IMPROVE GPU UTILIZATION FROM SYSTEM LEVEL

Click Cheng, NVIDIA Solution Architect
GTC China 2020
WHAT’S ABOUT THE TALK

Welcome

It’s
From system level of NVIDIA perspective, proposed several ways to improve GPU utilization;
Discuss several GPU monitoring metrics which reflect real GPU utilization;
Intro each solution mechanism, usage, discuss the benefit in some test cases;
Summary different solution positioning, comparison, etc;

It’s Not
Improve GPU utilization from scheduler level;
Optimize GPU utilization from coding level;
Overview
What’s About The Talk
GPU Utilization Discussion

Multi-Process Service
MPS Intro, Usage, Test Cases

Multi-Instance GPU
MIG Intro, Usage, Test Cases

Triton and vGPU Brief
Intro, Test Cases

Quick Summary
OVERVIEW
Why Is This Important

GPU is more and more powerful, and more precious.

Many applications are benefiting more from more powerful GPU.

While for some lower-utilized application, still can’t fully utilize GPU powerful computing capability.

Example, some developing scenario, inference scenario.

Especially for some inference cases with critical latency limitation, which not allowed batching for inference.

How to share and isolate among processes or users on one GPU?
GPU UTILIZATION
Metrics and Tools

GPU utilization: reflect how busy different resources on GPU are, metrics including GPU core (CUDA core, integer, FP32, Tensor Core), frame buffer (capacity, bandwidth), PCIe RX and TX, NVLink RX and TX, encoder and decoder, etc.

Generally, when we talk about GPU utilization, we are mostly talking about GPU utilization of CUDA core.

GPU utilization reflects an impact on delivered application performance somehow, but not necessarily.

Monitor tools

nvidia-smi or NVML, installed with GPU driver;

DCGM: Data Center GPU Manager, standalone package, using NVML and advanced data center profiling metrics;
“GPU Utilization” from nvidia-smi or NVML is a rough metric that reflects how busy GPU cores are utilized.

Defined by “Percent of time over the past sample period during which one or more kernels was executing on the GPU”, from NVML API Guide.

Extreme case, the metric is 100% even there’s only one thread launched to run kernel on GPU during past sample period.
GPU UTILIZATION METRIC
From DCGM

DCGM provides CLI dcgmi and API for C and Python language.

DCGM DCP (Data Center Profiling) provides lower level profiling metrics, which lists several utilization metrics in more accurate.

From these metrics, better reflect how well GPU resources are utilized to some extent.

Well, one GPU has many different resources (computing, memory, IO), it’s highly recommended to capture several metrics to understand GPU utilization, not just one or two.
## GPU Utilization Metric

**DCGM DCP Metrics**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Definition</th>
<th>DCGM Field ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphics Engine Activity</td>
<td>Ratio of time the graphics engine is active. The graphics engine is active if a graphics/compute context is bound and the graphics pipe or compute pipe is busy.</td>
<td>DCGM_FI_PROF_GR_ENGINE_ACTIVE</td>
</tr>
<tr>
<td>SM Activity</td>
<td>The ratio of cycles an SM has at least 1 warp assigned (computed from the number of cycles and elapsed cycles)</td>
<td>DCGM_FI_PROF_SM_ACTIVE</td>
</tr>
<tr>
<td>SM Occupancy</td>
<td>The ratio of number of warps resident on an SM. (number of resident as a percentage of the theoretical maximum number of warps per elapsed cycle)</td>
<td>DCGM_FI_PROF_SM_OCCUPANCY</td>
</tr>
<tr>
<td>Tensor Utilization</td>
<td>The ratio of cycles the tensor (HMMA) pipe is active (off the peak sustained elapsed cycles)</td>
<td>DCGM_FI_PROF_PIPE_TENSOR_ACTIVE</td>
</tr>
<tr>
<td>Memory BW Utilization</td>
<td>The ratio of cycles the device memory interface is active sending or receiving data.</td>
<td>DCGM_FI_PROF_DRAM_ACTIVE</td>
</tr>
<tr>
<td>FLOP Counts</td>
<td>Ratio of cycles the fp64 /fp32 / fp16 / HMMA</td>
<td>IMMA pipes are active.</td>
</tr>
<tr>
<td>NVLink Utilization</td>
<td>The number of bytes of active NVLink rx or tx data including both header and payload.</td>
<td>DCGM_FI_DEV_NVLINK_BANDWIDTH_L0</td>
</tr>
<tr>
<td>PCIe Utilization</td>
<td>pci_bytes_{rx, tx} - The number of bytes of active PCIe rx or tx data including both header and payload.</td>
<td>DCGM_FI_PROF_PCIE_{T</td>
</tr>
</tbody>
</table>
GPU UTILIZATION METRIC

