



Denglin Hamming™ V2

Multi-instance GPU (MIG)

Feature Guide

DL-DG/SW-044A-03

2024-09-02

Copyright©苏州登临科技有限公司，2019 - 2025，版权所有。

未经是苏州登临科技有限公司事先书面同意，不得以任何形式或方式复制或传播本文件的任何部分。

商标和许可



和其它苏州登临科技有限公司的其它登临科技的图标为苏州登临科技有限公司的商标。本手册中提及的所有其他商标均为其各自所有者的财产。

通知

所购买的产品、服务和特性由苏州登临科技有限公司与客户签订的合同规定。本文件中描述的所有或部分产品、服务和特性可能不在采购范围或使用范围内。除非合同中另有规定，本文件中的所有声明、信息和建议均按“原样”提供，无任何明示或暗示的保证或陈述。

本手册中的信息如有更改，恕不另行通知。本文件在编制过程中已尽一切努力确保内容的准确性，本文件中的所有声明、信息和建议不构成任何明示或暗示的保证。

苏州登临科技有限公司

苏州工业园区扬富路11号南岸新地一期商务楼栋5号1101室，江苏，中国

<http://www.denglin.ai>

email : support@denglin.ai

Change History

Version	Description
03	correct the typo.
02	Update product models.
01	Initial version.

Contents

[Change History](#)

[Contents](#)

[1 Overview](#)

[1.1 Introduction](#)

[1.2 Concepts](#)

[1.2.1 Terminology](#)

[1.2.2 Partitioning](#)

[1.2.2.1 GPU Instance](#)

[1.2.2.2 Profile Placement](#)

[1.2.3 CUDA Concurrency Mechanisms](#)

[2 MIG Device Names](#)

[2.1 Device Enumeration](#)

[2.2 CUDA Device Enumeration](#)

[3 Supported MIG Profiles](#)

[3.1 GS40 Profiles](#)

[3.2 GS30 Profiles](#)

[3.3 UL Profiles](#)

[4 Getting Started with MIG](#)

[4.1 Enable MIG Mode](#)

[4.2 List GPU Instance Profiles](#)

[4.3 Creating GPU Instances](#)

[4.4 Running CUDA Applications on Bare-Metal](#)

[4.4.1 GPU Instances](#)

[4.4.2 GPU Utilization Metrics](#)

[4.5 Destroying GPU Instances](#)

[4.6 Running CUDA Applications as Containers](#)

[4.6.1 Install Docker](#)

[4.6.2 Install Denglin Container Toolkit](#)

[4.6.3 Running Containers](#)

[4.6.4 MIG Support in Kubernetes](#)

[5 Device Nodes and Capabilities](#)

[5.1 System Level Interface](#)

[5.2 denglin-capabilities](#)

1 Overview

1.1 Introduction

The Multi-instance GPU (MIG) feature allows GPUs to be securely partitioned into up to 4 separate GPU Instances for CUDA applications, providing multiple users with separate GPU resources for optimal GPU utilization.

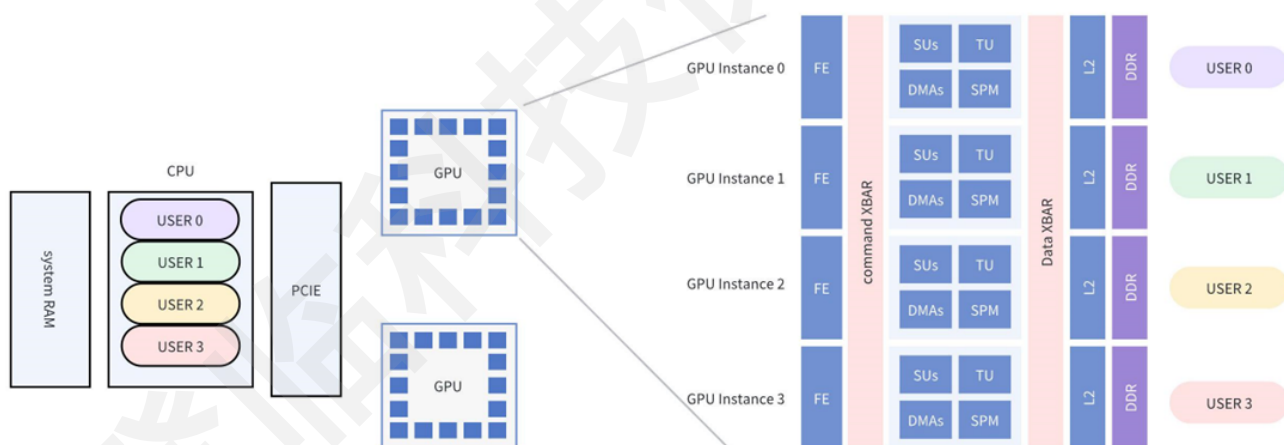
This feature is particularly beneficial for workloads that do not fully saturate the GPU's compute capacity and therefore users may want to run different workloads in parallel to maximize utilization.

With MIG, each instance's processors have separate and isolated paths through the entire memory system -- the on-chip crossbar ports, L2 cache banks, memory controllers, and DRAM address busses are all assigned uniquely to an individual instance. This ensures that an individual user's workload can run with predictable throughput and latency, with the same L2 cache allocation and DRAM bandwidth, even if other tasks are thrashing their own caches or saturating their DRAM interfaces.

