补丁基于 pytorch1.12.0 源代码, pytorch1.12.0 源代码地址:
https://github.com/pytorch/pytorch/tree/v1.12.0

文件名: torch/cuda/memory.py (修改)

```
import collections
import contextlib
import warnings
from typing import Any, Dict, Union

import torch
from . import is_initialized, _get_device_index, _lazy_init
from torch.types import Device

__all__ = ["caching_allocator_alloc", "caching_allocator_delete", "set_per_process_memory_fraction",
           "empty_cache", "memory_stats", "memory_stats_as_nested_dict", "reset_accumulated_memory_stats",
           "reset_peak_memory_stats", "reset_max_memory_allocated", "reset_max_memory_cached",
           "memory_allocated", "max_memory_allocated", "memory_reserved", "max_memory_reserved",
           "memory_cached", "max_memory_cached", "memory_snapshot", "memory_summary", "list_gpu_processes",
           "mem_get_info", "get_debug_atm", "clear_debug_atm", "get_storageimpl_profile", "clear_storageimpl_profile",
           "prefetch_init", "prefetch_all", "before_prefetch_wait_all", "create_swap_env", "close_swap_env"]

#************************************************************************
#@函数名称:                    get_debug_atm
#@功能描述:                    开发调试接口
#@参数:                        none
#@返回:                        调试输出
#************************************************************************
def get_debug_atm():
    r"""Returns the debug infomation ATM"""
    return torch._C._cuda_getDebugATM()
#************************************************************************
#@函数名称:                    clear_debug_atm
#@功能描述:                    清空调试输出
#@参数:                        none
#@返回:                        0:成功，其他:发生错误
#************************************************************************
```

```
45  def clear_debug_atm():
46      r"""clear the debug infomation ATM"""
47      return torch._C._cuda_clearDebugATM()
48
49  #*****************************************************************************
50  #@函数名称:                  get_storageimpl_profile
51  #@功能描述:                  开发调试接口
52  #@参数:                      none
53  #@返回:                      storage 调试输出
54  #*****************************************************************************
55  def get_storageimpl_profile():
56      r"""Returns the impl profile infomation ATM"""
57      return torch._C._cuda_getStorageImplProfileATM()
58  #*****************************************************************************
59  #@函数名称:                  clear_storageimpl_profile
60  #@功能描述:                  清空调试输出
61  #@参数:                      none
62  #@返回:                      0:成功，其他:发生错误
63  #*****************************************************************************
64  def clear_storageimpl_profile():
65      r"""clear the impl profile infomation ATM"""
66      return torch._C._cuda_clearStorageImplProfileATM()
67
68  #*****************************************************************************
69  #@函数名称:                  create_swap_env
70  #@功能描述:                  初始化数据转移上下文
71  #@参数:                      none
72  #@返回:                      none
73  #*****************************************************************************
74  def create_swap_env():
75      return torch._C._cuda_createSwapEnv()
76  #*****************************************************************************
77  #@函数名称:                  close_swap_env
78  #@功能描述:                  结束数据转移上下文
79  #@参数:                      none
80  #@返回:                      none
81  #*****************************************************************************
82  def close_swap_env():
83      return torch._C._cuda_closeSwapEnv()
84  #*****************************************************************************
85  #@函数名称:                  prefetch_init
86  #@功能描述:                  初始化数据换入队列
87  #@参数:                      none
88  #@返回:                      none
```

```
#*****************************************************************************
def prefetch_init():
    r"""prefetch init"""
    return torch._C._cuda_prefetchInit()
#*****************************************************************************
#@函数名称:                  prefetch_all
#@功能描述:                  执行主动换入操作