Using dcgm

Recommended monitor command with dcgm

$ dcgm dmon -e 1001,1002,1004,1005,1009,1010,1011,1012,150,155,110,111
HYPER QUEUE
Behind MPS

Hyper-Q is introduced since Kepler GPU.

To enable multiple CPU threads or processes to launch work on a single GPU simultaneously.

Supported connection types:

- Multiple CUDA streams;
- Multiple CPU threads;
- Multiple CPU processes;

Hyper-Q whitepaper:
for (int i = 0 ; i < nstreams ; ++i)
{
    kernel_A<<<1,1,0,streams[i]>>>(&d_a[2*i], time_clocks);
    total_clocks += time_clocks;
    kernel_B<<<1,1,0,streams[i]>>>(&d_a[2*i+1], time_clocks);
    total_clocks += time_clocks;
}
MULTI-PROCESS SERVICE

What’s MPS

An alternative, binary-compatible implementation of the CUDA Application Programming Interface (API).

Based on GPU Hyper-Q capability

- Enabling multiple CPU processes sharing one GPU context;
- Allowing kernels and memcpy in different processes can be executed simultaneously on the same GPU, to utilize GPU better;

MPS includes

- Control Daemon Process - The control daemon is responsible for starting and stopping the server, as well as coordinating connections between clients and servers.
- Server Process - The server is the clients’ shared connection to the GPU and provides concurrency between clients.
- Client Runtime - The MPS client runtime is built into the CUDA Driver library and may be used transparently by any CUDA application.
MULTI-PROCESS SERVICE
Without MPS VS With MPS

Without MPS

With MPS
System-wide provisioning with multiple users.

Client A from User 1 request;
Daemon create MPS server for User 1 and Client A runs;
Client B from User 1 request and assigned to MPS server, and to run;
Client C from User 2 request, and pending;
Util all clients from User 1 running end and MPS server exit for User 1, Daemon create MPS server for User 2, and Client C begin to run;
GPU Utilization
A single process may not utilize all the compute and memory-bandwidth capacity available on the GPU. MPS allows kernel and memcpy operations from different processes to overlap on the GPU, achieving higher utilization and shorter running times.

Reduced on-GPU Context Storage
The MPS server allocates one copy of GPU storage and scheduling resources shared by all its clients, thus reduces the resource storage.

Reduced on-GPU Context Switching
The MPS server shares one set of scheduling resources between all of its clients, eliminating the overhead of swapping when the GPU is scheduling between those clients.
Application process does not generate enough work to saturate the GPU. Applications like this are identified by having a small number of blocks-per-grid.

Application shows a low GPU occupancy because of a small number of threads-per-grid.

In strong-scaling case, some MPI processes may underutilize the available compute capacity. Especially for AI inference, with critical latency limitation, which not allowed batching for inference.
Volta MPS provides a few key improvements, compared with pre-Volta:

- Volta MPS clients submit work directly to the GPU without passing through the MPS server.
- Each Volta MPS client owns its own GPU address space instead of sharing GPU address space with all other MPS clients.
- Volta MPS supports limited execution resource provisioning for Quality of Service (QoS).
MULTI-PROCESS SERVICE

MPS Usage

Start MPS daemon process

```bash
nvidia-cuda-mps-control -d
```

Check MPS process

```bash
ps -ef | grep mps
```

Recommend to set compute mode to exclusive

```
sudo nvidia-smi -c EXCLUSIVE_PROCESS
```

Quit MPS daemon

```
echo quit | nvidia-cuda-mps-control
```
MULTI-PROCESS SERVICE