MIG can partition available GPU compute resources (including Streaming-Units, Tensor-Units, and GPU engines such as DMA engines or encoders, decoders), to provide a defined quality of service (QoS). MIG enables multiple GPU Instances to run in parallel on a single, physical Denglin GPU.

MIG supports the following deployment configurations:

- Bare-metal, including containers.
- GPU pass-through virtualization to Linux guests on top of supported hypervisors.
- GPU KVM virtualization to Linux guests on top of supported hypervisors.



The purpose of this document is to introduce the concepts behind MIG, deployment considerations and provide examples of MIG management to demonstrate how users can run CUDA applications on MIG supported GPUs.

1.2 Concepts

1.2.1 Terminology

This section introduces some terminology used to describe the concepts behind MIG.

GPU Context

A GPU context is analogous to a CPU process. It encapsulates all the resources necessary to execute operations on the GPU, including a distinct address space, memory allocations, etc. A GPU context has the following properties:

- Fault isolation
- Individually scheduled
- Distinct address space

GPU Engine

A GPU engine is what executes work on the GPU.

Streaming-unit

It is the most commonly used engine for Compute/Graphics engine, which executes the compute instructions.

SU is used for abbreviation.

Tensor-unit

Tensor-Unit executes the fast tensor operations, such as MAD, pooling, activation, etc.

TU is used for abbreviation.

DMA

Copy engine is for memory copy and fill.

Multi-media Pack

DEC for video decoding, ENC for video encoding, JPEG for jpeg decoding and coding. DEC, ENC and JPEGs are grouped in multi-media packs.

In DLIv2, 1 multi-media pack includes 2 DECs, 1 ENC and 2 JPEGs.

ME is used for abbreviation.

GPU Cluster

A GPU cluster is the smallest fraction of the GPU that combines a single GPU memory and other compute resources.

GPU Memory Cluster

A GPU memory cluster is the smallest fraction of the GPU's memory, it's same as gpu cluster, but particularly refers to the DDR in a GPU cluster.

GPU Instance

A GPU Instance (GI) is a combination of GPU clusters. Anything within a GPU instance always shares all the GPU memory and other GPU engines, but it's GPU engines can be further subdivided into compute instances (CI). A GPU instance provides memory QoS. Each GPU cluster includes dedicated GPU memory resources which limit both the available capacity and bandwidth, and provide memory QoS.

1.2.2 Partitioning

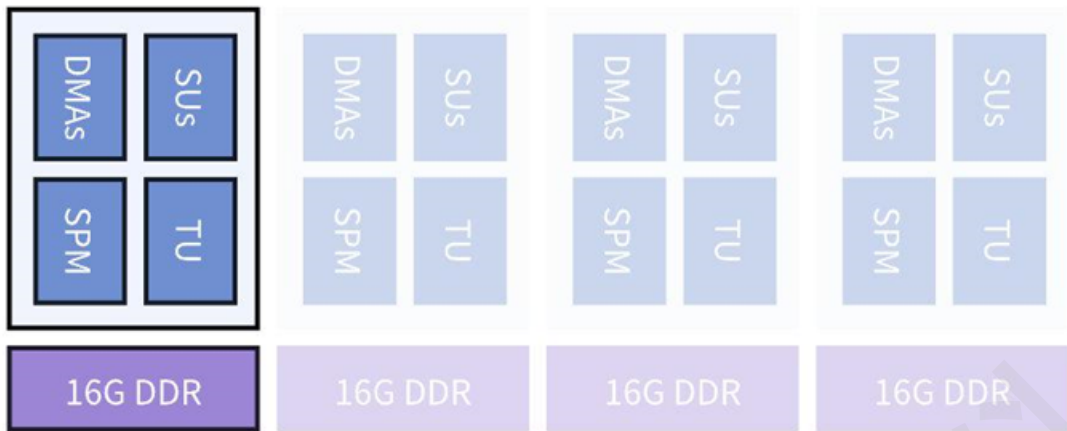
This section provides an overview of how the user can create various partitions on the GPU.

1.2.2.1 GPU Instance

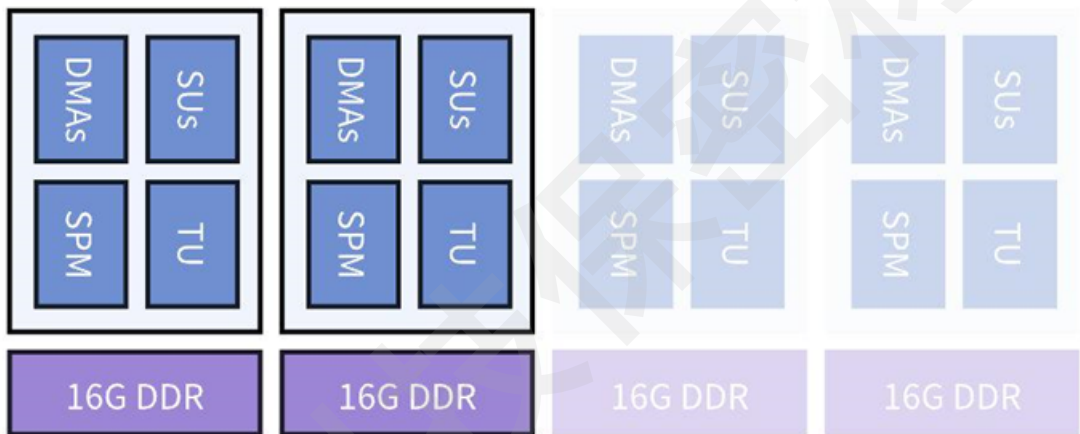
Partitioning of the GPU happens using clusters, so the Goldwasser II-XL GS40 (64GB) GPU can be split to 4 clusters shown in the diagram below:



As explained above, then to create a GPU Instance (GI) requires combining some number of cluster. In the diagram below, 1 cluster is used to create a 1c.1t GI profile:



Similarly, it can be combined 2 clusters to create the 2c.2t GI profile:



1.2.2.2 Profile Placement

The number of clusters that a GI can be created with is not arbitrary. Denglin driver APIs provide a number of "GPU Instance Profiles" and users can create GIs by specifying one of these profiles.

On a given GPU, multiple GIs can be created from a mix and match of these profiles, so long as enough slices are available to satisfy the request.

For a list of all supported combinations of profiles on MIG-enabled GPUs, refer to the section on [supported profiles](#).

Note that, as GPU Instances are created and destroyed at different locations, fragmentation can occur, and the physical position of one GPU Instance will play a role in which other GPU Instances can be instantiated next to it.

1.2.3 CUDA Concurrency Mechanisms

MIG has been designed to be largely transparent to CUDA applications - so that the CUDA programming model remains unchanged to minimize programming effort. CUDA already exposes multiple technologies for running work in parallel on the GPU and it is worthwhile showcasing how these technologies compare to MIG. Note that streams and MPS are part of the CUDA programming model and thus work when used with GPU Instances.

CUDA Streams are a CUDA Programming model feature where, in a CUDA application, different work can be submitted to independent queues and be processed independently by the GPU. CUDA streams can only be used within a single process and don't offer much isolation -- the address space is shared, the SUs are shared, the GPU memory bandwidth, caches and capacity are shared. And lastly any errors affect all the streams and the whole process.

Lastly, MIG is the new form of concurrency offered by Denglin GPUs while addressing some of the limitations with the other CUDA technologies for running parallel work.

Table 1-1 CUDA Concurrency Mechanism

Type	Streams	MIG
Partition Type	Single Process	Physical
Max Partitions	Unlimited	4
Compute resource Isolation	No	YES
Memory Protection	No	YES
Memory Bandwidth QoS	No	YES
Error Isolation	No	YES but Limited
Cross-Partition Inter-op	Always	IPC or by host memory in single process
Re-configure	Dynamic	When GI is idle

2 MIG Device Names

By default, a MIG device consists of a single “GPU Instance” (and a single “Compute Instance” if it is supported). The table below highlights a naming convention to refer to a MIG device by its GPU Instance's cluster number and its Tensor-Unit number.

Cluster number in the GPU instance

MIG 2c . 1t

Tensor-Unit number in the GPU instance

And there could be some suffixes such as `.6s` and `.2me` mean 6 SUs and 2 multi-media packs, respectively. For example, Full profile of Goldwasser II-L GS30 is `4c.3t.6s.6me`.

Notice that if not explicitly specified, SUs and multi-media packs are fully alive:

For Denglin Hamming V2:

```
fully_alive_SU_number = cluster_number * 2
fully_alive_ME_number = cluster_number * 2
```

2.1 Device Enumeration

GPU Instances (GIs) and Compute Instances (CIs) are enumerated in the new `/proc` filesystem layout for MIG:

```
$ ls -l /proc/driver/denglin-caps/
total 0
-r--r--r-- 1 root root 0 Aug 15 17:35 enabled-minors
-rw-r--r-- 1 root root 0 Aug 15 17:35 init-nodes
-r--r--r-- 1 root root 0 Aug 15 19:16 mig-minors
```

The corresponding device nodes (in mig-minors) are created under `/dev/denglin-caps`. Refer to the chapter on device nodes and capabilities for more information.

2.2 CUDA Device Enumeration

MIG supports running CUDA applications by specifying the CUDA device on which the application should be run. Enumeration of multiple MIG GPU instances of a same physical GPU is supported.

CUDA_VISIBLE_DEVICES has been extended to add support for MIG. Depending on the driver versions being used, two formats are supported:

1. GPU instance index starts from 0.

2. GPU UUID starts with `MIG-<UUID>`.

The example below shows how MIG devices are assigned GPU UUIDs in an 3-GPU system with each GPU configured differently.

```
$ dlsmi -L
GPU 0: Goldwasser II-L GS30 (UUID: GPU-ce3711b6-ec1e-49e5-9db8-1e17a439624e)
  MIG 2c.1t (profile: v2.2c.1t.128) Device 0: (UUID: MIG-acd445f6-b684-4b4c-beb2-4dc5953edcb9)
  MIG 2c.1t (profile: v2.2c.1t.128) Device 1: (UUID: MIG-f65dfdb2-5da4-4d81-bd4b-1a2bb210a6c6)

GPU 1: Goldwasser II-XL GS40 (UUID: GPU-69ec4511-8092-4681-a915-00b6dfaedc68)
  MIG 1c.1t (profile: v2.1c.1t.128) Device 0: (UUID: MIG-dd5ba980-defe-46d2-b301-e018b5595d35)
  MIG 1c.1t (profile: v2.1c.1t.128) Device 1: (UUID: MIG-0819ad9c-79d7-4e42-8110-e92d283ad68b)
  MIG 2c.2t (profile: v2.2c.2t.256) Device 2: (UUID: MIG-c4bb6b6e-44d3-4443-a24f-08ed66b80f4e)
```

3 Supported MIG Profiles

This table below provides an overview of the supported profiles on supported GPUs.