#@参数:                      none
#@返回:                      none
#*****************************************************************************
def prefetch_all():
    r"""prefetch all storage"""
    return torch._C._cuda_prefetchAll()
#*****************************************************************************
#@函数名称:                  before_prefetch_wait_all
#@功能描述:                  等待数据换出操作结束
#@参数:                      none
#@返回:                      none
#*****************************************************************************
def before_prefetch_wait_all():
    r"""wait all transfer done"""
    return torch._C._cuda_beforPrefetchWaitAll()
```

文件名: torch/csrc/cuda/Module.cpp (修改)

```
#include <ATen/cuda/Sleep.h>
#include <ATen/cuda/detail/CUDAHooks.h>
#include <ATen/cuda/jiterator.h>
#include <c10/cuda/ATMConfig.h>

PyObject * THCPModule_createSwapEnv(PyObject *_unused, PyObject *noargs)
{
  HANDLE_TH_ERRORS
  c10::cuda::CUDACachingAllocator::createSwapEnv();
  END_HANDLE_TH_ERRORS
  Py_RETURN_NONE;
}
PyObject * THCPModule_closeSwapEnv(PyObject *_unused, PyObject *noargs)
{
  HANDLE_TH_ERRORS
  c10::cuda::CUDACachingAllocator::closeSwapEnv();
  END_HANDLE_TH_ERRORS
  Py_RETURN_NONE;
}
```

```
PyObject * THCPModule_prefetchInit(PyObject *_unused, PyObject *noargs)
{
  HANDLE_TH_ERRORS
  c10::cuda::CUDACachingAllocator::prefetchInit();
  END_HANDLE_TH_ERRORS
  Py_RETURN_NONE;
}

PyObject * THCPModule_prefetchAll(PyObject *_unused, PyObject *noargs)
{
  HANDLE_TH_ERRORS
  c10::cuda::CUDACachingAllocator::prefetchAll();
  END_HANDLE_TH_ERRORS
  Py_RETURN_NONE;
}

PyObject * THCPModule_beforPrefetchWaitAll(PyObject *_unused, PyObject *noargs)
{
  HANDLE_TH_ERRORS
  c10::cuda::CUDACachingAllocator::beforPrefetchWaitAll();
  END_HANDLE_TH_ERRORS
  Py_RETURN_NONE;
}

PyObject *THCPModule_getC10DebugATM(PyObject *_unused, PyObject * arg)
{
  HANDLE_TH_ERRORS
  auto debug_log = c10::cuda::get_debug_log();
  std::string debug_output = "";
  int iter = 0;
  for (auto debug_log_el : debug_log->get_debug(c10::cuda::ATMLogLevel::DEBUG)) {
    debug_output += "[" + std::to_string(++iter) + "]" + debug_log_el.first + "|=>|" +
debug_log_el.second + "\n";
  }
  return THPUtils_packString(debug_output);
  // if (c10::cuda::CUDACachingAllocator::userEnabledLMS()) Py_RETURN_TRUE;
  // else Py_RETURN_FALSE;
  END_HANDLE_TH_ERRORS
}

PyObject *THCPModule_clearC10DebugATM(PyObject *_unused, PyObject * arg)
{
  HANDLE_TH_ERRORS
  c10::cuda::get_debug_log()->clear_debug(c10::cuda::ATMLogLevel::DEBUG);
```

```
177     Py_RETURN_NONE;
178     END_HANDLE_TH_ERRORS
179   }
180
181   PyObject *THCPModule_getC10StorageImplProfileATM(PyObject *_unused, PyObject *
182   arg)
183   {
184     HANDLE_TH_ERRORS
185     auto impl_profile = c10::cuda::get_impl_profile();
186     std::string profile_output = "";
187     int iter = 0;
188     for (auto impl_profile_el : impl_profile->get_storage_profile()) {
189       profile_output += std::to_string(impl_profile_el.first) + "," +
190                         std::to_string(impl_profile_el.second.data_ptr_) + "," +
191                         std::to_string(impl_profile_el.second.life_start_) + "," +
192                         std::to_string(impl_profile_el.second.life_end_) + "," +
193                         std::to_string(impl_profile_el.second.size_);
194       for (auto access_el : impl_profile_el.second.access_seq_)
195         profile_output += "," + std::to_string(access_el);
196       profile_output += "\n";
197     }
198     return THPUtils_packString(profile_output);
199     END_HANDLE_TH_ERRORS
200   }
201
202   PyObject *THCPModule_clearC10StorageImplProfileATM(PyObject *_unused, PyObject
203   * arg)
204   {
205     HANDLE_TH_ERRORS
206     c10::cuda::get_impl_profile()->clear_storage_profile();
207     Py_RETURN_NONE;
208     END_HANDLE_TH_ERRORS
209   }
210
211     {"_cuda_resetAccumulatedMemoryStats",
212   THCPModule_resetAccumulatedMemoryStats, METH_O, nullptr},
213     {"_cuda_resetPeakMemoryStats", THCPModule_resetPeakMemoryStats, METH_O,
214   nullptr},
215     {"_cuda_memorySnapshot", THCPModule_memorySnapshot, METH_NOARGS,
216   nullptr},
217     // NOTE: Add cuda function here
