Understanding CUDA Memory Usage

To debug CUDA memory use, PyTorch provides a way to generate memory snapshots that record the state of allocated CUDA memory at any point in time, and optionally record the history of allocation events that led up to that snapshot.

The generated snapshots can then be dragged and dropped onto the interactive viewer hosted at pytorch.org/memory_viz which can be used to explore the snapshot.

Generating a Snapshot

The common pattern for recording a snapshot is to enable memory history, run the code to be observed, and then save a file with a pickled snapshot:

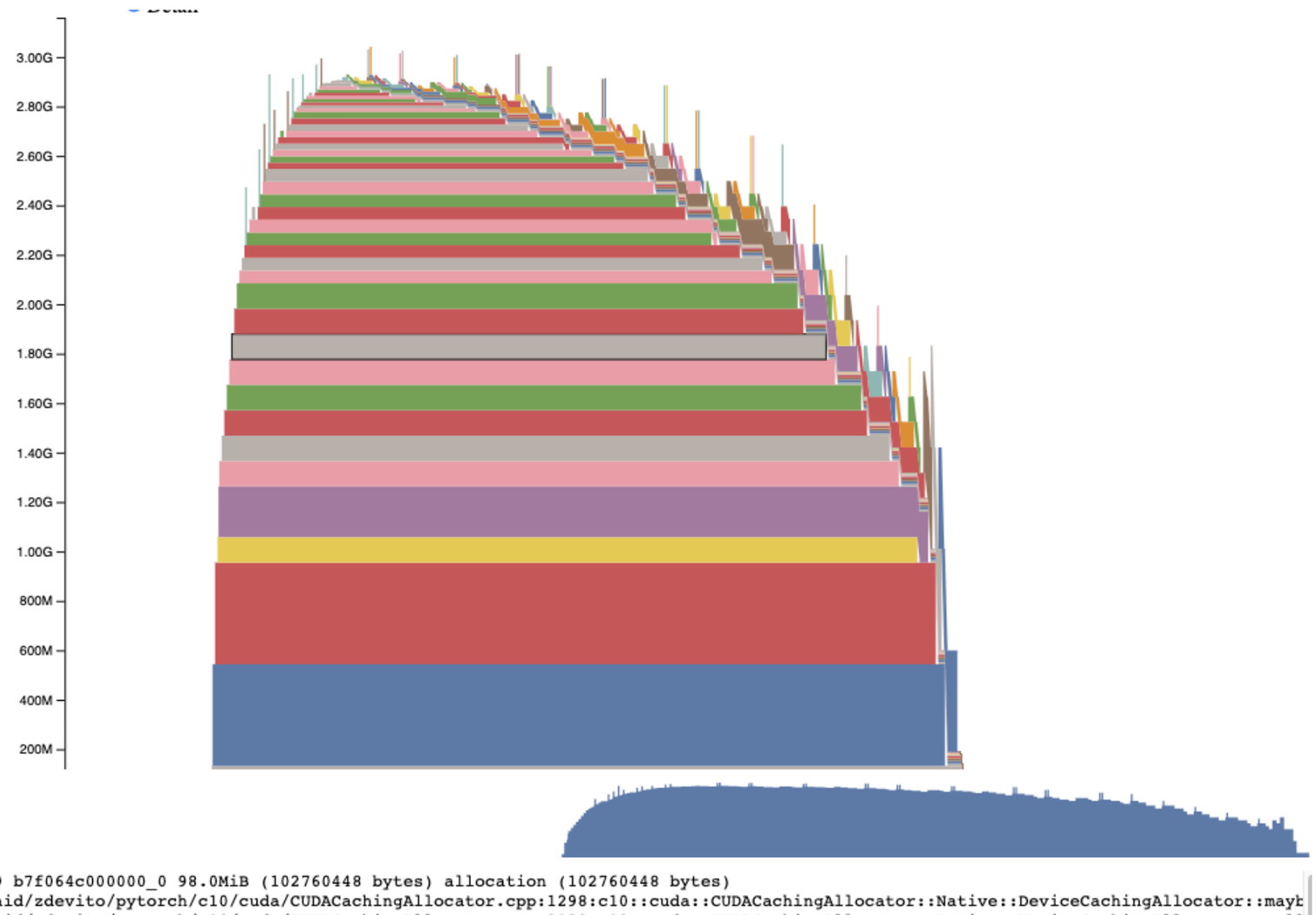
```
# enable memory history, which will
# add tracebacks and event history to snapshots
torch.cuda.memory._record_memory_history()

run_your_code()
torch.cuda.memory._dump_snapshot("my_snapshot.pickle")
```

Using the visualizer

Open pytorch.org/memory_viz and drag/drop the pickled snapshot file into the visualizer. The visualizer is a javascript application that runs locally on your computer. It does not upload any snapshot data.