Table 3-1 Overview of the supported profiles on GS30 and GS40

product types		GS40				GS30	
Possible MIG profiles	4c.4t	1					
	4c.3t						
	4c.2t					1	
	2c.2t		2	1			
	2c.1t						2
	1c.1t			2	4		

Please note that, Denglin software disallows partitioning the GPU with remaining clusters without Tensor-Unit. For example, Disallow 1c.1t or 2c.2t profile for GS30 products.

3.1 GS40 Profiles

The following diagram shows the profile placements supported on the Goldwasser II-XL GS40 product:

Table 3-2 Profile placements supported on the GS40

Config	Cluster 0	Cluster 1	Cluster 2	Cluster 3	TU	ME
1	4				4	8
2	2		1	1	2 + 1 + 1	4 + 2 + 2
3	1	1	2		1 + 1 + 2	2 + 2 + 4
4	1	2		1	1 + 2 + 1	2 + 4 + 2
5	1	1	1	1	1 + 1 + 1 + 1	2 + 2 + 2 + 2

The table below shows the supported profiles on the Goldwasser II-XL GS40 product.

Table 3-3 Supported profiles on GS40

Profile Name	Clusters	TU	SU	ME	L2 Cache Size	DMA Engines	Number of Instances Available
MIG 1c.1t	1	1	1/4	2	1/4	3	4

Profile Name	Clusters	TU	SU	ME	L2 Cache Size	DMA Engines	Number of Instances Available
MIG 2c.2t	2	2	2/4	4	2/4	6	2
MIG 4c.4t	4	4	4/4	8	4/4	12	1

3.2 GS30 Profiles

The following diagram shows the profile placements supported on the Goldwasser II-XL GS30 products:

Table 3-4 Profile placements supported on GS30 (1)

Config	Cluster 0	Cluster 1	Cluster 2	Cluster 3	TU	ME	Total ME
	Either cluster without TU						
1	4				2	6	6
2	2		2		1 + 1	2~4 + 2~4	

Table 3-5 Profile placements supported on GS30 (2)

Config	Cluster 0	Cluster 1	Cluster 2	Cluster 3	TU	ME	Total ME
	without TU						
1	4				2	6	
2	1 of 2		2 of 2		1 + 1	2~4 + 2~4	6
		1 of 2		2 of 2			

Note that, in config 2 of above GS30 product, cluster 0 and cluster 2 are combined to a GPU instance with profile `2c.1t`, while cluster 1 and cluster 3 are combined to another GPU instance of profile `2c.1t`.

Table 3-6 Profile placements supported on GS30 (3)

Config	Cluster 0	Cluster 1	Cluster 2	Cluster 3	TU	ME	Total ME
			without TU				
1	4				2	6	6
2	1 of 2		2 of 2		1 + 1	2~4 + 2~4	
		1 of 2		2 of 2			

The table below shows the supported profiles on the GS30 products.

Table 3-7 Supported profiles on GS30

Profile Name	Clusters	TU	Fraction of SUs	ME	L2 Cache Size	DMA Engines	Number of Instances Available
MIG 2c.1t	2	1	2/4	2~4	2/4	6	1
MIG 4c.2t	4	2	4/4	2~4	4/4	12	1

Full profile of GS30 is 4c.3t.6s.6me.

There could be .xs and .yme suffix (where x, y are numbers) which are not listed above. Those suffixes may exist or not exist --- even when any exists, x or y value will also differ --- according to practical products, as multi-media packs resides differently.

Example practical profiles such as MIG 1c.1t.1s.1me, MIG 2c.1t.2s.3me, etc.

3.3 UL Profiles

UL products only includes 1 cluster, hence it's still full profile when MIG mode enabled.

Table 3-8 UL Profiles

Profile Name	Clusters	TU	Fraction of SUs	ME	L2 Cache Size	DMA engines	Number of Instances Available
MIG 1c.1t	1/1	1	1/1	2	1/1	3	1

4 Getting Started with MIG

4.1 Enable MIG Mode

By default, MIG mode is not enabled on the GPU. For example, run the `dlsmi` tool shows that MIG mode is disabled:

```
$ dlsmi -i 0
```

DL-SMI 11.0										Driver version 0			
GPU Product-Type				Bus-Id	Cluster-Memory-Usage				GPU-Util				
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	0	1	2	3	MIG M.				
0	N/A	38C	P2	1w / 70w	2 / 32768	0	0	0	0	0%	Disabled		

MIG mode can be enabled on a per-GPU basis with the following command:

```
dlsmi -i <GPU IDs> -mig on
```

The GPUs can be selected using comma separated GPU indexes, PCI Bus Ids or UUIDs. If no GPU ID is specified, then MIG mode is applied to all the GPUs on the system.

```
$ sudo dlsmi -i 0 -mig on
Enabled MIG Mode for GPU 00000000:01:00.0
All done.
```

```
$ dlsmi -i 0
```