218     {"_cuda_createSwapEnv", THCPModule_createSwapEnv, METH_NOARGS, nullptr},
219     {"_cuda_closeSwapEnv", THCPModule_closeSwapEnv, METH_NOARGS, nullptr},
220     {"_cuda_prefetchInit", THCPModule_prefetchInit, METH_NOARGS, nullptr},
```

```
    {"_cuda_prefetchAll", THCPModule_prefetchAll, METH_NOARGS, nullptr},
    {"_cuda_beforPrefetchWaitAll", THCPModule_beforPrefetchWaitAll, METH_NOARGS,
nullptr},
    {"_cuda_getStorageImplProfileATM", THCPModule_getC10StorageImplProfileATM,
METH_NOARGS, nullptr},
    {"_cuda_clearStorageImplProfileATM", THCPModule_clearC10StorageImplProfileATM,
METH_NOARGS, nullptr},
    {"_cuda_getDebugATM", THCPModule_getC10DebugATM, METH_NOARGS, nullptr},
    {"_cuda_clearDebugATM", THCPModule_clearC10DebugATM, METH_NOARGS,
nullptr},
    {"_cuda_cudaHostAllocator", THCPModule_cudaHostAllocator, METH_NOARGS,
nullptr},
    {"_cuda_cudaCachingAllocator_raw_alloc",
THCPModule_cudaCachingAllocator_raw_alloc, METH_VARARGS, nullptr},
    {"_cuda_cudaCachingAllocator_raw_delete",
THCPModule_cudaCachingAllocator_raw_delete, METH_O, nullptr},
```

文件名: c10/core/EntityStorageImpl.h (新增)

```
#pragma once
#include <c10/core/ATMCommon.h>
#include <c10/core/Allocator.h>
#include <c10/cuda/ATMConfig.h>
#include <c10/util/intrusive_ptr.h>
// #include <c10/core/StorageImpl.h>
#include <mutex>
#include <condition_variable>

namespace c10 {
struct StorageImpl;
struct EntityStorageImpl;

struct EntityStorageRef {
    EntityStorageRef(EntityStorageImpl* impl) :
        impl_(impl) {}
    EntityStorageRef(const EntityStorageRef &impl_ref) :
        impl_(impl_ref.impl_) {}
    std::shared_ptr<EntityStorageImpl> impl_;
};

typedef EntityStorageRef* EntityStorageRef_t;

enum class EntityStorageStat : uint8_t {
    kOnline,   // on   device
```

```
265     kOffline, // off device
266     kTrans,     // on    transfer
267   };
268
269   enum class TransStat : uint8_t {
270     kNone,       // no mission
271     kPgOut,      // on pageout
272     kPgIn        // on pagein
273   };
274
275   struct EntityStorageImpl {
276     // Abstract class. These methods must be defined for a specific implementation (e.g.
277   CUDA)
278     virtual void do_pagein(void* dst, void* src, size_t size, bool sync) = 0;
279     virtual void do_pageout(void* dst, void* src, size_t size, bool sync) = 0;
280     virtual void do_pagein_cb() {
281       std::unique_lock<std::mutex> lock(mutex_);
282       trans_stat_ = TransStat::kNone;
283       entity_stat_ = EntityStorageStat::kOnline;
284     }
285     virtual void do_pageout_cb() {
286       std::unique_lock<std::mutex> lock(mutex_);
287       trans_stat_ = TransStat::kNone;
288       entity_stat_ = EntityStorageStat::kOffline;
289     }
290
291     EntityStorageImpl(StorageImpl* storage, c10::Allocator* host_allocator) :
292       storage_(storage), host_allocator_(host_allocator), dirty_(false),
293       trans_stat_(TransStat::kNone), entity_stat_(EntityStorageStat::kOnline) {
294     }
295
296     EntityStorageImpl() = delete;
297     virtual ~EntityStorageImpl() {}
298
299     void                    release_resources();
300     virtual void        ensure_data() {
301       #ifdef ATM_DEBUG_STORAGE
302       c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
303                                             "EntityStorageImpl::ensure_data", "");
304       #endif
305       ensure_data_internal(true);
306     }
307
308     // StorageImpl accessors defined in StorageImpl.h to avoid circular depencencies
```

```
309     const Allocator* allocator() const;
310     size_t                capacity() const;
311     Device                 device() const;
312     void*                  device_ptr() const;
313     c10::DataPtr        set_device_ptr(c10::DataPtr&& data_ptr);
314     void                   mark_dirty();
315     /*
316     * set synchronize true to use synchronize swapin
317     */
318     virtual void        ensure_data_internal(bool sync) {
319       std::unique_lock<std::mutex> lock(mutex_);
320       std::lock_guard<std::mutex> ensure_lock(ensure_mutex_);
321     }
322     virtual void        prefetch_internal() = 0;
323     // Wait transfer done, you should do understand what you're doing !!!