MPS Usage

nvidia-smi shows when running eight trtexec processes with MPS:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67  Driver Version:418.67  CUDA Version: 10.1               |
+-----------------------------------------------------------------------------+
| Name  Persistence-M| Bus-Id  Disp.A | Volatile Uncorr. ECC |
| Fan Temp Perf Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
+-----------------------------------------------------------------------------+
| 0 Tesla V100-32X... On | 00000000:06:00.0 Off | Off |
| N/A 46C P0 140W / 300W | 702MB / 16160MB | 100% Default |
+-----------------------------------------------------------------------------+

Processes: GPU PID Type  Process name    GPU Memory Usage
+-----------------------------------------------------------------------------+
| 0 91016 C  nvidia-cuda-mps-server  29MB |
| 0 91074 C  trtexec                 873MB |
| 0 91075 C  trtexec                 873MB |
| 0 91076 C  trtexec                 873MB |
| 0 91077 C  trtexec                 873MB |
| 0 91078 C  trtexec                 873MB |
| 0 91079 C  trtexec                 873MB |
| 0 91080 C  trtexec                 873MB |
| 0 91081 C  trtexec                 873MB |
+-----------------------------------------------------------------------------+
```
MPS TEST CASE 1
Simple Kernel with One Thread Running

Simple kernel code: (Ignore the computing content)

```c
__global__ void testMaxFlopsKernel(float * pData, int nRepeats, float v1, float v2)
{
    int tid = blockIdx.x* blockDim.x + threadIdx.x;
    float s = pData[tid], s2 = 10.0f - s, s3 = 9.0f - s, s4 = 9.0f - s2;
    for(int i = 0; i < nRepeats; i++)
    {
        s=v1-s*v2;
    }
    pData[tid] = ((s+s2)+(s3+s4));
}
```

To test: run four processes with and without MPS
To profile: profiling analysis the running characteristic
MPS TEST CASE 1

Test Results

Run multiple processes with mpirun, command like: `mpirun -np $NP ./testMPS`

<table>
<thead>
<tr>
<th>Category</th>
<th>Average Wall Clock Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 Process</td>
</tr>
<tr>
<td>MPS OFF</td>
<td>2924 ms</td>
</tr>
<tr>
<td>MPS ON</td>
<td>2924 ms</td>
</tr>
</tbody>
</table>

Without MPS, the kernel running time increases linearly along with the number of processes.

With MPS, the kernel run time of multi processes is almost the same as one process.

This is the extreme case, but it’s the best case to show MPS benefit.
MPS TEST CASE 1
Profiling Analysis

Use nvprof to capture trace:

```
node1:~$ nvprof -o ./profile-test2-%p --profile-child-processes mpirun -np 2 ./testMPS
==56763== NVPROF is profiling process 56763, command: ./testMPS
==56768== NVPROF is profiling process 56768, command: ./testMPS
...
Rank0: BlockSize(1, 1, 1), GridSize(1, 1, 1)
Rank0: Iteration: 1, Total Elapsed Time: 2918.924ms, Single kernel cost time: 2918.924ms
Rank0: Performance: 0.685GFLOPS
Rank1: BlockSize(1, 1, 1), GridSize(1, 1, 1)
Rank1: Iteration: 1, Total Elapsed Time: 2917.827ms, Single kernel cost time: 2917.827ms
Rank1: Performance: 0.685GFLOPS
...
==56768== Generated result file: /home/dgx/src/testMPS/profile-test2-56768
...
==56763== Generated result file: /home/dgx/src/testMPS/profile-test2-56763
```

Then import into NVVP profiler tool for visual profiling analysis.
Without MPS, four processes.

Four CUDA contexts on a V100 GPU. Although it seems like that they are running concurrently, the execution time for each kernel is lengthened. That is because that they are running under the GPU time slice rotation scheduling mechanism. These CUDA contexts need to be switched in each time slice which introduces extra time overhead.
MPS TEST CASE 1
Profiling Analysis: With MPS

With MPS, four processes.

Only one CUDA context to run these four processes.

The kernels from different processes are really running overlapped.
MPS TEST CASE 2
ResNet-50 Inference in 7ms Budget

This example is to run ResNet-50 inference with TensorRT engine.

We use NGC container “nvcr.io/nvidia/tensorrt:19.07-py3” on SXM2 V100 16GB.

We run and compare several scenarios in 7ms inference time budget:

- Batching in single process;
- No batching (batch size is 1) in multiple processes, without MPS;
- No batching (batch size is 1) in multiple processes, with MPS;
- Batching and multiple processes combination;

At the same time, we capture some utilization metrics with dcgmi, to quantify GPU usage.

dcgmi dmon -e 1001,1002,1004,1005,1009,1010,1011,1012
MPS TEST CASE 2
Steps to Test

Start container

nvidia-docker run -it --name click-trt --privileged -v /home/click/models:/click nvcr.io/nvidia/tensorrt:19.07-py3 bash

Build out ResNet-50 TRT engine (using caffemodel here)

## Example, for batch size 1, 32, ...


Test in single process

trtexec --loadEngine=/workspace/rn50-bs1.engine --iterations=1000 --workspace=1024 --fp16

trtexec --loadEngine=/workspace/rn50-bs32.engine --iterations=10000 --workspace=1024 --fp16 --batch=32

Test in multi processes with MPI

mpirun -np 8 --allow-run-as-root trtexec --loadEngine=/workspace/rn50-bs1.engine --iterations=1000 --workspace=1024 --fp16 > trt-mps-mpi-8.log
MPS TEST CASE 2

Test Results

Batching is the recommended way to reach best throughput.

Without batching, i.e. BS=1 cases, MPS can bring ~3X throughput.

Batching and MPS can be combined, to improve throughput to some extent.
MPS TEST CASE 2

GPU Utilization Metrics - MPS OFF

GPU Utilization Metrics - Without Batching, Without MPS

<table>
<thead>
<tr>
<th>NP</th>
<th>GPU Util</th>
<th>SM Activity</th>
<th>Tensor Util</th>
<th>SM Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0.331</td>
<td>0.035</td>
<td>0.042</td>
</tr>
<tr>
<td>2</td>
<td>0.88</td>
<td>0.333</td>
<td>0.039</td>
<td>0.043</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0.343</td>
<td>0.038</td>
<td>0.046</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>0.352</td>
<td>0.038</td>
<td>0.046</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>0.343</td>
<td>0.038</td>
<td>0.046</td>
</tr>
</tbody>
</table>
MPS TEST CASE 2

GPU Utilization Metrics - MPS ON

GPU Utilization Metrics - Without Batching, With MPS
MPS TEST CASE 2