DL-SMI 11.0										Driver version 0.71.0			
GPU Product-Type				Bus-Id	Cluster-Memory-Usage				GPU-Util				
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	0	1	2	3	MIG M.				
0	N/A	38C	P2	1w / 70w	0 / 32768	0	0	0	0	0%	Enabled		

4.2 List GPU Instance Profiles

Denglin driver provides a few profiles that users can be used for options when configuring the MIG feature in Goldwasser products. The profiles are the sizes and capabilities of the GPU instances that can be created by the user. The driver also provides information about the placements, which indicate the type and number of instances that can be created.

```
$ dlsmi -L
GPU 0: Goldwasser II-L GS30 (UUID: GPU-61ce5095-e3f5-43f0-a55b-5ab225aafa4b)
GPU 1: Goldwasser II-XL GS40 (UUID: GPU-6ba67f8f-25b0-437d-ba29-8b76c9e06216)
$ dlsmi mig -lgip
```

GPU	Profile Name	Profile ID	Instances Free/Total	Memory GiB	SU	TU	ENC	DEC	JPG
0	MIG 2c.1t.3s	18	2/2	16.00	3	1	0	0	0
0	MIG 4c.2t.6s	36	1/1	32.00	6	2	0	0	0
1	MIG 1c.1t	17	4/4	8.00	2	1	0	0	0
1	MIG 2c.2t	34	2/2	16.00	4	2	0	0	0
1	MIG 4c.4t	68	1/1	32.00	8	4	0	0	0

List the possible placements available using the following command. The syntax of the placement is `{<index>}:<GPU Cluster Count>` and shows the placement of the instances on the GPU.

```
$ dlsmi mig -lgipp
GPU 0 Profile ID 18 Placements: {0,1}:2
GPU 0 Profile ID 36 Placements: {0}:4
GPU 1 profile ID 17 Placements: {0,1,2,3}:1
GPU 1 Profile ID 34 Placements: {0,2}:2
GPU 1 Profile ID 68 Placements: {0}:4
```

Take GPU 1 (Goldwasser II-XL GS40) for example, the command shows that the user can create 2 instances of type 2c.2t (profile ID 34) or 4 instances of 1c.1t (profile ID 17).

4.3 Creating GPU Instances

Before starting to use MIG, the user needs to create GPU instances using the `-cgi` option. One of three options can be used to specify the instance profiles to be created:

1. Profile ID (e.g. 9, 14, 5)
2. Short name of the profile (e.g. 2c.2t)
3. Full profile name of the instance (e.g. MIG 2c.2t)

Note:

Without creating GPU instances (and corresponding compute instances), CUDA workloads *cannot* be run on the GPU. In other words, simply enabling MIG mode on the GPU is not sufficient. Also note that, the created MIG devices are not persistent across system reboots. Thus, the user or system administrator needs to recreate the desired MIG configurations if the GPU or system is reset.

The following example shows how the user can create GPU instances. In this example, the user can create two GPU instances (of type 2c.2t), with each GPU instance having half of the available compute and memory capacity.

In this example, we purposefully use profile ID and short profile name to showcase how either option can be used:

```
$ sudo dlsmi -i 2 mig -cgi 34,2c.2t
Successfully created GPU instance ID 0 on GPU 2 using profile ID 34
Successfully created GPU instance ID 2 on GPU 2 using profile ID 34
```

Now list the available GPU instances:

```
$ dlsmi -i 2 mig -lgi
```

GPU instances:										
GPU	Profile Name	Profile ID	Instance ID	Memory GiB	SU	TU	ENC	DEC	JPG	Placement Start:Size
2	MIG 2c.2t	34	0	16.00	4	2	0	0	0	0:2
2	MIG 2c.2t	34	2	16.00	4	2	0	0	0	2:2

Now verify that the GIs are created:

```
$ dlsmi -i 2
```

DL-SMI 11.0 Driver version 0.71.0										
GPU	Product-Type	Bus-Id	Cluster-Memory-Usage	GPU-Util						
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	0	1	2	3	MIG	M.
0	Goldwasser	II-XL	GS40	0000:03:00.0	0%					
N/A	38C	P2	1w / 70W	4 / 32768	1	1	1	1	Enabled	

MIG devices:										
GPU	GI	MIG	SU	TU	ENC	DEC	JPG	Memory-Usage		
ID	Dev									
0	0	0	4	2	0	0	0	2 / 16384		
0	2	1	4	2	0	0	0	2 / 16384		

4.4 Running CUDA Applications on Bare-Metal

4.4.1 GPU Instances

The following example shows how two CUDA applications can be run in parallel on two different GPU instances. In this example, the `d1_dma_bench` CUDA sample is run simultaneously on the two GIs created on the GS40.

```
$ dlsmi -L
GPU 0: Goldwasser II-L GS30 (UUID: GPU-61ce5095-e3f5-43f0-a55b-5ab225aafa4b)
GPU 1: Goldwasser II-XL GS40 (UUID: GPU-6ba67f8f-25b0-437d-ba29-8b76c9e06216)
  MIG 2c.2t (profile: v2.2c.2t.256)   Device 0: (UUID: MIG-70a693a2-21de-4dd2-9b9e-63803c3c55c9)
  MIG 2c.2t (profile: v2.2c.2t.256)   Device 1: (UUID: MIG-9542c699-8fe2-4b5d-82d0-83a5e606f703)

CUDA_VISIBLE_DEVICES=MIG-70a693a2-21de-4dd2-9b9e-63803c3c55c9 d1_dma_bench --window=2g &
CUDA_VISIBLE_DEVICES=MIG-9542c699-8fe2-4b5d-82d0-83a5e606f703 d1_dma_bench --window=2g &
```