324     virtual void        unsafe_wait_transfer() = 0;
325
326     virtual void        pageout_internal() { do_pageout_cb(); }
327     virtual void        pagein_internal()   { do_pagein_cb(); }
328     virtual void        need_prefetch_internal() {}
329
330     virtual void        pageout_internal_sync() { }
331     virtual void        pagein_internal_sync()   { }
332
333     uint64_t              id() const { return entity_id_; }
334
335     // Initialized at or soon after construction
336     StorageImpl*       const storage_;
337     c10::Allocator* const host_allocator_;
338     uint64_t                    entity_id_;
339
340     mutable std::mutex       mutex_;
341     mutable std::mutex       ensure_mutex_;
342
343     // Guarded by mutex_
344     c10::DataPtr               host_data_ptr_;
345     bool                        dirty_;
346
347     TransStat                   trans_stat_;
348     EntityStorageStat        entity_stat_;
349   };
350
351
352   namespace cuda {
```

```
} // cuda
} // c10
```

文件名: c10/core/ATMCommon.h (新增)

```
#pragma once
/// ensure data (no ensure)
#define ATM_ENSURE_DATA
/// debug log
// #define ATM_DEBUG_1
/// access pattern log
// #define ATM_DEBUG_2
/// access pattern log (CUDACachingAllocator)
// #define ATM_DEBUG_3
/// access pattern log inside (ATMConfig)
// #define ATM_DEBUG_4
/// atm storage debug code
#define ATM_DEBUG_STORAGE
```

文件名: c10/cuda/ATMConfig.h (新增)

```
#pragma once
// #include <c10/macros/Macros.h>
#include <c10/core/Allocator.h>
#include <c10/core/ATMCommon.h>
// #include <c10/core/TensorImpl.h>
// #include <c10/core/StorageImpl.h>
#include <c10/cuda/CUDAMacros.h>
#include <c10/util/IntrusiveList.h>

#include <mutex>
#include <exception>
#include <map>
#include <vector>
#include <string>
#include <iterator>
#include <cstdint>
#include <chrono>
#include <cstdio>
#include <typeinfo>

// 2^15
#define MAX_LOG_PRESERVED 32768
namespace c10 {
```

```

struct TensorImpl;
struct StorageImpl;

namespace cuda {
// List of [ Calling Function + Debug Log ]
typedef std::vector<std::pair<std::string, std::string>> DebugLogList;

class ATMDebugLog;
class ImplProfile;

C10_CUDA_API ATMDebugLog* get_debug_log();
C10_CUDA_API ImplProfile* get_impl_profile();

class ATMConfig {
  public:
  ATMConfig() = default;
};

enum class ATMLogLevel {
  DEBUG,
  INFO,
  WARNING,
  ERROR
};

class ATMDebugLog {
  public:
  ATMDebugLog() = default;
  void add_debug(const ATMLogLevel level, const std::string &func, const std::string
&info) {
    std::unique_lock<std::mutex> lock(mutex_);
    switch (level) {
      case ATMLogLevel::DEBUG : {
        count_debug_log_++;
        debug_log_.push_back(std::make_pair(func, info));
        if (count_debug_log_ % MAX_LOG_PRESERVED == 0)
handle_log_oom("debug", debug_log_, count_debug_log_);
        break; }
      case ATMLogLevel::INFO : {
        count_info_log_++;
        info_log_.push_back(std::make_pair(func, info));
        if (count_info_log_ % MAX_LOG_PRESERVED == 0) handle_log_oom("info",
info_log_, count_info_log_);
```

```
441             break; }
442         case ATMLogLevel::WARNING : {
443             count_warning_log_++;
444             warning_log_.push_back(std::make_pair(func, info));
445             if (count_warning_log_ % MAX_LOG_PRESERVED == 0)
446 handle_log_oom("warning", warning_log_, count_warning_log_);
447             break; }
448         case ATMLogLevel::ERROR : {
449             count_error_log_++;
450             error_log_.push_back(std::make_pair(func, info));
451             if (count_error_log_ % MAX_LOG_PRESERVED == 0) handle_log_oom("error",
452 error_log_, count_error_log_);
453             break; }
454       }
455
456   }
457   const DebugLogList& get_debug(ATMLogLevel level) const {
458       switch (level) {
459         case ATMLogLevel::DEBUG : return debug_log_;
460         case ATMLogLevel::INFO  : return info_log_;
461         case ATMLogLevel::WARNING : return warning_log_;
462         case ATMLogLevel::ERROR : return error_log_;
463       }
464       return debug_log_;
465   }
466   void clear_debug(ATMLogLevel level) {
467       std::unique_lock<std::mutex> lock(mutex_);
468       switch (level) {
469         case ATMLogLevel::DEBUG : debug_log_.clear();
470                                   break;
471         case ATMLogLevel::INFO  : info_log_.clear();
472                                   break;
473         case ATMLogLevel::WARNING : warning_log_.clear();
474                                   break;
475         case ATMLogLevel::ERROR : error_log_.clear();
476                                   break;
477       }
478   }
479   private:
480   void handle_log_oom(std::string log_name, DebugLogList& log_list, int log_count) {
481       if (log_count % MAX_LOG_PRESERVED) return;
482       log_count -= MAX_LOG_PRESERVED;
483       int iter = 0;
484       FILE* fd = fopen((log_name + ".atm.log").c_str(), "a+");
```

```
485         for (auto log_el : log_list) {
486           std::string debug_output = "[" + std::to_string((++iter) + log_count) + "]" +
487     log_el.first + "|=>|" + log_el.second + "\n";
488           fprintf(fd, "%s", debug_output.c_str());
489         }
490         log_list.clear();
491       }
492
493       std::mutex mutex_;
494       // Guarded by mutex_
495       DebugLogList debug_log_;
496       DebugLogList info_log_;
497       DebugLogList warning_log_;
498       DebugLogList error_log_;
499       int count_debug_log_;
500       int count_info_log_;
501       int count_warning_log_;
502       int count_error_log_;
503     };
504
505     struct ImplProfileEl {
506       uint64_t data_ptr_;
507       int64_t life_start_;
508       int64_t life_end_;
509       uint64_t size_; // in Byte
510       std::vector<int64_t> access_seq_;
511       uint8_t by_operator;
512     };
513     class ImplProfile {
514       public:
515       ImplProfile() = default;
516       void tensorLifeStart(const c10::TensorImpl* tensor_ptr);
517       //     const void *data_ptr = tensor_ptr->data();
518       //     tensor_profile_.insert(
519       //         std::make_pair(reinterpret_cast<uint64_t>(tensor_ptr),
520       //         ImplProfileEl{
521       //             reinterpret_cast<uint64_t>(data_ptr),
522       //
523     std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now().