Profiling Analysis

BS=1, NP=8, MPS OFF

BS=1, NP=8, MPS ON
MPS TEST CASE 3

JPEG Resize

JPEG to JPEG resizing is an essential workload for many internet services, including training and inference for image classification, object detection, etc.

And for some service provider, to cut storage expense, they might just storage one image instead of several dozens in different resolutions.

Fastvideo, an NVIDIA Preferred Partner, developed an image processing SDK with CUDA acceleration (one of their customer was Flickr), since there’re multi phases in the whole JPEG resize implementation pipeline, like copy from storage to CPU memory, then copy to GPU memory, JPEG decoding, resizing, sharp, JPEG encoding, copy to CPU memory, etc. They’ve done many optimizations across the whole pipeline, and one technical they adopted is NVIDIA MPS, to optimize the throughput of the GPU system.

We use Fastvideo SDK to perform this testing.
MPS TEST CASE 3

Test Results

Resize JPEG from 1920x1080 to 480x270.
Up to 3.5x throughput improvement when MPS enabled.

<table>
<thead>
<tr>
<th>Processes Number</th>
<th>FPS - MPS OFF</th>
<th>FPS - MPS ON</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1152</td>
<td>1633</td>
<td>1.42</td>
</tr>
<tr>
<td>4</td>
<td>1025</td>
<td>2319</td>
<td>2.26</td>
</tr>
<tr>
<td>6</td>
<td>1016</td>
<td>2786</td>
<td>2.74</td>
</tr>
<tr>
<td>8</td>
<td>1014</td>
<td>3024</td>
<td>2.98</td>
</tr>
<tr>
<td>10</td>
<td>1011</td>
<td>3190</td>
<td>3.15</td>
</tr>
<tr>
<td>12</td>
<td>1014</td>
<td>3301</td>
<td>3.25</td>
</tr>
<tr>
<td>14</td>
<td>1154</td>
<td>3367</td>
<td>2.92</td>
</tr>
<tr>
<td>16</td>
<td>1012</td>
<td>3458</td>
<td>3.42</td>
</tr>
<tr>
<td>18</td>
<td>1009</td>
<td>3558</td>
<td>3.53</td>
</tr>
</tbody>
</table>
Resize JPEG from 1280x720 to 320x180.
Up to 4.4x throughput improvement when MPS enabled.

<table>
<thead>
<tr>
<th>Processes Number</th>
<th>FPS - MPS OFF</th>
<th>FPS - MPS ON</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>937</td>
<td>2007</td>
<td>2.14</td>
</tr>
<tr>
<td>4</td>
<td>904</td>
<td>2910</td>
<td>3.22</td>
</tr>
<tr>
<td>6</td>
<td>897</td>
<td>3451</td>
<td>3.85</td>
</tr>
<tr>
<td>8</td>
<td>894</td>
<td>3813</td>
<td>4.26</td>
</tr>
<tr>
<td>10</td>
<td>890</td>
<td>3848</td>
<td>4.32</td>
</tr>
<tr>
<td>12</td>
<td>891</td>
<td>3878</td>
<td>4.35</td>
</tr>
<tr>
<td>14</td>
<td>900</td>
<td>3860</td>
<td>4.29</td>
</tr>
<tr>
<td>16</td>
<td>889</td>
<td>3921</td>
<td>4.41</td>
</tr>
<tr>
<td>18</td>
<td>886</td>
<td>3942</td>
<td>4.45</td>
</tr>
</tbody>
</table>
MULTI-INSTANCE GPU
GPU ARCHITECTURE AND CUDA

CUDA 8.0
2016
PASCAL
HBM, NVLINK, FP16

CUDA 9.0
2017
VOLTA
HBM, NVLINK, TENSOR CORES, MPS

CUDA 10.0
2018
TURING
TENSOR CORES, RT CORES

CUDA 11.0
2020
AMPERE
HBM, NVLINK, TENSOR CORES, PARTITIONING
A100 GPU
Highest Performance, Efficiency and Utilization

<table>
<thead>
<tr>
<th>New Technology</th>
<th>Benefit over Volta</th>
</tr>
</thead>
</table>
| Faster Tensor Core for AI, support FP16 & bfloat16 | >2x V100 RN50 & Transformer train  
~3x Tensor Core FLOPS  
Dramatically reduce time-to-soln. |
| New Tensor Core for HPC | 2.5x FP64 FLOPS  
Accelerate core HPC kernels |
| Wider + Faster Memory | 1.7x memory bandwidth  
Up to 40GB per GPU  
Larger model & dataset |
| New NVLINK3 + PCIe Gen4 | 2x NVLINK bandwidth  
2x PCIe bandwidth + SR-IOV |
| **New Multi-Instance GPU, with Fault and Perf Isolation** | Up to 7 concurrent GPUs  
Higher utilization  
Substantially lower entry cost |
| New Hardware Engines | JPEG HW decoder, 5 video NVDEC  
Optical flow accelerator |
NEW MULTI-INSTANCE GPU (MIG)
