* configuration parameters. *type* identifies a specific type of that generic resource (e.g. a specific model of GPU). *count* specifies how many resources are required and has a default value of 1. For example:
sbatch --gres=gpu:kepler:2 ....

Requests for typed vs non-typed generic resources must be consistent within a job. For example, if you request *--gres=gpu:2* with **sbatch**, you would not be able to request *--gres=gpu:tesla:2* with **srun** to create a job step. The same holds true in reverse, if you request a typed GPU to create a job allocation, you should request a GPU of the same type to create a job step.

Several additional resource requirement specifications are available specifically for GPUs and detailed descriptions about these options are available in the man pages for the job submission commands. As for the *--gpu** option, these options are only supported by Slurm's select/cons_tres plugin.

--cpus-per-gpu
: Count of CPUs allocated per GPU.

--gpu-bind
: Define how tasks are bound to GPUs.

--gpu-freq
: Specify GPU frequency and/or GPU memory frequency.

--mem-per-gpu
: Memory allocated per GPU.

Jobs will be allocated specific generic resources as needed to satisfy the request. If the job is suspended, those resources do not become available for use by other jobs.

Job steps can be allocated generic resources from those allocated to the job using the *--gres* option with the *srun* command as described above. By default, a job step will be allocated all of the generic resources that have been requested by the job, except those implicitly requested when a job is exclusive. If desired, the job step may explicitly specify a different generic resource count than the job. This design choice was based upon a scenario where each job executes many job steps. If job steps were granted access to all generic resources by default, some job steps would need to explicitly specify zero generic resource counts, which we considered more confusing. The job step can be allocated specific generic resources and those resources will not be available to other job steps. A simple example is shown below.

```
#!/bin/bash
#
# gres_test.bash
# Submit as follows:
# sbatch --gres=gpu:4 -n4 -N1-1 gres_test.bash
#
srun --gres=gpu:2 -n2 --exclusive show_device.sh &
srun --gres=gpu:1 -n1 --exclusive show_device.sh &
srun --gres=gpu:1 -n1 --exclusive show_device.sh &
wait
```

AutoDetect §

If *AutoDetect=nvml*, *AutoDetect=nvidia*, *AutoDetect=rsmi*, *AutoDetect=nrt*, or *AutoDetect=oneapi* are set in *gres.conf*, configuration details will automatically be filled in for any system-detected GPU. This removes the need to explicitly configure GPUs in gres.conf, though the *Gres*= line in slurm.conf is still required in order to tell slurmctld how many GRES to expect.

Note that *AutoDetect=nvml*, *AutoDetect=rsmi*, and *AutoDetect=oneapi* need their corresponding GPU management libraries installed on the node and found during Slurm configuration in order to work. Both *AutoDetect=nvml* and *AutoDetect=nvidia* detect

NVIDIA GPUs. *AutoDetect=nvidia* (added in Slurm 24.11) doesn't require the nvml library to be installed, but doesn't detect MIGs or NVlinks.

By default, all system-detected devices are added to the node. However, if *Type* and *File* in gres.conf match a GPU on the system, any other properties explicitly specified (e.g. *Cores* or *Links*) can be double-checked against it. If the system-detected GPU differs from its matching GPU configuration, then the GPU is omitted from the node with an error. This allows *gres.conf* to serve as an optional sanity check and notifies administrators of any unexpected changes in GPU properties.

If not all system-detected devices are specified by the slurm.conf configuration, then the relevant slurmd will be drained. However, it is still possible to use a subset of the devices found on the system if they are specified manually (with AutoDetect disabled) in gres.conf.

