KubeCon | CloudNativeCon

OPEN SOURCE SUMMIT

China 2023

KubeCon | CloudNativeCon
OPEN SOURCE SUMMIT
China 2023

Container Live Migration in Kubernetes Production Environment

Yenan Lang And Hua Liu, Tencent

History of live migration

2011

- Checkpoint Restore In Userspace(CRIU) started

2015

- Jan Kubernetes community initiated an ongoing discussion :
  [Pod lifecycle checkpointing · Issue #3949]
- Jul Kubernetes v1.0.0 was released without live migration, and live migration has never been supported since then

2018

- Google Borg presented their experience of using live migration at the LPC conference

Tencent

2021

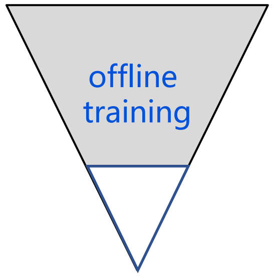
- Apr Container live migration project was started

2022

- Feb Container live migration was deployed in our production environment, serving 20000 migrations daily

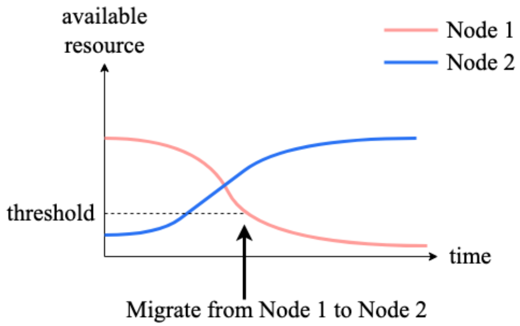
The first implementation of live migration in a production environment of Kubernetes

Why we want container live migration



Objective: reducing cost of offline training

- Our Ads team developed an offline training platform based on Flink
- The training is insensitive to latency
- The training cost accounts for a significant portion of the total cost



Strategy: utilizing low price resources

- While the price is low, it always comes with low stability, such as AWS spot instance
- We should keep the workload running on high-performance nodes
- The cost of rescheduling Flink task manager is too high
- Achieving low-cost rescheduling through container live migration

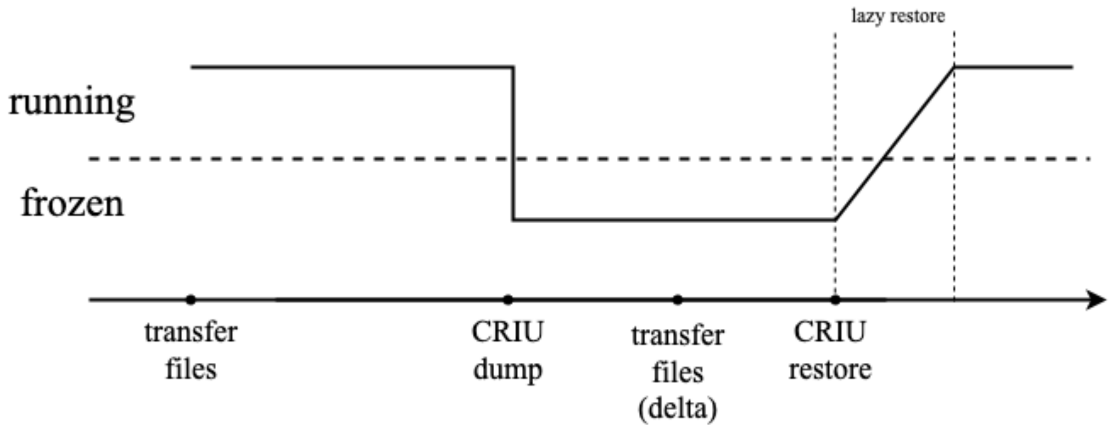
Overview

- **container live migration**
  - **process migration**
    - CRIU
    - linux kernel — time ns
    - data compression
    - lazy restore
  - **K8s**
    - container network — IP migration on flannel
    - runtime
      - runc
      - migration runtime (invokes runc checkpoint/restore)
    - Orchestration
      - migration mode
        - pre-copy
        - hybrid
      - node to node direct transfer
      - pod with multiple containers
  - **Flink** — slightly modified job manager

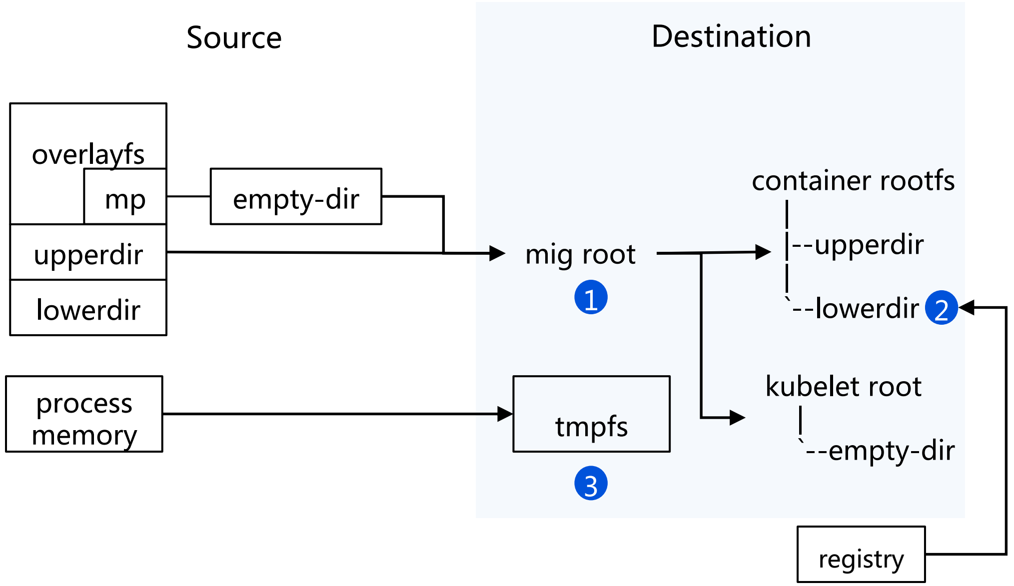
Effect

- No need to modify K8s source code
- Not requiring latest K8s （we are running on K8s v1.18）
- Live migration for pods of Deployment and Statefulset is supported