Optimize GPU Utilization, Expand Access to More Users with Guaranteed Quality of Service

- **Up To 7 GPU Instances In a Single A100**: Dedicated SM, Memory, L2 cache, Bandwidth for hardware QoS & isolation
- Simultaneous Workload Execution With Guaranteed Quality Of Service: All MIG instances run in parallel with predictable throughput & latency
- Right Sized GPU Allocation: Different sized MIG instances based on target workloads
- Flexibility: to run any type of workload on a MIG instance
- Diverse Deployment Environments: Supported with Bare metal, Docker, Kubernetes, Virtualized Env.
MIG ISOLATION

Computational Isolation
• SM are not shared between MIGs
• This provides high QoS for each MIG users

DRAM Bandwidth Isolation
• Slices of the L2 cache are physically associated with particular DRAM channels and memory
• Isolating MIGs to non-overlapping sets of L2 cache slices does two things:
  • Isolates BW
  • Allocates DRAM memory between the MIGs

Configuration Isolation
• Creating GPU Instances or Compute Instances do not disturb work running on existing instances

Error Isolation
• Resources within the chip are separately resettable
# GPU Instance Profiles

For A100-SXM4-40GB

<table>
<thead>
<tr>
<th>GPU Instance</th>
<th>Number of Instances Available</th>
<th>SMs</th>
<th>Memory</th>
<th>NVDECs</th>
<th>Training</th>
<th>Inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1g.5gb</td>
<td>7</td>
<td>14</td>
<td>5 GB</td>
<td>0</td>
<td>BERT Fine-tuning (e.g. SQuAD), Multiple chatbots, Jupyter notebooks</td>
<td>Multiple inference (e.g. TRITON); ResNet-50, BERT, WnD networks</td>
</tr>
<tr>
<td>2g.10gb</td>
<td>3</td>
<td>28</td>
<td>10 GB</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3g.20gb</td>
<td>2</td>
<td>42</td>
<td>20 GB</td>
<td>2</td>
<td>Training on ResNet-50, BERT, WnD networks</td>
<td></td>
</tr>
<tr>
<td>4g.20gb</td>
<td>1</td>
<td>56</td>
<td>20 GB</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7g.40gb</td>
<td>1</td>
<td>98</td>
<td>40 GB</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FLEXIBLE MIG CONFIGURATIONS FOR DIFFERENT SCENARIOS

<table>
<thead>
<tr>
<th>Slice #1</th>
<th>Slice #2</th>
<th>Slice #3</th>
<th>Slice #4</th>
<th>Slice #5</th>
<th>Slice #6</th>
<th>Slice #7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- 18 possible configurations
- NVML or NVIDIA-SMI to create and retire Instance
- Config. can be dynamically updated when the GPU slices involved are idle
EXAMPLE: TWO LEVEL PARTITIONING

GPU Instances and Compute Instances

4 Parallel CUDA processes / containers

One container

Debugger

GPU Instance

4g.20gb

GPU Instance

2g.10gb

GPU Instance

1g.5gb

GPU

Memory

GPU

Memory

Compute Instance

Compute Instance
ENABLEMENT ACROSS SOFTWARE STACK

- Support for bare-metal and containerized environments
  - Interaction directly via NVML/nvidia-smi
  - Kubernetes (device enumeration, resource type), Slurm
  - Docker CLI
- Monitoring and management (including device metrics association to MIG)
USER WORKFLOW: MIG MANAGEMENT
List/Create/Update/Destroy Instances via NVML and nvidia-smi

GPU reset required to enable/disable MIG mode (one-time operation)

Use NVML/nvidia-smi (even through containers) to manage MIG

Example: Create new instance with nvidia-smi
MIG: RUNNING DOCKER CONTAINERS

User Workflow

- Run GPU containers with MIG using "-gpus" option in Docker 19.03
  - Primarily for single node development and testing
- Enabled via NVIDIA Container Toolkit (previously known as nvidia-docker2)
- Users configure MIG partitions using NVML/nvidia-smi
- Launching the container requires specifying the GPU instances to expose to the container