Example *gres.conf* file:

```
# Configure four GPUs (with MPS), plus bandwidth
AutoDetect=nvml
Name=gpu Type=gp100  File=/dev/nvidia0 Cores=0,1
Name=gpu Type=gp100  File=/dev/nvidia1 Cores=0,1
Name=gpu Type=p6000  File=/dev/nvidia2 Cores=2,3
Name=gpu Type=p6000  File=/dev/nvidia3 Cores=2,3
Name=mps Count=200  File=/dev/nvidia0
Name=mps Count=200  File=/dev/nvidia1
Name=mps Count=100  File=/dev/nvidia2
Name=mps Count=100  File=/dev/nvidia3
Name=bandwidth Type=lustre Count=4G Flags=CountOnly
```

In this example, since *AutoDetect=nvml* is specified, *Cores* for each GPU will be checked against a corresponding GPU found on the system matching the *Type* and *File* specified. Since *Links* is not specified, it will be automatically filled in according to what is found on the system. If a matching system GPU is not found, no validation takes place and the GPU is assumed to be as the configuration says.

For *Type* to match a system-detected device, it must either exactly match or be a substring of the GPU name reported by slurmd via the AutoDetect mechanism. This

GPU name will have all spaces replaced with underscores. To see the detected GPUs and their names, run: `slurmd -C`

```
$ slurmd -C
NodeName=node0 ... Gres=gpu:geforce_rtx_2060:1 ...
Found gpu:geforce_rtx_2060:1 with Autodetect=nvml (Substring of gpu name ma
UpTime=...
```

In this example, the GPU's name is reported as `geforce_rtx_2060`. So in your slurm.conf and gres.conf, the GPU *Type* can be set to `geforce`, `rtx`, `2060`, `geforce_rtx_2060`, or any other substring, and **slurmd** should be able to match it to the system-detected device `geforce_rtx_2060`. To check your configuration you may run: `slurmd -G` This will test and print the gres setup based on the current configuration, including any autodetected gres that are being ignored.

Accounting §

GPU memory and GPU utilization can be tracked as TRES for tasks using GPU resources. If `AccountingStorageTRES=gres/gpu` is configured, gres/gpumem and gres/gpuutil will automatically be configured and gathered from GPU jobs. gres/gpumem and gres/gpuutil can also be set individually when gres/gpu is not set.

gres/gpumem and gres/gpuutil are only available for NVIDIA GPUs when using `AutoDetect=nvml`, and AMD GPUs when using `AutoDetect=rsmi`.

NVML does not support utilization metrics for MIGs, so Slurm does not provide gpumem or gpuutil accounting for MIG devices. See NVIDIA's MIG User Guide.

Here is an example node with two NVIDIA A100 GPUs:

```
$ nvidia-smi -L
GPU 0: NVIDIA A100-PCIE-40GB (UUID: GPU-...)
GPU 1: NVIDIA A100-PCIE-40GB (UUID: GPU-...)
```

The following is an excerpt from an example slurm.conf and gres.conf which will automatically enable tracking for gres/gpumem and gres/gpuutil.

slurm.conf:

```
AccountingStorageTres=gres/gpu
NodeName=n1 Gres=gpu:a100:2
```

gres.conf:

```
AutoDetect=nvml
```

Here's an example of a job with two tasks that uses one GPU per task, two GPUs total.

```
$ srun --tres-per-task=gres/gpu:1 -n2 --gpus=2 --mem=2G gpu_burn
GPU 0: NVIDIA A100-PCIE-40GB (UUID: GPU-...)
GPU 0: NVIDIA A100-PCIE-40GB (UUID: GPU-...)
```

After the job has finished, we can see utilization details of these TRES using sacct. Note that there are multiple tasks here and we can use TRESUsageIn[Min,Max,Ave,Tot] to examine the "highwater marks" for gpumem and gpuutil over all the ranks in the step, as is true for other TRESUsage* values:

```
$ sacct -j 1277.0 --format=tresusageinave -p
TRESUsageInAve|
cpu=00:00:11,energy=0,fs/disk=87613,gres/gpumem=36266M,gres/gpuutil=100,mem
$ sacct -j 1277.0 --format=tresusageintot -p
TRESUsageInTot|
cpu=00:00:22,energy=0,fs/disk=175227,gres/gpumem=72532M,gres/gpuutil=200,me
```