Process migration



- We only dump the process once because the performance of iterative migration is unstable
- Process files are transferred using a delta-transfer algorithm to reduce process down time
- Optimized ghost files handling：unlimited size + delta-transfer
- Accelerating the dump：data compression + parallel transfer
- lazy restore：on-demand memory loading on destination side

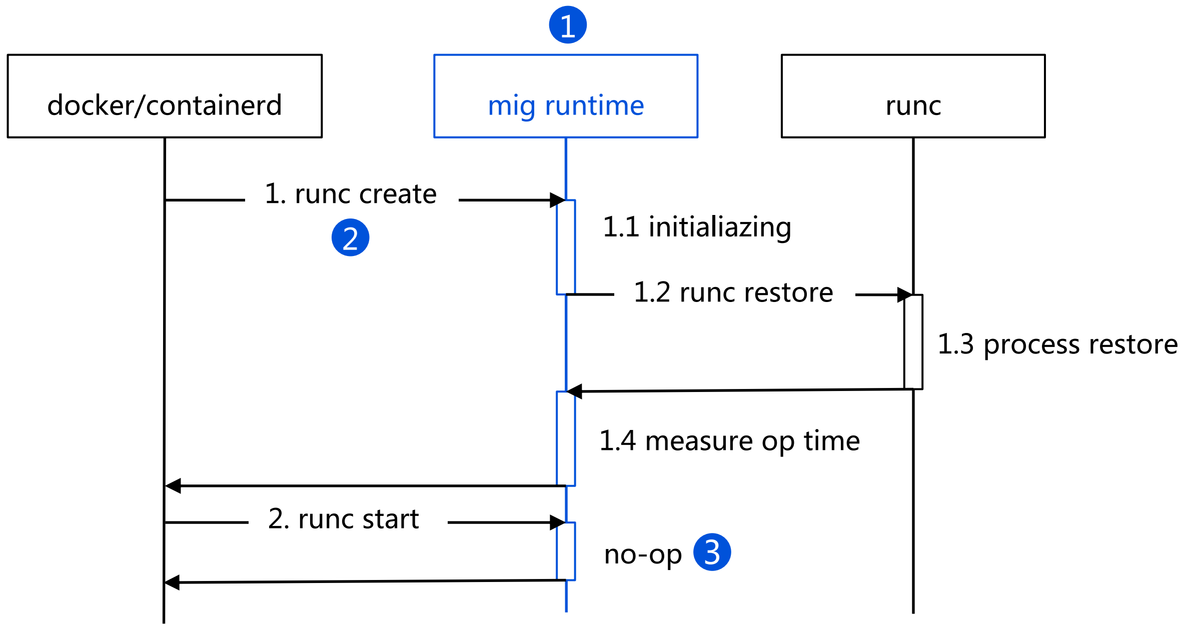


1. Leveraging syscall rename for fast file system recovery
2. Recovery container lowerdir by pulling image from registry
3. Process memory is stored in tmpfs

Runtime

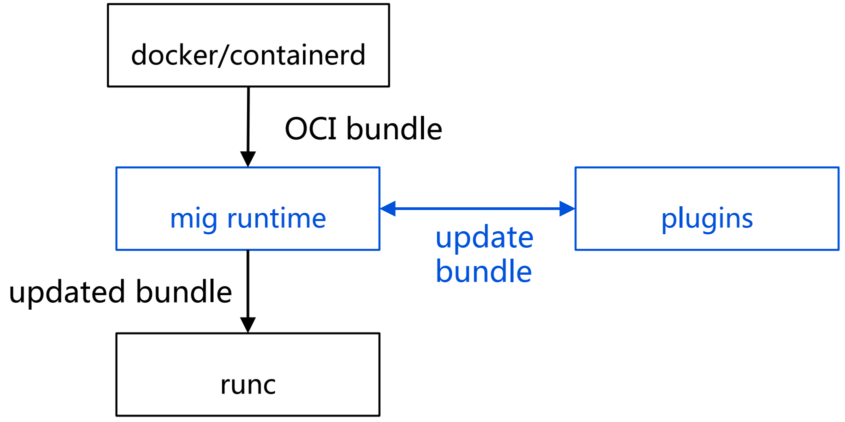
Workflow of container restore

no need to modify kubelet/runtime source code



1. Mig runtime intercepts runc calls and determines whether to create or restore a container according to the 'env' in the OCI bundle.
2. In case of restoring a container, *runc create* is replaced with *runc restore*
3. The process is running as soon as *runc restore* returns, so *runc start* is replaced with no-op