Now verify the two CUDA applications are running on two separate GPU instances:

```
$ dlsmi
```

DL-SMI 11.0												Driver version 0.71.0							
GPU Product-Type												Bus-Id		Cluster-Memory-Usage				GPU-Util	
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage		0	1	2	3	MIG	M.								
=====																			
0	N/A	38C	P2	1w	70w	0000:01:00.0	0	0	0	0	0%								
Goldwasser II-L GS30												0	0	0	0	Enabled			
=====																			
2	N/A	38C	P2	10w	120w	0000:03:00.0	1025	1025	1025	1025	100%								
Goldwasser II-XL GS40												1025	1025	1025	1025	Enabled			
=====																			
MIG devices:																			
GPU	GI	MIG	SU	TU	ENC	DEC	JPG	Memory-Usage											
ID	Dev																		
=====																			
0	0	0	4	2	0	0	0	2050 / 16384											
=====																			
0	2	1	4	2	0	0	0	2050 / 16384											
=====																			
Processes								Cluster-Memory-Usage				GPU-Memory							
GPU	GI	PID	TASK	Process name				0	1	2	3	Usage							
=====																			
2	0	3299	9	d1_dma_bench															

						1024	1024	0	0	2048	
+	-----	+									
	2	2	3300	10	d1_dma_bench						
						0	0	1024	1024	2048	
+	-----	+									

4.4.2 GPU Utilization Metrics

DLML (and dlsmi) supports attribution of utilization metrics to MIG devices. When MIG enabled, dlsmi displays GPU utilization values with respect to the whole GPU. That means, if 1 of the 4 GPU instances with 1c.1t profile is fully busy on a GS40 product, GPU utilization should be 25%.

4.5 Destroying GPU Instances

Once the GPU is in MIG mode, GIs can be configured dynamically. The following example shows how the GIs created in the previous examples can be destroyed.

If the intention is to destroy all the GIs, then this can be accomplished with the following commands:

```
$ sudo dlsmi mig -dgi
Successfully destroy GPU instance ID 0 on GPU 0
Successfully destroy GPU instance ID 2 on GPU 0
Successfully destroy GPU instance ID 0 on GPU 1
Successfully destroy GPU instance ID 2 on GPU 1
Successfully destroy GPU instance ID 3 on GPU 1
Successfully destroy GPU instance ID 0 on GPU 2
Successfully destroy GPU instance ID 2 on GPU 2
```

In this example, we delete the specific GI 1 in GPU 1.

```
$ dlsmi -i 1 mig -lgi
+-----+
| GPU instances:                                     |
| GPU Profile Profile Instance Memory SU TU ENC DEC JPG Placement |
| Name          ID      ID      GiB           Start:Size |
|=====|
| 1 MIG 1c.1t   17      2      8.00    2  1  0  0  0    2:1 |
+-----+
| 1 MIG 1c.1t   17      3      8.00    2  1  0  0  0    1:1 |
+-----+
| 1 MIG 2c.1t   18      0     16.00    4  1  0  0  0    0:2 |
+-----+

$ sudo dlsmi -i 1 mig -dgi -gi 1
Successfully destroy GPU instance ID 1 on GPU 1
```

4.6 Running CUDA Applications as Containers

Denglin container toolkit has been enhanced to provide support for MIG devices, allowing users to run GPU containers with runtimes such as Docker. This section provides an overview of running Docker containers with MIG.

4.6.1 Install Docker

Many Linux distributions may come with Docker-CE pre-installed. If not, use the Docker installation script to install Docker.

```
$ curl https://get.docker.com | sh \
&& sudo systemctl start docker \
&& sudo systemctl enable docker
```

4.6.2 Install Denglin Container Toolkit

```
tar -xf denglin-hamming_ss-<version>-<platform>-container.tar
sudo sh denglin-hamming_ss-<version>-<platform>-container.run [--target <extract_path>]
sudo kill -SIGHUP dockerd
```

If `<extract_path>` is not specified, it will extract to `denglin-hamming_ss-<version>-<platform>-container` in the current path.

4.6.3 Running Containers

To run containers on specific MIG devices, the `DENGLIN_DEVICES` variable can be used.

`DENGLIN_DEVICES` supports the following formats to specify MIG devices:

1. `MIG-<UUID>/<GPU instance ID>`
2. `<GPUDeviceIndex>:<MIGDeviceIndex>`

If `dlrt` is not the default container runtime, then `--runtime=dlrt` should be added to docker run command.