524     time_since_epoch()).count(),
525       //             0,
526       //             tensor_ptr->storage().nbytes()
527       //         })
528       //     );
```

```
529     //     return;
530     // }
531     void tensorSetStorage(const c10::TensorImpl* tensor_ptr);
532     void tensorLifeEnds(const c10::TensorImpl* tensor_ptr);
533     //     tensor_profile_[reinterpret_cast<uint64_t>(tensor_ptr)].life_end_ =
534     //
535   std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now().
536   time_since_epoch()).count();
537     //     return;
538     // }
539     void storageLifeStart(const c10::StorageImpl* storage_ptr) {
540       std::unique_lock<std::mutex> lock(mutex_, std::try_to_lock);
541       // const void *data_ptr = tensor_ptr->data();
542       storage_profile_.insert(std::make_pair(reinterpret_cast<uint64_t>(storage_ptr),
543         ImplProfileEl{
544           0,
545
546   std::chrono::duration_cast<std::chrono::microseconds>(std::chrono::system_clock::now
547   ().time_since_epoch()).count(),
548           0, 0, {}, 0
549         })
550       );
551       return;
552     }
553     void storageLifeEnds(const c10::StorageImpl* storage_ptr) {
554       std::unique_lock<std::mutex> lock(mutex_, std::try_to_lock);
555       if          (storage_profile_.find(reinterpret_cast<uint64_t>(storage_ptr))          ==
556   storage_profile_.end()) {
557         storage_profile_.insert(std::make_pair(reinterpret_cast<uint64_t>(storage_ptr),
558           ImplProfileEl{ 0, 0, 0, 0, {}, 0}) );
559         #ifdef ATM_DEBUG_4
560         get_debug_log()->add_debug(ATMLogLevel::DEBUG,
561                                    "ImplProfile::storageLifeEnds",
562                                    std::to_string(reinterpret_cast<uint64_t>(storage_ptr))
563   + "Not Found");
564         #endif
565         // return;
566       }
567       storage_profile_[reinterpret_cast<uint64_t>(storage_ptr)].life_end_ =
568
569   std::chrono::duration_cast<std::chrono::microseconds>(std::chrono::system_clock::now
570   ().time_since_epoch()).count();
571     }
572     void  storageSetStorage(const  c10::StorageImpl*  storage_ptr,  void*  data_ptr,  size_t
```

```
size) {
      if            (storage_profile_.find(reinterpret_cast<uint64_t>(storage_ptr))            ==
storage_profile_.end()) {
        #ifdef ATM_DEBUG_4
        get_debug_log()->add_debug(ATMLogLevel::DEBUG,
                                   "ImplProfile::storageSetStorage",
                                   std::to_string(reinterpret_cast<uint64_t>(storage_ptr))
+ "Not Found");
        #endif
        return;
      }
      std::unique_lock<std::mutex> lock(mutex_, std::try_to_lock);
      storage_profile_[reinterpret_cast<uint64_t>(storage_ptr)].data_ptr_                    =
reinterpret_cast<uint64_t>(data_ptr);
      storage_profile_[reinterpret_cast<uint64_t>(storage_ptr)].size_ = size;
    }

    void storageAppendAccess(const c10::StorageImpl* storage_ptr) {
      if            (storage_profile_.find(reinterpret_cast<uint64_t>(storage_ptr))            ==
storage_profile_.end()) {
        storage_profile_.insert(std::make_pair(reinterpret_cast<uint64_t>(storage_ptr),
          ImplProfileEl{ 0, 0, 0, 0, {}, 0}) );
        #ifdef ATM_DEBUG_4
        get_debug_log()->add_debug(ATMLogLevel::DEBUG,
                                   "ImplProfile::storageAppendAccess",
                                   std::to_string(reinterpret_cast<uint64_t>(storage_ptr))
+ "Not Found");
        #endif
        // return;
      }
      std::unique_lock<std::mutex> lock(mutex_, std::try_to_lock);
      storage_profile_[reinterpret_cast<uint64_t>(storage_ptr)].access_seq_.push_back(

std::chrono::duration_cast<std::chrono::microseconds>(std::chrono::system_clock::now
().time_since_epoch()).count()
      );
    }

    void clear_storage_profile() { storage_profile_.clear(); }
    std::map<uint64_t, ImplProfileEl>& get_storage_profile() { return storage_profile_; }
    private:

    std::mutex mutex_;
    // Guarded by mutex_
```

```
    std::map<uint64_t, ImplProfileEl> tensor_profile_;
    std::map<uint64_t, ImplProfileEl> storage_profile_;

};

} // cuda
} // c10
```

文件名: c10/cuda/ATMConfig.cpp (新增)

```
#include <c10/cuda/ATMConfig.h>
namespace c10 { namespace cuda {

// ATMDebugLog debug_logger;
// ATMDebugLog* get_debug_log() { return &debug_logger; }
// ImplProfile impl_profile;
// ImplProfile* get_impl_profile() { return &impl_profile; }

} }
```

文件名: c10/core/StorageImpl.h (修改)

```
#include <c10/util/intrusive_ptr.h>

#include <c10/core/ATMCommon.h>
#include <c10/core/EntityStorageImpl.h>

#include <c10/cuda/ATMConfig.h>

struct C10_API StorageImpl : public c10::intrusive_ptr_target {
  public:
    struct use_byte_size_t {};

    StorageImpl(
          use_byte_size_t /*use_byte_size*/,
          size_t size_bytes,
          at::DataPtr data_ptr,