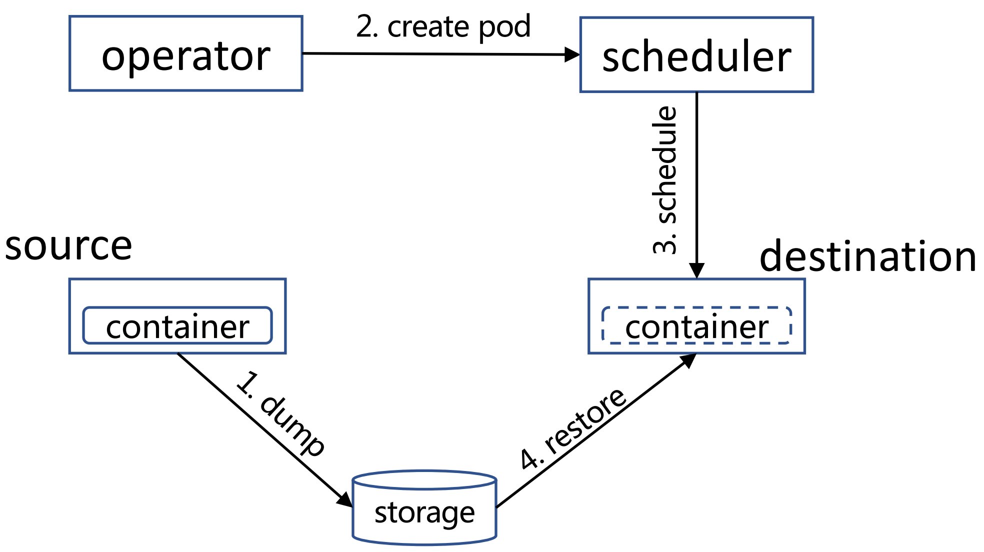
runtime extension



- While K8s not fully leverage all OCI features, such as hooks, extending at the bundle level allows for the utilization of all available features
- The OCI bundle is closer to the OS, making it more flexible and simpler to extend at this level
- We have developed a series of plugins based on the mig runtime that serve big data and AI applications, including device management, improved isolation, and log management

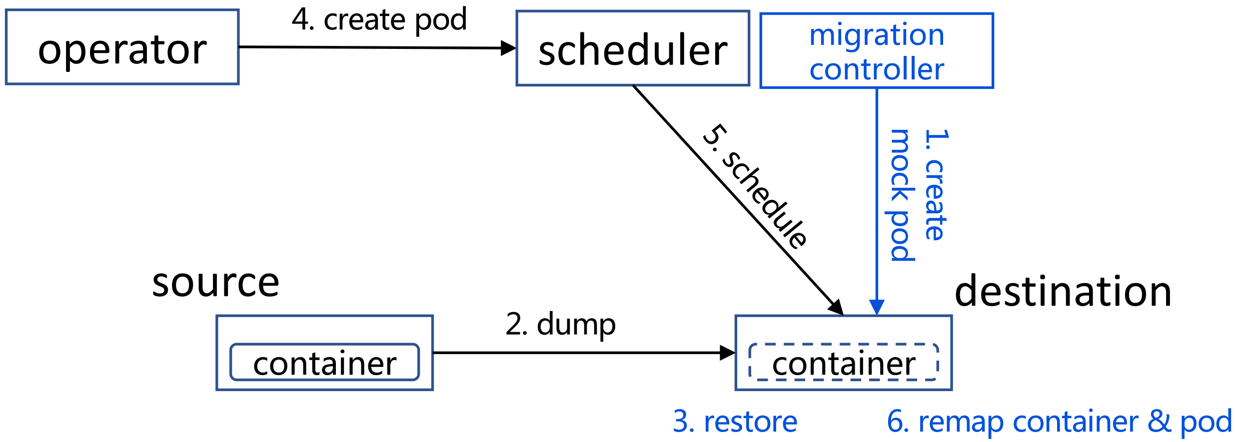
Orchestration

Transfer Twice



- process down time = $t_{dump} + t_{create} + t_{schedule} + t_{restore}$
- t_{dump} *and* $t_{restore}$ are affected by local bandwidth and remote storage performance. During large-scale migrations, remote storage may become a bottleneck
- t_{create} and $t_{schedule}$ exhibit unstable performance in busy clusters
- Since the old Pod must be deleted before the new Pod can be created, it is not possible to implement hybrid migration

Node to Node Direct Transfer



- We have managed to remap containers to different Pods without modifying the K8s code. With the remapping, we can
  - restore the process immediately after the dump is completed
  - make the new pod and the old pod independent. Based on this, we have implemented the hybrid migration mode
- The restore operation is performed using local data, resulting in minimal time consumption. Therefore, the down time for the process depends only on the time it takes to dump the process memory
- Both the operator and scheduler operations occur after the restore operation, this ensures stable performance of live migration even in busy clusters

Performance: Pod recreate vs migration

Pod recreate

percentile	time (s)
10th	7
50th	9
75th	13
90th	17.3
99th	32.66

Pod recreate time: the total time required for creating the Pod, scheduling, starting the pause container, executing init containers, and starting the main containers

PS： Pod recreate time does not include the time required for terminating containers and deleting the Pod, as these operations can be optimized to some extent

pre-copy migration(mem: 10G)

percentile	total (s)	down time (秒)
10th	24	11
50th	31	17
75th	38	21
90th	45	26
99th	63	37

total： the time from triggering the live migration to the completion of all operations

down time： the time from CRIU dump to the completion of process restoring

hybrid migration

mem	down time (s)	pre-dump time(s)	lazy restore time(s)
1	**0.7**	2	1.1
16	**1.3**	15.8	16.6
32	**1.4**	24.8	35.4
64	**2.9**	59.25	73.5

down time： the time from CRIU dump to the completion of process restoring

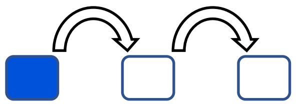
pre-dump time： the cumulative time for executing CRIU pre-dumps

lazy restore time：the time of loading all process memory

Live migration and cloud native

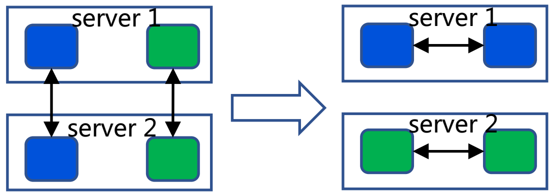
Rescheduling is the core ability of cloud native

Scenario 1：spot instance



- Providing a stable operating environment for jobs on unstable resources
- When resources are reclaimed, workload can be migrated to other nodes