GPU Management §

In the case of Slurm's GRES plugin for GPUs, the environment variable `CUDA_VISIBLE_DEVICES` is set for each job step to determine which GPUs are available for its use on each node. This environment variable is only set when tasks are launched on a specific compute node (no global environment variable is set for the *salloc* command and the environment variable set for the *sbatch* command only reflects the GPUs allocated to that job on that node, node zero of the allocation). CUDA version 3.1 (or higher) uses this environment variable in order to run multiple jobs or job steps on a node with GPUs and ensure that the resources assigned to each are unique. In the example above, the allocated node may have four or more graphics devices. In that case, `CUDA_VISIBLE_DEVICES` will reference unique devices for each file and the output might resemble this:

```
JobStep=1234.0 CUDA_VISIBLE_DEVICES=0,1
JobStep=1234.1 CUDA_VISIBLE_DEVICES=2
JobStep=1234.2 CUDA_VISIBLE_DEVICES=3
```

NOTE: Be sure to specify the *File* parameters in the *gres.conf* file and ensure they are in the increasing numeric order.

The `CUDA_VISIBLE_DEVICES` environment variable will also be set in the job's Prolog and Epilog programs. Note that the environment variable set for the job may differ from that set for the Prolog and Epilog if Slurm is configured to constrain the device files visible to a job using Linux cgroup. This is because the Prolog and Epilog programs run outside of any Linux cgroup while the job runs inside of the cgroup and may thus have a different set of visible devices. For example, if a job is allocated the device "/dev/nvidia1", then `CUDA_VISIBLE_DEVICES` will be set to a value of "1" in the Prolog and Epilog while the job's value of `CUDA_VISIBLE_DEVICES` will be set to a value of "0" (i.e. the first GPU device visible to the job). For more information see the Prolog and Epilog Guide.

When possible, Slurm automatically determines the GPUs on the system using NVML. NVML (which powers the `nvidia-smi` tool) numbers GPUs in order by their PCI bus IDs. For this numbering to match the numbering reported by CUDA, the `CUDA_DEVICE_ORDER` environmental variable must be set to `CUDA_DEVICE_ORDER=PCI_BUS_ID`.

GPU device files (e.g. */dev/nvidia1*) are based on the Linux minor number assignment, while NVML's device numbers are assigned via PCI bus ID, from lowest to highest. Mapping between these two is nondeterministic and system dependent, and could vary between boots after hardware or OS changes. For the most part, this assignment seems fairly stable. However, an after-bootup check is required to guarantee that a GPU device is assigned to a specific device file.

Please consult the NVIDIA CUDA documentation for more information about the `CUDA_VISIBLE_DEVICES` and `CUDA_DEVICE_ORDER` environmental variables.

MPS Management §

CUDA Multi-Process Service (MPS) provides a mechanism where GPUs can be shared by multiple jobs, where each job is allocated some percentage of the GPU's resources. The total count of MPS resources available on a node should be configured in the *slurm.conf* file (e.g. "NodeName=tux[1-16] Gres=gpu:2,mps:200"). Several options are available for configuring MPS in the *gres.conf* file as listed below with examples following that:

1. No MPS configuration: The count of gres/mps elements defined in the *slurm.conf* will be evenly distributed across all GPUs configured on the node. For example, "NodeName=tux[1-16] Gres=gpu:2,mps:200" will configure a count of 100 gres/mps resources on each of the two GPUs.
2. MPS configuration includes only the *Name* and *Count* parameters: The count of gres/mps elements will be evenly distributed across all GPUs configured on the node. This is similar to case 1, but places duplicate configuration in the gres.conf file.
3. MPS configuration includes the *Name*, *File* and *Count* parameters: Each *File* parameter should identify the device file path of a GPU and the *Count* should identify the number of gres/mps resources available for that specific GPU device. This may be useful in a heterogeneous environment. For example, some GPUs on a node may be more powerful than others and thus be associated with a higher gres/mps count. Another use case would be to prevent some GPUs from being used for MPS (i.e. they would have an MPS count of zero).

Note that *Type* and *Cores* parameters for gres/mps are ignored. That information is copied from the gres/gpu configuration.