The following example shows running `dlsmi` from within a container using both formats. As can be seen in the example, only one MIG device as chosen is visible to the container when using either format.

```
# running on host, all mig devices can be seen
$ dlsmi
```

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+										
DL-SMI 11.0				Driver version 0.71.0						
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+										
GPU Product-Type				Bus-Id	Cluster-Memory-Usage			GPU-Util		
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	0	1	2	3	MIG	M.
=====										
0			Goldwasser II-XL GS40	0000:01:00.0						0%
0	39C	P0	28W / 120W	344 / 32768	86	86	86	86		Enabled

+-----+									
+-----+									
MIG devices:									
+-----+									
GPU	GI	MIG	SU	TU	ENC	DEC	JPG	Memory-Usage	
ID	Dev								
=====									
0	0	0	2	1	0	0	0	2 / 8192	
+-----+									
0	1	1	2	1	0	0	0	2 / 8192	
+-----+									
0	2	2	2	1	0	0	0	2 / 8192	
+-----+									
0	3	3	2	1	0	0	0	2 / 8192	
+-----+									
+-----+									
+-----+									
Processes					Cluster-Memory-Usage			GPU-Memory	
GPU	GI	PID	TASK	Process name	0	1	2	3	Usage
=====									
No running processes found									
+-----+									

```
$ dlsmi -L
GPU 0: Goldwasser II-XL GS40 (UUID: GPU-3bf391d1-ac00-4073-b199-21c49719881c)
  MIG 1c.1t (profile: v2.1c.1t.128) Device 0: (UUID: MIG-4ab241cd-3d7a-4134-972c-72f2e6e45b09)
  MIG 1c.1t (profile: v2.1c.1t.128) Device 1: (UUID: MIG-4d7e115a-2d69-4a03-9602-c1a3b0bf0f37)
  MIG 1c.1t (profile: v2.1c.1t.128) Device 2: (UUID: MIG-3711a6e4-b333-4e85-9238-a6f5f5d71365)
  MIG 1c.1t (profile: v2.1c.1t.128) Device 3: (UUID: MIG-3132ef67-2071-4568-8e0e-c294262c31cb)

# MIG-GPU-<UUID>/<GPU instance ID>
$ docker run -it --rm --runtime=dlsrt -e DENGlin_DEVICES=MIG-GPU-3bf391d1-ac00-4073-b199-21c49719881c/0 ubuntu:16.04 dlsmi -L
GPU 0: Goldwasser II-XL GS40 (UUID: GPU-3bf391d1-ac00-4073-b199-21c49719881c)
  MIG 1c.1t (profile: v2.1c.1t.128) Device 0: (UUID: MIG-4ab241cd-3d7a-4134-972c-72f2e6e45b09)

# <GPUDeviceIndex>:<MIGDeviceIndex>
$ docker run -it --rm --runtime=dlsrt -e DENGlin_DEVICES=0:0 ubuntu:16.04 dlsmi -L
GPU 0: Goldwasser II-XL GS40 (UUID: GPU-3bf391d1-ac00-4073-b199-21c49719881c)
  MIG 1c.1t (profile: v2.1c.1t.128) Device 0: (UUID: MIG-4ab241cd-3d7a-4134-972c-72f2e6e45b09)
```

4.6.4 MIG Support in Kubernetes

You can proceed to deploy `denglin-device-plugin` in your cluster, so that Kubernetes can schedule pods on the available MIG devices.

```
tar -xf denglin-hamming_ss-<version>-<platform>-k8s.tar
sh denglin-hamming_ss-<version>-<platform>-images.run [--target <extract_path>]
sh denglin-hamming_ss-<version>-<platform>-k8s.run [--target <extract_path>]
cd <extract_path>/denglin-device-plugin
kubectl apply -f denglin_device_plugin.yaml
```

MIG resource will be available on the node in this format `denglinai.com/mig-Xg.Ygb`, X is the cluster number, Y is the memory size of the MIG resource.

A pod consumes 2 MIG resource can be defined as follow:

```
apiVersion: v1
kind: Pod
metadata:
  name: denglin-gpu-pod
spec:
  restartPolicy: OnFailure
  containers:
    - name: denglin-gpu-pod
      image: ubuntu:18.04
      resources:
        limits:
          denglinai.com/mig-Xg.Ygb: 2
```

5 Device Nodes and Capabilities

Currently, Denglin kernel driver exposes its interfaces through a few system-wide device nodes. Each physical GPU is represented by its own device node -- e.g. `denglin0`, `denglin1` etc. This is shown below for a 3-GPU system.

```
/dev
├─ denglin0
├─ denglin0-jpeg
├─ denglin0-vid
├─ denglin1
├─ denglin1-jpeg
├─ denglin1-vid
├─ denglin2
├─ denglin2-jpeg
├─ denglin2-vid
├─ denglin-caps/
├─ denglin-ctrl
└─ denglin-uvm
```

`denglin-capabilities` has been introduced in Denglin driver. The idea being that access to a specific `capability` is required to perform certain actions through the driver. If a user has access to the `capability`, the action will be carried out. If a user does not have access to the `capability`, the action will fail.

For example, the `mig-config` capability allows one to create and destroy MIG instances on any MIG-capable GPU. Without this capability, all attempts to create or destroy a MIG instance will fail.

The following sections walk through the system level interface for managing these new `denglin-capabilities`, including the steps necessary to grant and revoke access to them.

5.1 System Level Interface

The level interface is `/dev` based to work with `denglin-capabilities`: The `/dev` based interface relies on `cgroups` and `mknod` permissions. Technically, the `/dev` based interface also relies on user-permissions as a second-level access control mechanism (on the actual device node files themselves), but the primary access control mechanism is `cgroups`.