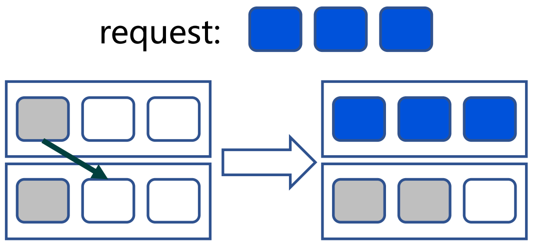
Scenario 2：topology optimization



node to node communication   local communication

- Optimize the topology based on the actual runtime of workloads, without relying solely on predictions and planning
- Continuously adjust the workload topology to achieve the optimal state

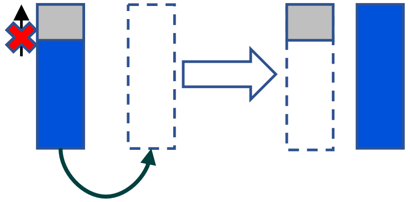
Scenario 3：resource fragment



schedule fail   schedule succeed

- Dynamically adjust resource distribution on-demand to meet the resource requirements of different applications

Scenario 4：VPA

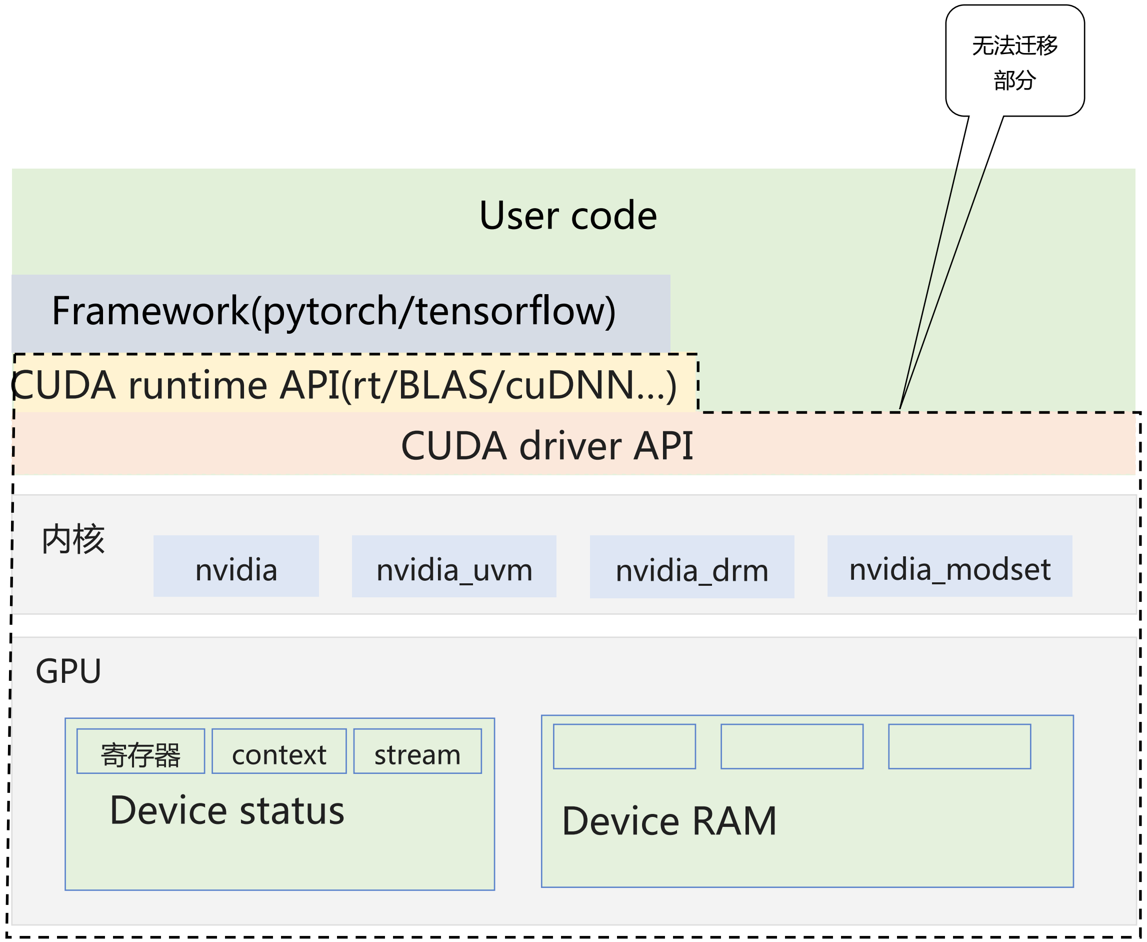


- When local resources are insufficient for scaling, migrate to other nodes

Low-cost rescheduling based on live migration

- Utilize live migration technology to avoid job restart and reduce the cost of rescheduling
- Deeply integrate with the compute engines to provide general purpose live migration capability
- Achieve an optimal topology and improve training speed by utilizing GPU live migration