Note the *Count* parameter is translated to a percentage, so the value would typically be a multiple of 100.

Note that if NVIDIA's NVML library is installed, the GPU configuration (i.e. *Type*, *File*, *Cores* and *Links* data) will be automatically gathered from the library and need not be recorded in the *gres.conf* file.

By default, job requests for MPS are required to fit on a single gpu on each node. This can be overridden with a flag in the *slurm.conf* configuration file. See OPT_MULTIPLE_SHARING_GRES_PJ.

Note the same GPU can be allocated either as a GPU type of GRES or as an MPS type of GRES, but not both. In other words, once a GPU has been allocated as a gres/gpu resource it will not be available as a gres/mps. Likewise, once a GPU has been allocated as a gres/mps resource it will not be available as a gres/gpu. However the same GPU can be allocated as MPS generic resources to multiple jobs belonging to multiple users, so long as the total count of MPS allocated to jobs does not exceed the configured count. Also, since shared GRES (MPS) cannot be allocated at the same time as a sharing GRES (GPU) this option only allocates all sharing GRES and no underlying shared GRES. Some example configurations for Slurm's gres.conf file are shown below.

```
# Example 1 of gres.conf
# Configure four GPUs (with MPS)
AutoDetect=nvml
Name=gpu Type=gp100 File=/dev/nvidia0 Cores=0,1
Name=gpu Type=gp100 File=/dev/nvidia1 Cores=0,1
Name=gpu Type=p6000 File=/dev/nvidia2 Cores=2,3
Name=gpu Type=p6000 File=/dev/nvidia3 Cores=2,3
# Set gres/mps Count value to 100 on each of the 4 available GPUs
Name=mps Count=400
```

```
# Example 2 of gres.conf
# Configure four different GPU types (with MPS)
```

```
AutoDetect=nvml
Name=gpu Type=gtx1080 File=/dev/nvidia0 Cores=0,1
Name=gpu Type=gtx1070 File=/dev/nvidia1 Cores=0,1
Name=gpu Type=gtx1060 File=/dev/nvidia2 Cores=2,3
Name=gpu Type=gtx1050 File=/dev/nvidia3 Cores=2,3
Name=mps Count=1300   File=/dev/nvidia0
Name=mps Count=1200   File=/dev/nvidia1
Name=mps Count=1100   File=/dev/nvidia2
Name=mps Count=1000   File=/dev/nvidia3
```

NOTE: *gres/mps* requires the use of the *select/cons_tres* plugin.

Job requests for MPS will be processed the same as any other GRES except that the request must be satisfied using only one GPU per node and only one GPU per node may be configured for use with MPS. For example, a job request for "--gres=mps:50" will not be satisfied by using 20 percent of one GPU and 30 percent of a second GPU on a single node. Multiple jobs from different users can use MPS on a node at the same time. Note that GRES types of GPU and MPS can not be requested within a single job. Also jobs requesting MPS resources can not specify a GPU frequency.

A prolog program should be used to start and stop MPS servers as needed. A sample prolog script to do this is included with the Slurm distribution in the location *etc/prolog.example*. Its mode of operation is if a job is allocated gres/mps resources then the Prolog will have the `CUDA_VISIBLE_DEVICES`, `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE`, and `SLURM_JOB_UID` environment variables set. The Prolog should then make sure that an MPS server is started for that GPU and user (UID == User ID). It also records the GPU device ID in a local file. If a job is allocated gres/gpu resources then the Prolog will have the `CUDA_VISIBLE_DEVICES` and `SLURM_JOB_UID` environment variables set (no `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE`). The Prolog should then terminate any MPS server associated with that GPU. It may be necessary to modify this script as needed for the local environment. For more information see the Prolog and Epilog Guide.