          at::Allocator* allocator,
          bool resizable)
          : data_ptr_(std::move(data_ptr)),
            size_bytes_(size_bytes),
            resizable_(resizable),
            received_cuda_(false),
            allocator_(allocator),
```

```
661             entity_(allocator ? allocator->as_entity(this) : nullptr) {
662       #ifdef ATM_DEBUG_1
663       c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
664                                             "StorageImpl::constructor",
665                                             "Pre Allocated DataPtr Device: " + device().str());
666       #endif
667       #ifdef ATM_DEBUG_2
668       auto impl_profile_ = c10::cuda::get_impl_profile();
669       impl_profile_->storageLifeStart(this);
670       impl_profile_->storageSetStorage(this, data_ptr_.get(), nbytes());
671       #endif
672       if (resizable) {
673         TORCH_INTERNAL_ASSERT(
674             allocator_, "For resizable storage, allocator must be provided");
675       }
676     }
677
678     StorageImpl(
679         use_byte_size_t /*use_byte_size*/,
680         size_t size_bytes,
681         at::Allocator* allocator,
682         bool resizable)
683         : StorageImpl(
684               use_byte_size_t(),
685               size_bytes,
686               allocator->allocate(size_bytes),
687               allocator,
688               resizable) {
689       #ifdef ATM_DEBUG_1
690       c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
691                                             "StorageImpl::constructor",
692                                             "No Pre Allocated DataPtr Device: " +
693   device().str());
694       #endif
695       #ifdef ATM_DEBUG_2
696       c10::cuda::get_impl_profile()->storageLifeStart(this);
697       #endif
698     }
699
700     StorageImpl& operator=(StorageImpl&& other) = default;
701     StorageImpl& operator=(const StorageImpl&) = delete;
702     StorageImpl() = delete;
703     StorageImpl(StorageImpl&& other) : entity_(other.entity_) {
704       #ifdef ATM_DEBUG_1
```

```
705         c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
706                                               "StorageImpl::constructor",
707                                               "No Pre Allocated (From Other) DataPtr Device:
708     " + device().str());
709         #endif
710         // #ifdef ATM_DEBUG_2
711         // c10::cuda::get_impl_profile()->storageLifeStart(this);
712         // #endif
713       }
714       StorageImpl(const StorageImpl&) = delete;
715       ~StorageImpl() override;
716
717       void reset() {
718         data_ptr_.clear();
719         size_bytes_ = 0;
720       }
721
722       template <typename T>
723       inline T* data() const {
724         #ifdef ATM_ENSURE_DATA
725         if (atm_enabled()) entity_.impl_->ensure_data();
726         #endif
727         #ifdef ATM_DEBUG_1
728         T x_;
729         const char* type_name = typeid(x_).name();
730         c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
731                                               "StorageImpl::data"+       std::string(type_name)
732     +"(const)",
733                                               "Accessed Data");
734         #endif
735         #ifdef ATM_DEBUG_2
736         c10::cuda::get_impl_profile()->storageAppendAccess(this);
737         #endif
738         return unsafe_data<T>();
739       }
740
741       template <typename T>
742       inline T* unsafe_data() const {
743         #ifdef ATM_ENSURE_DATA
744         if (atm_enabled()) entity_.impl_->ensure_data();
745         #endif
746         return static_cast<T*>(this->data_ptr_.get());
747       }
748
```

```
749     void release_resources() override;
750
751     size_t nbytes() const {
752       return size_bytes_;
753     }
754
755     // TODO: remove later
756     void set_nbytes(size_t size_bytes) {
757       size_bytes_ = size_bytes;
758     }
759
760     bool resizable() const {
761       return resizable_;
762     };
763
764     at::DataPtr& data_ptr() {
765       #ifdef ATM_ENSURE_DATA
766       if (atm_enabled()) entity_.impl_->ensure_data();
767       #endif
768       return data_ptr_;
769     };
770
771     const at::DataPtr& data_ptr() const {
772       #ifdef ATM_ENSURE_DATA
773       if (atm_enabled()) entity_.impl_->ensure_data();
774       #endif
775       return data_ptr_;
776     };
777
778     // Returns the previous data_ptr
779     at::DataPtr set_data_ptr(at::DataPtr&& data_ptr) {
780       at::DataPtr old_data_ptr(std::move(data_ptr_));
781       data_ptr_ = std::move(data_ptr);
782       #ifdef ATM_DEBUG_1
783       // printf("Set DataPtr Device: %s\n", device().str().c_str());
784       c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
785                                             "StorageImpl::set_data_ptr",