Challenges of GPU live migration



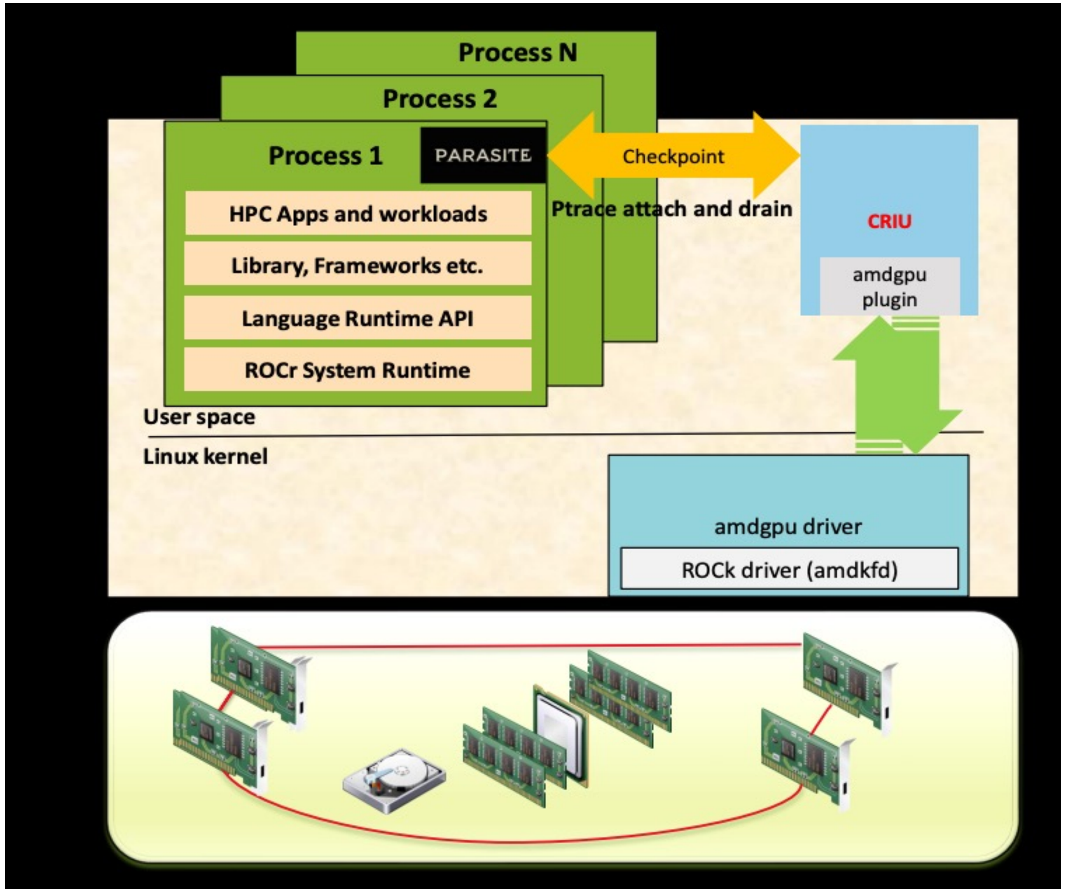
CUDA stack

As a passthrough device, GPU does not support CR

- Challenges in importing/exporting GPU hardware state
- Mapping VRAM to the application address
- Migration of device files
  - /dev/nvidiactl
  - /dev/nvidia#num
- Support from Vender
  - NVIDIA vGPU support live migration
    - Need license
    - VM scenarios
  - CUDA does not support checkpoint/restore
  - AMD ROCm is attempting to support CR

Industry's attempt - AMD ROCm

AMD upstream CR features to Linux/CRIU community

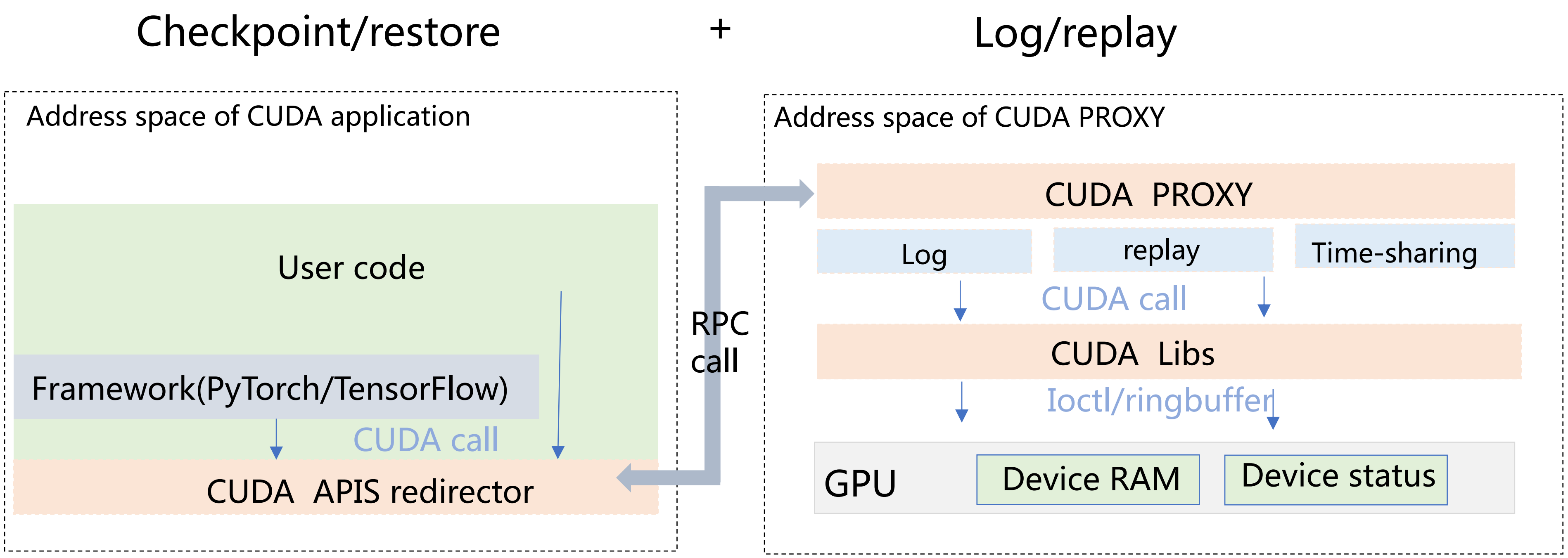


AMD ROCm CR流程

- CRIU：add 3 hook to cooperate with plugin
  - CR_PLUGIN_HOOK__RESUME_DEVICES_LATE
  - CR_PLUGIN_HOOK__HANDLE_DEVICE_VMA
  - CR_PLUGIN_HOOK__UPDATE_VMA_MAP
- AMDGPU plugin
  - Link CRIU and KFD modules.
  - Checkpoint/restore GPU State
- KFD extends resource CR
  - Memory
  - Queues
  - Events
  - Topology
- Extends ioctls
  - CRIU_PAUSE
  - CRIU_PROCESS_INFO
  - CRIU_DUMPER
  - CRIU_RESTORER
  - CRIU_RESUME