Active Memory Timeline

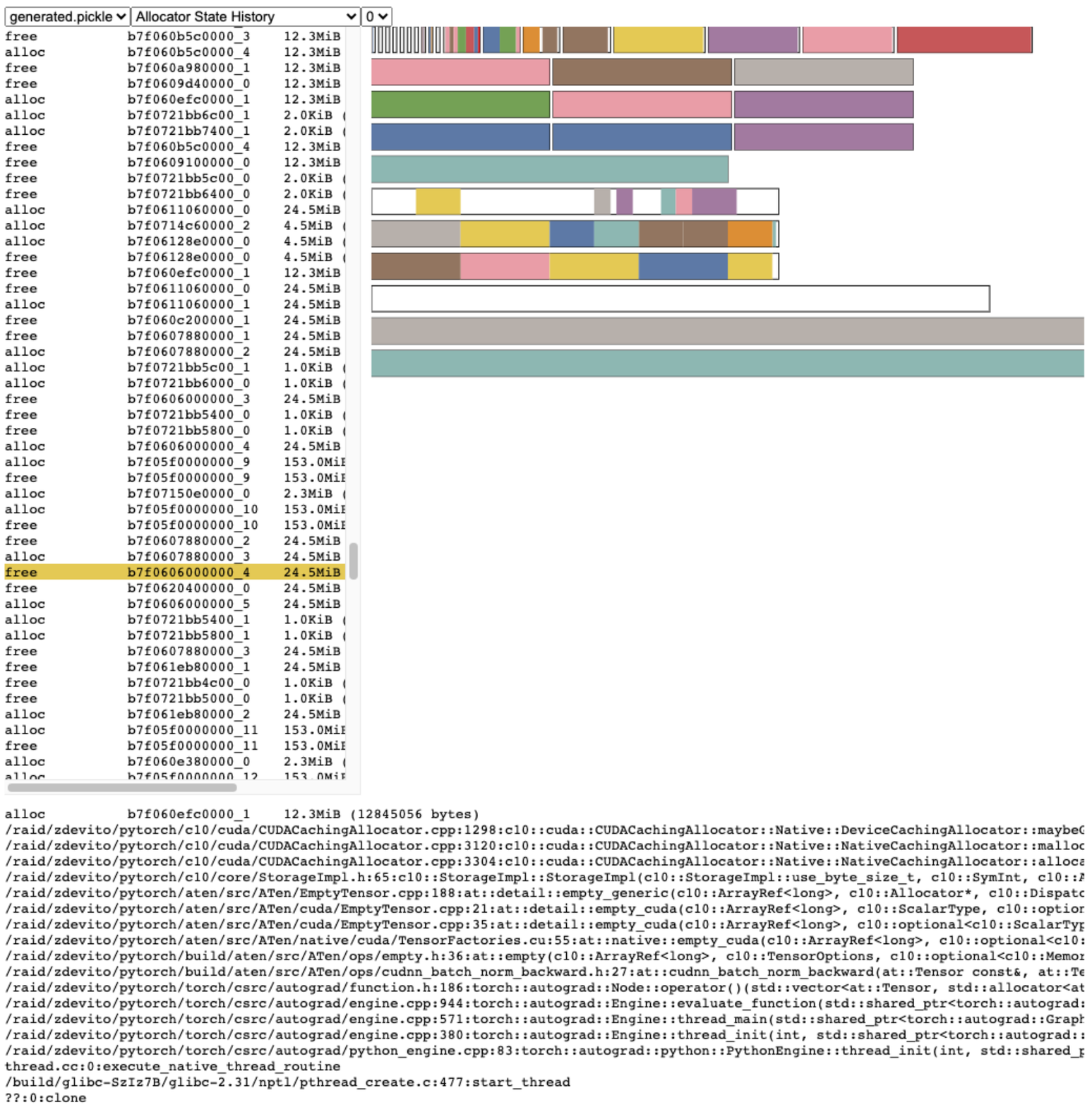
The Active Memory Timeline shows all the live tensors over time in the snapshot on a particular GPU. Pan/Zoom over the plot to look at smaller allocations. Mouse over allocated blocks to see a stack trace for when that block was allocated, and details like its address. The detail slider can be adjusted to render fewer allocations and improve performance when there is a lot of data.




```
249 b7f064c000000_0 98.0MiB (102760448 bytes) allocation (102760448 bytes)
/raid/zdevito/pytorch/c10/cuda/CUDACachingAllocator.cpp:1298:c10::cuda::CUDACachingAllocator::Native::DeviceCachingAllocator::mayb
/raid/zdevito/pytorch/c10/cuda/CUDACachingAllocator.cpp:3120:c10::cuda::CUDACachingAllocator::Native::NativeCachingAllocator::mall
/raid/zdevito/pytorch/c10/cuda/CUDACachingAllocator.cpp:3304:c10::cuda::CUDACachingAllocator::Native::NativeCachingAllocator::allc
/raid/zdevito/pytorch/c10/core/StorageImpl.h:65:c10::StorageImpl::StorageImpl(c10::StorageImpl::use_byte_size_t, c10::SymInt, c10:
/raid/zdevito/pytorch/aten/src/ATen/EmptyTensor.cpp:188:at::detail::empty_generic(c10::ArrayRef<long>, c10::Allocator*, c10::Dispa
/raid/zdevito/pytorch/aten/src/ATen/cuda/EmptyTensor.cpp:21:at::detail::empty_cuda(c10::ArrayRef<long>, c10::ScalarType, c10::opti
/raid/zdevito/pytorch/aten/src/ATen/cuda/EmptyTensor.cpp:35:at::detail::empty_cuda(c10::ArrayRef<long>, c10::optional<c10::ScalarT
/raid/zdevito/pytorch/aten/src/ATen/native/cuda/TensorFactories.cu:55:at::native::empty_cuda(c10::ArrayRef<long>, c10::optional<c1