786                                             "Set DataPtr Device: " + device().str());
787       #endif
788       #ifdef ATM_DEBUG_2
789       c10::cuda::get_impl_profile()->storageSetStorage(this, data_ptr_.get(), nbytes());
790       #endif
791       return old_data_ptr;
792     };
```

```
793
794     void set_data_ptr_noswap(at::DataPtr&& data_ptr) {
795       data_ptr_ = std::move(data_ptr);
796     }
797
798     // TODO: Return const ptr eventually if possible
799     void* data() {
800       #ifdef ATM_ENSURE_DATA
801       if (atm_enabled()) entity_.impl_->ensure_data();
802       #endif
803       #ifdef ATM_DEBUG_1
804       c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
805                                             "StorageImpl::data",
806                                             "Accessed Data");
807       #endif
808       #ifdef ATM_DEBUG_2
809       c10::cuda::get_impl_profile()->storageAppendAccess(this);
810       #endif
811       return data_ptr_.get();
812     }
813
814     void* data() const {
815       #ifdef ATM_ENSURE_DATA
816       if (atm_enabled()) entity_.impl_->ensure_data();
817       #endif
818       #ifdef ATM_DEBUG_1
819       c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
820                                             "StorageImpl::data_ptr(const)",
821                                             "Accessed Data");
822       #endif
823       #ifdef ATM_DEBUG_2
824       c10::cuda::get_impl_profile()->storageAppendAccess(this);
825       #endif
826       return data_ptr_.get();
827     }
828
829     at::DeviceType device_type() const {
830       return data_ptr_.device().type();
831     }
832
833     at::Allocator* allocator() {
834       #ifdef ATM_DEBUG_1
835       // printf("Used Allocator Once\n");
836       c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
```

```
                                                "StorageImpl::allocator",
                                                "Used Allocator Once");
    #endif
    return allocator_;
  }

  const at::Allocator* allocator() const {
    #ifdef ATM_DEBUG_1
    c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
                                                "StorageImpl::allocator(const)",
                                                "Used Allocator Once");
    #endif
    return allocator_;
  };

  // You generally shouldn't use this method, but it is occasionally
  // useful if you want to override how a tensor will be reallocated,
  // after it was already allocated (and its initial allocator was
  // set)
  void set_allocator(at::Allocator* allocator) {
    allocator_ = allocator;
  }

  Device device() const {
    return data_ptr_.device();
  }

  void set_resizable(bool resizable) {
    if (resizable) {
      // We need an allocator to be resizable
      AT_ASSERT(allocator_);
    }
    resizable_ = resizable;
  }

  /**
   * Can only be called when use_count is 1
   */
  void UniqueStorageShareExternalPointer(
      void* src,
      size_t size_bytes,
      DeleterFnPtr d = nullptr) {
    UniqueStorageShareExternalPointer(
        at::DataPtr(src, src, d, data_ptr_.device()), size_bytes);
```

```
881       }
882
883       /**
884        * Can only be called when use_count is 1
885        */
886       void UniqueStorageShareExternalPointer(
887             at::DataPtr&& data_ptr,
888             size_t size_bytes) {
889          data_ptr_ = std::move(data_ptr);
890          size_bytes_ = size_bytes;
891          allocator_ = nullptr;
892          resizable_ = false;
893       }
894
895       // This method can be used only after storage construction and cannot be used
896       // to modify storage status
897       void set_received_cuda(bool received_cuda) {
898          received_cuda_ = received_cuda;
899       }
900
901       bool received_cuda() {
902          return received_cuda_;
903       }
904
905       // manual method should be deprecated other than debug
906       void pageout_manual();
907       void pagein_manual();
908       void need_prefetch();
909
910       bool atm_enabled() const { return entity_.impl_.get() != nullptr; }
911
912       EntityStorageRef& entity() {
913          return entity_;
914       }
915
916       DataPtr data_ptr_;
917      private:
918       size_t size_bytes_;
919       bool resizable_;
920       // Identifies that Storage was received from another process and doesn't have
921       // local to process cuda memory allocation
922       bool received_cuda_;
923
924       Allocator* allocator_;
```

```
925
926     EntityStorageRef entity_;
927   };
928
929   inline const c10::Allocator* EntityStorageImpl::allocator() const {
930     return storage_->allocator();
931   }
932   inline size_t EntityStorageImpl::capacity() const {
933     return storage_->nbytes();
934   }
935   inline void* EntityStorageImpl::device_ptr() const {
936     return storage_->data_ptr_.get();