Jobs requesting MPS resources will have the `CUDA_VISIBLE_DEVICES` and `CUDA_DEVICE_ORDER` environment variables set. The device ID is relative to those resources under MPS server control and will always have a value of zero in the current

implementation (only one GPU will be usable in MPS mode per node). The job will also have the `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` environment variable set to that job's percentage of MPS resources available on the assigned GPU. The percentage will be calculated based upon the portion of the configured Count on the Gres is allocated to a job of step. For example, a job requesting "--gres=mps:200" and using configuration example 2 above would be allocated
15% of the gtx1080 (File=/dev/nvidia0, 200 x 100 / 1300 = 15), or
16% of the gtx1070 (File=/dev/nvidia0, 200 x 100 / 1200 = 16), or
18% of the gtx1060 (File=/dev/nvidia0, 200 x 100 / 1100 = 18), or
20% of the gtx1050 (File=/dev/nvidia0, 200 x 100 / 1000 = 20).

An alternate mode of operation would be to permit jobs to be allocated whole GPUs then trigger the starting of an MPS server based upon comments in the job. For example, if a job is allocated whole GPUs then search for a comment of "mps-per-gpu" or "mps-per-node" in the job (using the "scontrol show job" command) and use that as a basis for starting one MPS daemon per GPU or across all GPUs respectively.

Please consult the NVIDIA Multi-Process Service documentation for more information about MPS.

Note that a vulnerability exists in previous versions of the NVIDIA driver that may affect users when sharing GPUs. More information can be found in CVE-2018-6260 and in the Security Bulletin: NVIDIA GPU Display Driver - February 2019.

NVIDIA MPS has a built-in limitation regarding GPU sharing among different users. Only one user on a system may have an active MPS server, and the MPS control daemon will queue MPS server activation requests from separate users, leading to serialized exclusive access of the GPU between users (see Section 2.3.1.1 - Limitations in the MPS docs). So different users cannot truly run concurrently on the GPU with MPS; rather, the GPU will be time-sliced between the users (for a diagram depicting this process, see Section 3.3 - Provisioning Sequence in the MPS docs).

MIG Management §

Beginning in version 21.08, Slurm now supports NVIDIA *Multi-Instance GPU* (MIG) devices. This feature allows some newer NVIDIA GPUs (like the A100) to split up a GPU into up to seven separate, isolated GPU instances. Slurm can treat these MIG instances as individual GPUs, complete with cgroup isolation and task binding.

To configure MIGs in Slurm, specify `AutoDetect=nvml` in *gres.conf* for the nodes with MIGs, and specify `Gres` in *slurm.conf* as if the MIGs were regular GPUs, like this: `NodeName=tux[1-16] gres=gpu:2`. An optional GRES type can be specified to distinguish MIGs of different sizes from each other, as well as from other GPUs in the cluster. This type must be a substring of the "MIG Profile" string as reported by the node in its slurmd log under the `debug2` log level. Here is an example slurm.conf for a system with 2 gpus, one of which is partitioned into 2 MIGs where the "MIG Profile" is `nvidia_a100_3g.20gb`:

```
AccountingStorageTRES=gres/gpu,gres/gpu:a100,gres/gpu:a100_3g.20gb
GresTypes=gpu
NodeName=tux[1-16] gres=gpu:a100:1,gpu:a100_3g.20gb:2
```

The MultipleFiles parameter allows you to specify multiple device files for the GPU card.

The sanity-check AutoDetect mode is not supported for MIGs. Slurm expects MIG devices to already be partitioned, and does not support dynamic MIG partitioning.

For more information on NVIDIA MIGs (including how to partition them), see the MIG user guide.

Sharding §

Sharding provides a generic mechanism where GPUs can be shared by multiple jobs. While it does permit multiple jobs to run on a given GPU it does not fence the processes running on the GPU, it only allows the GPU to be shared. Sharding, therefore, works best with homogeneous workflows. It is recommended to limit the number of shards on a node to equal the max possible jobs that can run simultaneously on the node (i.e. cores). The total count of shards available on a node should be configured in the *slurm.conf* file (e.g. "NodeName=tux[1-16] Gres=gpu:2,shard:64"). Several options are available for configuring shards in the *gres.conf* file as listed below with examples following that:

1. No Shard configuration: The count of gres/shard elements defined in the *slurm.conf* will be evenly distributed across all GPUs configured on the node. For example, "NodeName=tux[1-16] Gres=gpu:2,shard:64" will configure a count of 32 gres/shard resources on each of the two GPUs.
2. Shard configuration includes only the *Name* and *Count* parameters: The count of gres/shard elements will be evenly distributed across all GPUs configured on the node. This is similar to case 1, but places duplicate configuration in the gres.conf file.
3. Shard configuration includes the *Name*, *File* and *Count* parameters: Each *File* parameter should identify the device file path of a GPU and the *Count* should identify the number of gres/shard resources available for that specific GPU device. This may be useful in a heterogeneous environment. For example, some GPUs on a node may be more powerful than others and thus be associated with a higher gres/shard count. Another use case would be to prevent some GPUs from being used for sharding (i.e. they would have a shard count of zero).

Note that *Type* and *Cores* parameters for gres/shard are ignored. That information is copied from the gres/gpu configuration.

Note that if NVIDIA's NVML library is installed, the GPU configuration (i.e. *Type*, *File*, *Cores* and *Links* data) will be automatically gathered from the library and need not be recorded in the *gres.conf* file.

Note the same GPU can be allocated either as a GPU type of GRES or as a shard type of GRES, but not both. In other words, once a GPU has been allocated as a gres/gpu resource it will not be available as a gres/shard. Likewise, once a GPU has been allocated as a gres/shard resource it will not be available as a gres/gpu. However the same GPU can be allocated as shard generic resources to multiple jobs belonging to multiple users, so long as the total count of SHARD allocated to jobs does not exceed the configured count.

By default, job requests for shards are required to fit on a single gpu on each node. This can be overridden with a flag in the *slurm.conf* configuration file. See OPT_MULTIPLE_SHARING_GRES_PJ.

In order for this to be correctly configured, the appropriate nodes need to have the *shard* keyword added as a GRES for the relevant nodes as well as being added to the *GresTypes* parameter. If you want the shards to be tracked in accounting then *shard* also needs to be added to *AccountingStorageTRES*. See the relevant settings in an example slurm.conf:

```
AccountingStorageTRES=gres/gpu,gres/shard
GresTypes=gpu,shard
NodeName=tux[1-16] Gres=gpu:2,shard:64
```

Some example configurations for Slurm's gres.conf file are shown below.

```
# Example 1 of gres.conf
# Configure four GPUs (with Sharding)
AutoDetect=nvml
Name=gpu Type=gp100 File=/dev/nvidia0 Cores=0,1
Name=gpu Type=gp100 File=/dev/nvidia1 Cores=0,1
Name=gpu Type=p6000 File=/dev/nvidia2 Cores=2,3
Name=gpu Type=p6000 File=/dev/nvidia3 Cores=2,3
# Set gres/shard Count value to 8 on each of the 4 available GPUs
Name=shard Count=32
```

```
# Example 2 of gres.conf
# Configure four different GPU types (with Sharding)
AutoDetect=nvml
Name=gpu Type=gtx1080 File=/dev/nvidia0 Cores=0,1
Name=gpu Type=gtx1070 File=/dev/nvidia1 Cores=0,1
Name=gpu Type=gtx1060 File=/dev/nvidia2 Cores=2,3
Name=gpu Type=gtx1050 File=/dev/nvidia3 Cores=2,3
Name=shard Count=8    File=/dev/nvidia0
Name=shard Count=8    File=/dev/nvidia1
Name=shard Count=8    File=/dev/nvidia2
Name=shard Count=8    File=/dev/nvidia3
```

NOTE: *gres/shard* requires the use of the *select/cons_tres* plugin.

Job requests for shards can not specify a GPU frequency.

Jobs requesting shards resources will have the `CUDA_VISIBLE_DEVICES`, `ROCR_VISIBLE_DEVICES`, or `GPU_DEVICE_ORDINAL` environment variable set which would be the same as if it were a GPU.

Steps with shards have `SLURM_SHARDS_ON_NODE` set indicating the number of shards allocated.

Last modified 10 April 2025