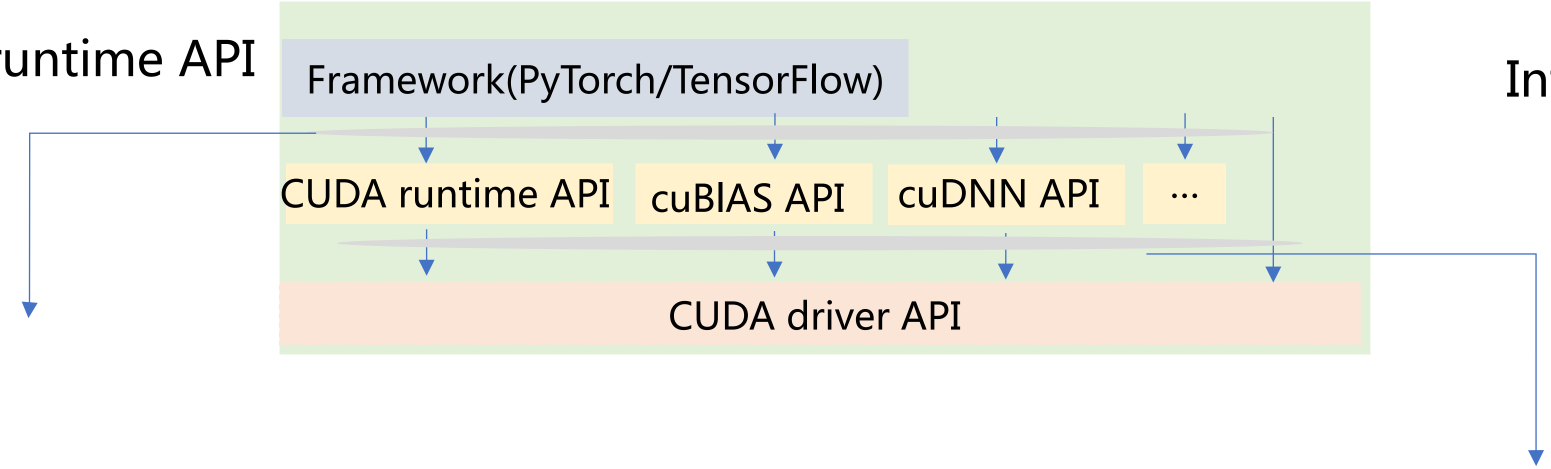
Solution for CUDA

Separating and managing GPU and CPU states



Solution : intercept runtime or driver API?

Intecept runtime API



Intercept driver API

Pros :

1. All APIs are public, no private ones

Cons :

1. Large numbers of APIs (including driver APIs);
2. Determined by the user's image and changes rapidly;
3. Programs need to be recompiled (libcudart);
4. Two types of APIs: C/C++.

Pros :

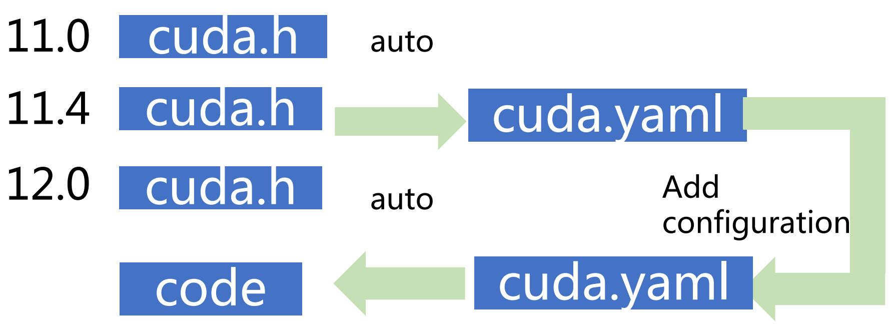
1. Have high stability bound to the NVIDIA KO;
2. Limited numbers of APIs(~400);
3. Programs no need to be recompiled;
4. One type of API: C

Cons :

1. cuGetExportTable
2. Lots of hidden functions

Generate redirect APIS- Open VS Close

Open part : Generate automatically




```yaml
- !CudaFunc
  name: cuMemcpyHtoD_v2
  stage: 2
  is_macro: false
  record_api: false
  multi_version: false
  args:
  - !FuncArg
    name: dstDevice
    type: CUdeviceptr
    rpc_type: ptr
  - !FuncArg
    name: srcHost
    type: const void*
    rpc_type: mem_data
    rpc_type_binds: ByteCount
  - !FuncArg
    name: ByteCount
    type: size_t
```

```yaml
- !CudaFunc
  name: cuMemAlloc_v2
  stage: 2
  is_macro: false
  has_return: true
  record_api: true
  multi_version: false
  args:
  - !FuncArg
    name: dptr
    type: CUdeviceptr*
    rpc_type: ptr_result
    rpc_type_binds: ''
    resource_type: ''
    resource_map: ''
  - !FuncArg
    name: bytesize
    type: size_t
```