/raid/zdevito/pytorch/build/aten/src/ATen/ops/empty.h:58:at::empty_symint(c10::ArrayRef<c10::SymInt>, c10::TensorOptions, c10::opt
/raid/zdevito/pytorch/aten/src/ATen/native/TensorFactories.cpp:417:at::native::empty_like(at::Tensor const&, c10::optional<c10::Sc
/raid/zdevito/pytorch/build/aten/src/ATen/ops/empty_like.h:27:at::empty_like(at::Tensor const&, c10::TensorOptions, c10::optional<
/raid/zdevito/pytorch/build/aten/src/ATen/ops/cudnn_batch_norm.h:27:at::cudnn_batch_norm(at::Tensor const&, at::Tensor const&, c10
/raid/zdevito/pytorch/build/aten/src/ATen/ops/_batch_norm_impl_index.h:27:at::_batch_norm_impl_index(at::Tensor const&, c10::optic
/raid/zdevito/pytorch/build/aten/src/ATen/ops/batch_norm.h:27:at::batch_norm(at::Tensor const&, c10::optional<at::Tensor> const&,
/raid/zdevito/pytorch/torch/nn/functional.py:2480:batch_norm
/raid/zdevito/pytorch/torch/nn/modules/batchnorm.py:171:forward
/raid/zdevito/pytorch/torch/nn/modules/module.py:1527:_call_impl
/raid/zdevito/pytorch/torch/nn/modules/module.py:1518:_wrapped_call_impl
/raid/zdevito/vision/torchvision/models/resnet.py:93:forward
/raid/zdevito/pytorch/torch/nn/modules/module.py:1527:_call_impl
/raid/zdevito/pytorch/torch/nn/modules/module.py:1518:_wrapped_call_impl
/raid/zdevito/pytorch/torch/nn/modules/container.py:217:forward
/raid/zdevito/pytorch/torch/nn/modules/module.py:1527:_call_impl
/raid/zdevito/pytorch/torch/nn/modules/module.py:1518:_wrapped_call_impl
/raid/zdevito/vision/torchvision/models/resnet.py:273:_forward_impl
/raid/zdevito/vision/torchvision/models/resnet.py:285:forward
```