5.2 denglin-capabilities

The system level interface for interacting with `/dev` based capabilities is actually through a combination of `/proc` and `/dev`.

First, a new major device is now associated with `denglin-caps` and can be read from the standard `/proc/devices` file.

```
$ cat /proc/devices | grep denglin-caps
247 denglin-caps
```

Second, the exact same set of files exist under `/proc/driver/denglin-caps`. These files are not for control access to the capability directly but instead, the contents of these files point at a device node under `/dev`, through which `cgroups` can be used to control access to the capability.

This can be seen in the example below:

```
$ cat /proc/driver/denglin-caps/mig-config
DeviceFileMinor: 128
DeviceFileMode: 0644
DeviceFileModify: 1
```

The combination of the device major for `denglin-caps` and the value of `DeviceFileMinor` in this file indicate that the `mig-config` capability (which allows a user to create and destroy MIG devices) is controlled by the device node with a `major:minor` of 247:128. As such, one will need to use `cgroups` to grant a process read access to this device in order to configure MIG devices.

The standard location for these device nodes is under `/dev/denglin-caps` as seen in the example below:

```
$ ls -l /dev/denglin-caps/
total 0
crw-rw-rw- 1 root root 247, 4 8月 15 20:24 gpu1-gi0-ci0
crw-rw-rw- 1 root root 247, 6 8月 15 20:24 gpu1-gi2-ci0
crw-rw-rw- 1 root root 247, 128 8月 15 19:28 mig-config
crw-rw-rw- 1 root root 247, 129 8月 15 19:28 mig-monitor
```

These device nodes cannot be automatically created/deleted by Denglin driver at the same time it creates/deletes files underneath `/proc/driver/denglin-caps` due to GPL compliance issues, and to support defer insert kernel module after container starts.

Instead, a user-level program called `d1-modprobe` is provided, which can be invoked from user-space in order to create required device nodes inside `/proc` with proper permissions.

For example:

```
# Simulate a clean environment, without denglin caps nodes
$ sudo rm -rf /dev/denglin*

$ d1-modprobe

$ ls -l /dev/denglin*
crw-rw-rw- 1 root root 248, 0 8月 24 17:13 /dev/denglin0
crw-rw-rw- 1 root root 245, 0 8月 24 17:13 /dev/denglin0-jpeg
crw-rw-rw- 1 root root 246, 0 8月 24 17:13 /dev/denglin0-vid
crw-rw-rw- 1 root root 248, 1 8月 24 17:13 /dev/denglin1
crw-rw-rw- 1 root root 245, 1 8月 24 17:13 /dev/denglin1-jpeg
```



```
crw-rw-rw- 1 root root 246, 1 8月 24 17:13 /dev/denglin1-vid
crw-rw-rw- 1 root root 248, 2 8月 24 17:13 /dev/denglin2
crw-rw-rw- 1 root root 245, 2 8月 24 17:13 /dev/denglin2-jpeg
crw-rw-rw- 1 root root 246, 2 8月 24 17:13 /dev/denglin2-vid
crw-rw-rw- 1 root root 244, 0 8月 24 17:13 /dev/denglin-ctrl
crw-rw-rw- 1 root root 242, 0 8月 24 17:13 /dev/denglin-shm
crw-rw-rw- 1 root root 243, 0 8月 24 17:13 /dev/denglin-uvmm
```

/dev/denglin-caps:

total 0

```
crw-rw-rw- 1 root root 247, 0 8月 24 17:13 gpu0-gi0-ci0
crw-rw-rw- 1 root root 247, 4 8月 24 17:13 gpu1-gi0-ci0
crw-rw-rw- 1 root root 247, 8 8月 24 17:13 gpu2-gi0-ci0
cr--r--r-- 1 root root 247, 128 8月 24 17:13 mig-config
crw-rw-rw- 1 root root 247, 129 8月 24 17:13 mig-monitor
```

`dl-modprobe` looks at the `DeviceFileMode` filed in the capability file and creates the `mig-config` device node with the permissions indicated.

Programs such as `dl-smi` (and denglin underlying driver libraries) will automatically invoke `dl-modprobe` (when available) to create these device nodes on your behalf. In other scenarios it is not necessarily required to use `dl-modprobe` to create these device nodes, but it does make the process simpler.

If you actually want to prevent `dl-modprobe` from modify modes for `mig-config` device node, you can do the following:

```
# Update the file with a "DeviceFileModify" setting of 0
$ echo "DeviceFileModify: 0" > /proc/driver/denglin-caps/mig-config
```

You will then be responsible for managing modes of `mig-config` yourself. If you want to change that in the future, simply reset it to a value of `"DeviceFileModify: 1"` with the same command sequence.

One final thing to note is that the minor numbers for all possible capabilities are predetermined and can be queried under various files of the form:

```
$ cat /proc/driver/denglin-caps/mig-minors
gpu0-gi0-ci0 0
gpu0-gi1-ci0 1
gpu0-gi2-ci0 2
gpu0-gi3-ci0 3
gpu1-gi0-ci0 4
gpu1-gi1-ci0 5
gpu1-gi2-ci0 6
gpu1-gi3-ci0 7
...
gpu31-gi3-ci0 127
```