content of cuda.yaml

Closed part: Reverse engineering

Libcudart, libblas, libcudnn use many hidden funtions

1. Get function ptr array via cuGetExportTable
2. Directly call function pointers when needed, without invoking the exposed CUDA API

```c
CUresult cuGetExportTable(const void **ppExportTable,
                          const CUuuid *pExportTableId);
```

```c
static Cuuuid uuids[] = {
    {0x6b, 0xd5, 0xfb, 0x6c, 0x5b, 0xf4, 0xe7, 0x4a, 0x89, 0x87, 0xd9, 0x39, 0x12, 0xfd, 0x9d, 0xf9},
    {0xa0, 0x94, 0x79, 0x8c, 0x2e, 0x74, 0x2e, 0x74, 0x93, 0xf2, 0x8, 0x0, 0x20, 0xc, 0xa, 0x66},
    {0x42, 0xd8, 0x5a, 0x81, 0x23, 0xf6, 0xcb, 0x47, 0x82, 0x98, 0xf6, 0xe7, 0x8a, 0x3a, 0xec, 0xdc},
    {0xc6, 0x93, 0x33, 0x6e, 0x11, 0x21, 0xdf, 0x11, 0xa8, 0xc3, 0x68, 0xf3, 0x55, 0xd8, 0x95, 0x93},
    {0xd4, 0x8, 0x20, 0x55, 0xbd, 0xe6, 0x70, 0x4b, 0x8d, 0x34, 0xba, 0x12, 0x3c, 0x66, 0xe1, 0xf2},
    {...},
}
```

Solution – which APIs to log/replay

APIs/state those do not need log/replay

- Registers
  - Drain active kernels via cudaDeviceSynchronize
  - No active kernels during migration
- APIs that does not change Context
  - cuLaunchKernel
  - cuMemcpy.*

APIs/state those do need log/replay

- Resources create/delete APIs
  - cuCtxCreate/cuCtxDestroy
  - cuMemAlloc/cuMemFree
  - …
- APIs those change Context
  - cuInit
  - cuCtxSetCurrent
  - …

Solution- Mutable Vs Immutable after replay

The key point of Log/replay is to classify CUDA resource, distinguishing between mutable and immutable ones

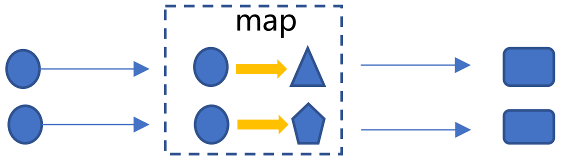
Opaque to Application

Resources：

Context	Module
Stream	Function
Event	

- Applications do not need to understand these resources
- Use as parameters of CUDA calls

Solution：Do remap when replay



Visible to Application

Resources：

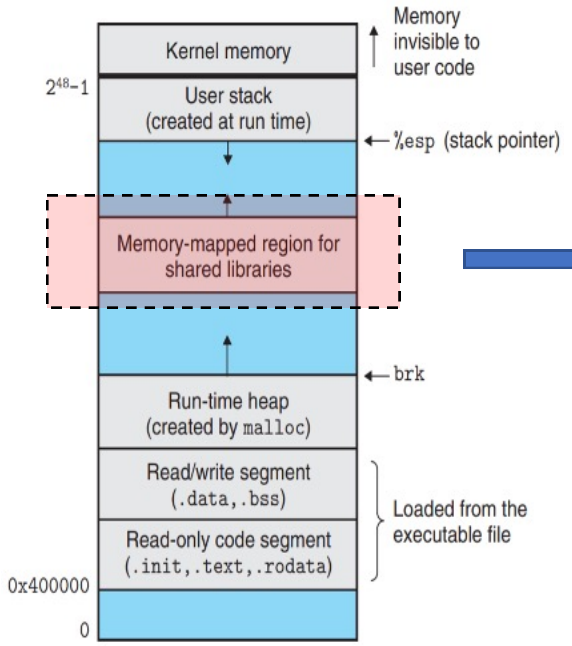
CUdeviceptr

- PyTorch/TensorFlow manage GPU memory themselves.
  - Allocate VRAM during startup
  - Memcpy/launchkernel pass partial memory
  - Unable to avoid address conflict issues after replay

It is necessary to keep the VRAM addresses unchanged during replay.

Solution: How to keep VRAM address unchanged

We should recreate memory layout of CUDA proxy



Big-size malloc

File based mmap

Anonymous mmap

CUDA related mapping
- ✓ malloc called by cuda libs
- ✓ GPU memory mmap

Disable ASLR

log/replay mmaps in order

Process memory layout

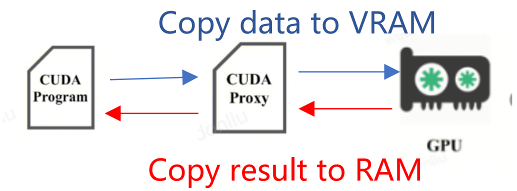
Operations that affect the mmap layout

How to log/replay

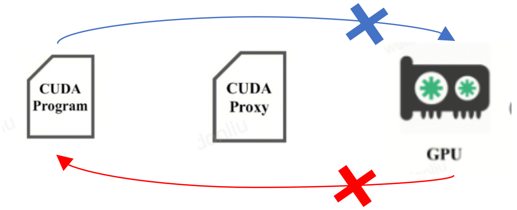
Memory layout ： From separation to unity

Two ways to use VRAM

1. cuMemAlloc/cuMemCpy.*

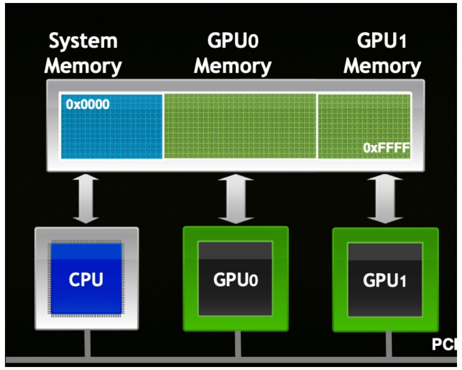


2. App accesses VRAM directly



GPU access RAM directly

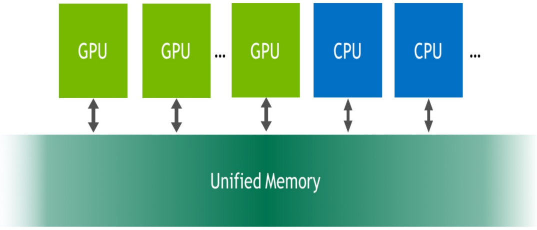
UVA
Unified Virtual Address



cudaHostAlloc allocats Pinned memory

1. Speed up data transfer beween CPU/GPU和GPU p2p OK
2. ZERO COPY：GPU access RAM directly NOT OK
3. Huge performance loss
4. PyTorch/TensorFlow do not use.