```bash
$ docker run \
  --gpus "device=0:0,0:1" \
  nvidia/cuda:11.0-base nvidia-smi -L

GPU 0: A100-SXM4-40GB (UUID: GPU-2ceff3df-31b3-caf2-eace-a494b4b7926b) 
  MIG 3g.20gb Device 0: (UUID: MIG-GPU-2ceff3df-31b3-caf2-eace-a494b4b7926b/1/0) 
  MIG 3g.20gb Device 1: (UUID: MIG-GPU-2ceff3df-31b3-caf2-eace-a494b4b7926b/2/0) 

$ docker run \
  --gpus "device=MIG-GPU-2ceff3df-31b3-caf2-eace-a494b4b7926b/1/0" \
  nvidia/cuda:11.0-base nvidia-smi -L

GPU 0: A100-SXM4-40GB (UUID: GPU-2ceff3df-31b3-caf2-eace-a494b4b7926b) 
  MIG 3g.20gb Device 0: (UUID: MIG-GPU-2ceff3df-31b3-caf2-eace-a494b4b7926b/1/0)
```
MIG: RUNNING CONTAINERS USING K8S

User Workflow

- MIG configured on the node ahead of time
- Expected to be transparent to the end user
- Simple exposure model for homogenous nodes
- Other exposure options still in discussion and not settled yet
- User jobs will be able to only execute on a single Compute Instance