Allocator State History

The Allocator State History shows individual allocator events in a timeline on the left. Select an event in the timeline to see a visual summary of the allocator state at that event. This summary shows each individual segment returned from cudaMalloc and how it is split up into blocks of individual allocations or free space. Mouse over segments and blocks to see the stack trace when the memory was allocated. Mouse over events to see the stack trace when the event occurred, such as when a tensor was freed. Out of memory errors are reported as OOM events. Looking at the state of memory during an OOM may provide insight into why an allocation failed even though reserved memory still exists.



The stack trace information also reports the a ress at which an allocation occurre . The a ress 7f064c000000_0 refers to the ( )lock at a ress 7f064c000000 which is the "_0"th time this a ress was allocate . This unique string can e looke up in the Active Memory Timeline an searche in the Active State History to examine the memory state when a tensor was allocate or free .

Sna shot API Reference

torch.cu a.memory._recor _memory_history(*enable='all'*, *context='all'*, *stacks='all'*, *max_entries=9223372036854775807*, *device=None*) [SO RCE]

Ena le recor ing of stack traces associate with memory allocations, so you can tell what allocate any piece of memory in `torch.cuda.memory._snapshot()`.

In a ition too keeping stack traces with each current allocation an free, this will also ena le recor ing of a history of all alloc/free events.

se `torch.cuda.memory._snapshot()` to retrieve this information, an the tools in *_memory_viz.py* to visualize snapshots.

The Python trace collection is fast (2us per trace), so you may consi er ena ling this on pro uction o s if you anticipate ever having to e ug memory issues.

C++ trace collection is also fast (~50ns/frame), which for many typical programs works out to ~2us per trace, ut can vary epen ing on stack epth.

Parameters

- **ena le** (*Literal*[*None*, *"state"*, *"all"*], *optional*) – *None*, isa le recor ing memory history. *"state"*, keep information for currenly allocate memory. *"all"*, a itionally keep a history of all alloc/free calls. Defaults to "all".
- **context** (*Literal*[*None*, *"state"*, *"alloc"*, *"all"*], *optional*) – *None*, Do not recor any trace acks. *"state"*, Recor trace acks for currently allocate memory. *"alloc"*, a itionally keep trace acks for alloc calls. *"all"*, a itionally keep trace acks for free calls. Defaults to "all".
- **stacks** (*Literal*[*"python"*, *"all"*], *optional*) – *"python"*, inclu e Python, TorchScript, an in uctor frames in trace acks *"all"*, a itionally inclu e C++ frames Defaults to "all".
- **max_entries** (*int*, *optional*) – Keep a maximum of *max_entries* alloc/free events in the recor e history recor e .

torch.cu a.memory._snapshot(*device=None*) [SO RCE]

Save a snapshot of C DA memory state at the time it was calle .

The state is represente as a ictionary with the following structure.

```
class Snapshot(TypedDict):
    segments : List[Segment]
    device_traces: List[List[TraceEntry]]

class Segment(TypedDict):
    # Segments are memory returned from a cudaMalloc call.
    # The size of reserved memory is the sum of all Segments.
    # Segments are cached and reused for future allocations.
    # If the reuse is smaller than the segment, the segment
    # is split into more then one Block.
    # empty_cache() frees Segments that are entirely inactive.
    address: int
    total_size: int #  cudaMalloc'd size of segment
    stream: int
    segment_type: Literal['small', 'large'] # 'large' (>1MB)
    allocated_size: int # size of memory in use
    active_size: int # size of memory in use or in active_awaiting_free state
    blocks : List[Block]

class Block(TypedDict):
    # A piece of memory returned from the allocator, or
    # current cached but inactive.
    size: int
    requested_size: int # size requested during malloc, may be smaller than
                        # size due to rounding
    address: int
    state: Literal['active_allocated', # used by a tensor
                   'active_awaiting_free', # waiting for another stream to finish using
                                           # this, then it will become free
                   'inactive',] # free for reuse
    frames: List[Frame] # stack trace from where the allocation occurred

class Frame(TypedDict):
        filename: str
        line: int
        name: str
```

```
    'alloc'  # memory allocated
    'free_requested', # the allocated received a call to free memory
    'free_completed', # the memory that was requested to be freed is now
                      # able to be used in future allocation calls
    'segment_alloc', # the caching allocator ask cudaMalloc for more memory
                     # and added it as a segment in its cache
    'segment_free',  # the caching allocator called cudaFree to return memory
                     # to cuda possibly trying free up memory to
                     # allocate more segments or because empty_caches was called
    'oom',           # the allocator threw an OOM exception. 'size' is
                     # the requested number of bytes that did not succeed
    'snapshot'       # the allocator generated a memory snapshot
                     # useful to coorelate a previously taken
                     # snapshot with this trace
]
addr: int # not present for OOM
frames: List[Frame]
size: int
stream: int
device_free: int # only present for OOM, the amount of
                 # memory cuda still reports to be free
```

Returns

The Snapshot ictionary o ect

torch.cu a.memory._ ump_snapshot(*filename='dump_snapshot.pickle'*) [SO RCE]

Save a pickle version of the *torch.memory._snapshot()* ictionary to a file.

This file can e opene y the interactive snapshot viewer at pytorch.org/memory_viz

Parameters

filename (*str, optional*) – Name of the file to create. Defaults to “ ump_snapshot.pickle”.

‹ Previous

Next ›