UVM
Unified Memory



cudaMallocManaged allocates unified memo

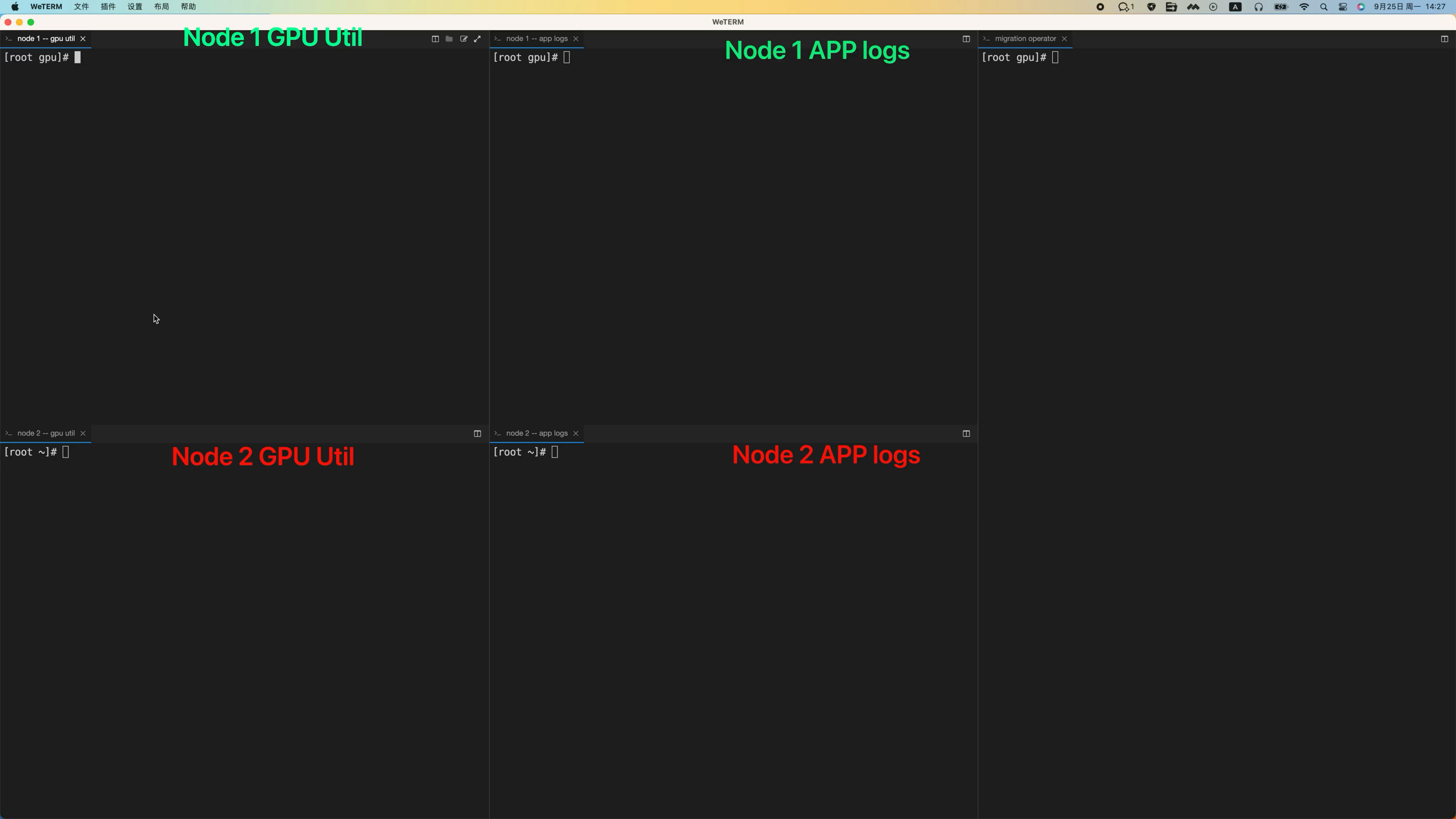
1. Does not specify memory location when allocating.
2. Accessing triggers #PF, leading to sync RAM/VRAM
3. Cannot control the data location, and performance significantly decreases when the VRAM is exhausted.
4. Pytorch/TensorFlow do not use.

DEMO

进展： Successfully ran the benchmark from https://github.com/pytorch/benchmark

Testcases	Exection times		Fatbinary数	Kernel数
	Non-rpc	rpc		
test_pytorch_CycleGAN_and_pix2pix_train_cuda	41.508	53.361s(++28%)	1064	43249
test_BERT_pytorch_train_cuda	6.639	14.771s(+122%)	820	34362

Demo: Migrate BERT from Node 1 to Node2


WeTERM 文件 插件 设置 布局 帮助
9月25日 周一 14:27
WeTERM
node 1 -- gpu util
Node 1 GPU Util
[root gpu]#
node 1 -- app logs
Node 1 APP logs
[root gpu]#
migration operator
[root gpu]#
node 2 -- gpu util
[root ~]#
Node 2 GPU Util
node 2 -- app logs
[root ~]#
Node 2 APP logs