```
apiVersion: v1
className: Pod
metadata:
  name: gpu-example
spec:
  containers:
  - name: gpu-example
    image: nvidia/cuda:11.0-base
    resources:
      limits:
        nvidia.com/gpu: 1
    nodeSelector:
      nvidia.com/gpu.product: A100-SXM4-PA10-SIM-1G-XV
      nvidia.com/cuda.runtime: 11.0
      nvidia.com/cuda.driver: 450.28.0
```
MIG TEST CASE 1 - BERT LARGE INFEERENCE

Test Results

Perf among 7 MIG 1g.5gb slice is very stable and consistent. MIG provides great perf isolation and QoS.

2.1x throughput when MIG is enabled for this case and config.
MIG TEST CASE 1 - BERT LARGE INFERENCE

GPU Utilization Metrics

GPU Device Level Utilization Metrics

- SM Activity
- Tensor Util
- Memory Activity

MIG: 1*1g.5gb
- No MIG: Whole GPU
- MIG: 7*1g.5gb
MIG TEST CASE 2 - JASPER INference

Test Results

Throughput: amount of audio seconds processed by GPU in one second

With MIG enabled, throughput up to 3.4x improvement.
TRITON AND VGPU BRIEF
INEFFICIENCY LIMITS INNOVATION
Difficulties with Deploying Data Center Inference

Single Model Only

Single Framework Only

Solutions can only support models from one framework

Custom Development

Developers need to reinvent the plumbing for every application

Some systems are overused while others are underutilized
NVIDIA TRITON INFRINGEMENT SERVER
Production Data Center Inference Server

Maximize real-time inference performance of GPUs

Quickly deploy and manage multiple models per GPU per node

Easily scale to heterogeneous GPUs and multi GPU nodes

Integrates with orchestration systems and auto scalers via latency and health metrics

Now open source for thorough customization and integration
DYNAMIC BATCHING

2.5X Faster Inferences/Second at a 50ms End-to-End Server Latency Threshold

**Triton Inference Server** groups inference requests based on customer defined metrics for optimal performance.

Customer defines
1) batch size (required)
2) latency requirements (optional)

Example: No dynamic batching (batch size 1 & 8) vs dynamic batching
VGPU FOR GRAPHICS AND COMPUTING

Virtual PC
- NVIDIA Graphics Driver
  - vGPU

Virtual PC
- NVIDIA Graphics Driver
  - vGPU

Virtual Workstation
- NVIDIA Quadro Driver
  - vGPU

Virtual Workstation
- NVIDIA Quadro Driver
  - vGPU

Virtual Compute
- NVIDIA Compute Driver
  - vGPU

Virtual Compute
- NVIDIA Compute Driver
  - vGPU

Hypervisor
- NVIDIA GRID vGPU manager

Hardware
- CPUs

Server
- NVIDIA GPU
  - H.264 Encode
VGPU FOR COMPUTING

vCS

Hypervisor provides best security, isolation guarantee.

vCS provides a good option for cost sensitive customers and those new comers to GPU computing, or application of low-utilized GPU scenarios.

Flexible scheduler strategy: Best effort, fixed-share, equal-share.

Flexible scheduler time slice (1-20 ms controllable).

Perf is guaranteed even that it’s time-round sharing for SM resources.
## CUDA CONCURRENCY MECHANISMS

Triton, MPS, vGPU and MIG

<table>
<thead>
<tr>
<th></th>
<th>Parallel work</th>
<th>Address space isolation</th>
<th>SM performance isolation</th>
<th>Memory performance isolation</th>
<th>Error isolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRITON (CUDA Streams)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MPS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (by percentage, not partitioning)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>vGPU</td>
<td>Yes</td>
<td>Yes (With hypervisor)</td>
<td>Yes (Time-slicing)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MIG</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## COMPARISON

### Part 1

<table>
<thead>
<tr>
<th></th>
<th>MPS</th>
<th>vGPU</th>
<th>TRITON</th>
<th>MIG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intro Link</strong></td>
<td>MPS Whitepaper</td>
<td>Official Link</td>
<td>Github</td>
<td>MIG Whitepaper-NDA</td>
</tr>
<tr>
<td><strong>Open Source</strong></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Free</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Main Positioning</strong></td>
<td>Improve GPU utilization for applications that doesn’t fully utilize GPU, by schedule multi-process, with limited execution resource.</td>
<td>Offer a consistent user experience for every virtual workflow and improve GPU utilization in some scenario, by split GPU into multiple vGPUs as memory size equal partition, by integrating with hypervisor (virtual machine technology).</td>
<td>Provide a cloud inferencing solution optimized for NV GPU, with an inference service via HTTP or gRPC endpoint.</td>
<td>Improve GPU utilization and serve more users with physical resource isolation and QoS guarantee.</td>
</tr>
<tr>
<td><strong>Target Applications</strong></td>
<td>Applications that doesn’t fully utilize GPU: HPC-MPI application, training, inference with small matrix size.</td>
<td>3D Rendering, vGaming, training, inference.</td>
<td>Inference.</td>
<td>Training, inference, HPC.</td>
</tr>
</tbody>
</table>
## COMPARISON
### Part 2

<table>
<thead>
<tr>
<th>Supported GPU</th>
<th>MPS</th>
<th>vGPU</th>
<th>TRITON</th>
<th>MIG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPU since Kepler</td>
<td>P100, P40, P4, P6, V100, T4, RTX8000, RTX6000, M10, M60</td>
<td>All GPU</td>
<td>A100</td>
</tr>
<tr>
<td>Supported OS</td>
<td>Linux</td>
<td>Linux, Windows</td>
<td>Linux</td>
<td>Linux</td>
</tr>
<tr>
<td>Extra Software Needed</td>
<td>No</td>
<td>Hypervisor(KVM, Citrix, VMWare, etc)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Benefits</td>
<td>Improve GPU utilization, improve throughput</td>
<td>Improve GPU utilization via time-sharing, improve user experience</td>
<td>Improve GPU utilization, improve throughput</td>
<td>Improve GPU utilization, improve throughput, serve more users, provide QoS and fault isolation.</td>
</tr>
<tr>
<td>GPU Resource Isolation</td>
<td>Context level isolation, memory and SM sharing</td>
<td>GPU memory isolation, SM sharing by rotation.</td>
<td>TRTIS executes model(app) instance as Thread(CPU)-Stream(GPU). SM sharing is via multi-stream.</td>
<td>GPU memory isolation, SM isolation, other engines isolation(CEs, NVDEC).</td>
</tr>
</tbody>
</table>
## COMPARISON
### Part 3

### Simple Comparison Among MPS, vGPU, TRITON, MIG

<table>
<thead>
<tr>
<th></th>
<th>MPS</th>
<th>vGPU</th>
<th>TRITON</th>
<th>MIG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>QoS</strong></td>
<td>No strong guarantee</td>
<td>Guarantee in time-slicing sharing envelop</td>
<td>No strong guarantee</td>
<td>Strong, the best guarantee</td>
</tr>
<tr>
<td><strong>Ease of Use</strong></td>
<td>Easy</td>
<td>Medium</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td><strong>Support</strong></td>
<td>Forum</td>
<td>Professional team</td>
<td>Github issue</td>
<td>Professional team</td>
</tr>
<tr>
<td><strong>Considerations/Limitations</strong></td>
<td>No fault tolerance. Really not suitable for arbitrary combination of multi-user applications, especially for public cloud scenario with full isolation requirements.</td>
<td>Not really sharing SM as this is a time-sharing/slicing implementation.</td>
<td>Mainly confined to inference type workloads. Multi-streaming currently not effective to TF based models (limiting factor from TensorFlow).</td>
<td>Only for compute workloads in MIG mode, don’t support P2P between GPU compute instances.</td>
</tr>
<tr>
<td><strong>Correlations</strong></td>
<td>MPS, vGPU, TRITON, MIG are not mutually exclusive solutions. Example: you can run MPS or TRITON in vGPU environment. Example: you can run MPS or vGPU in MIG-enabled A100 system. Example: you can even run multi processes in TRITON with MPS enabled, under vGPU with MIG-enabled system.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>