937   }
938   inline c10::Device EntityStorageImpl::device() const {
939     return storage_->device();
940   }
941   inline c10::DataPtr EntityStorageImpl::set_device_ptr(c10::DataPtr&& data_ptr) {
942     std::swap(storage_->data_ptr_, data_ptr);
943     return std::move(data_ptr);
944   }
945
946   文件名: c10/core/StorageImpl.cpp (修改)
947
948   #include <c10/core/StorageImpl.h>
949   #include <c10/util/Exception.h>
950   #include <ATen/cuda/CachingHostAllocator.h>
951   namespace c10 {
952   StorageImpl::~StorageImpl(){
953     if (atm_enabled())
954       entity_.impl_->mark_dirty();
955   }
956   void StorageImpl::release_resources() {
957     #ifdef ATM_DEBUG_2
958     c10::cuda::get_impl_profile()->storageLifeEnds(this);
959     #endif
960     if (atm_enabled())
961       entity_.impl_->mark_dirty();
962     data_ptr_.clear();
963   }
964   void StorageImpl::pagein_manual() {
965     #ifdef ATM_DEBUG_STORAGE
966     c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
967                                           "StorageImpl::pagein_manual", "");
968     #endif
```

```
    if (atm_enabled()) {
        entity_.impl_->pagein_internal();
    }
}
void StorageImpl::pageout_manual() {
    #ifdef ATM_DEBUG_STORAGE
    c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
                                          "StorageImpl::pageout_manual", "");
    #endif
    if (atm_enabled())
        entity_.impl_->pageout_internal();
}
void StorageImpl::need_prefetch() {
    #ifdef ATM_DEBUG_STORAGE
    c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
                                          "StorageImpl::need_prefetch", "");
    #endif
    if (atm_enabled()) {
        entity_.impl_->need_prefetch_internal();
    }
}
void EntityStorageImpl::release_resources() {}
void EntityStorageImpl::mark_dirty() {
    std::lock_guard<std::mutex> lock(mutex_);
    dirty_ = true;
}
}
```

文件名: c10/cuda/CUDAStream.h (修改)

```
/**
 * Get a new stream from the CUDA stream pool for Automem.
 */
TORCH_API CUDAStream getCustomCUDAStream(DeviceIndex device = -1);

C10_API std::ostream& operator<<(std::ostream& stream, const CUDAStream& s);
```

文件名: c10/cuda/CUDAStream.cpp (修改)

```
static std::once_flag device_flags[C10_COMPILE_TIME_MAX_GPUS];
static std::atomic<uint32_t> low_priority_counters[C10_COMPILE_TIME_MAX_GPUS];
static std::atomic<uint32_t> high_priority_counters[C10_COMPILE_TIME_MAX_GPUS];
static cudaStream_t low_priority_streams[C10_COMPILE_TIME_MAX_GPUS]
                                        [kStreamsPerPool];
```

```
1013    static cudaStream_t high_priority_streams[C10_COMPILE_TIME_MAX_GPUS]
1014                                                 [kStreamsPerPool];
1015
1016
1017    // ATM streams
1018    static constexpr unsigned int kAutoMemFlags = cudaStreamNonBlocking;
1019    static std::once_flag atm_device_flags[C10_COMPILE_TIME_MAX_GPUS];
1020    static std::atomic<uint32_t> atm_counters[C10_COMPILE_TIME_MAX_GPUS];
1021    static cudaStream_t atm_streams[C10_COMPILE_TIME_MAX_GPUS][kStreamsPerPool];
1022
1023    std::ostream& operator<<(std::ostream& stream, StreamIdType s) {
1024       switch (s) {
1025          case StreamIdType::DEFAULT:
1026             stream << "DEFAULT";
1027             break;
1028          case StreamIdType::LOW:
1029             stream << "LOW";
1030             break;
1031          case StreamIdType::HIGH:
1032             stream << "HIGH";
1033             break;
1034          case StreamIdType::EXT:
1035             stream << "EXT";
1036             break;
1037          case StreamIdType::ATM:
1038             stream << "ATM";
1039             break;
1040          default:
1041             stream << static_cast<uint8_t>(s);
1042             break;
1043       }
1044       return stream;
1045    }
1046
1047    // Creates the ATM stream pools for the specified device
1048    // Warning: only call once per device!
1049    static void initDeviceAutoMemStreamState(DeviceIndex device_index) {
1050       // Switches to the requested device so streams are properly associated
1051       // with it.
1052       CUDAGuard device_guard{device_index};
1053
1054       for (const auto i : c10::irange(kStreamsPerPool)) {
1055          auto& stream = atm_streams[device_index][i];
1056
```

```
1057          C10_CUDA_CHECK(cudaStreamCreateWithFlags(&stream, kAutoMemFlags));
1058        }
1059        atm_counters[device_index] = 0;
1060      }
1061
1062      static void initAutoMemStreamsOnce(DeviceIndex device_index) {
1063        // Inits default streams (once, globally)
1064        std::call_once(atm_device_flags[device_index],           initDeviceAutoMemStreamState,
1065      device_index);
1066      }
1067      // See Note [StreamId assignment]
1068      cudaStream_t CUDAStream::stream() const {
1069        c10::DeviceIndex device_index = stream_.device_index();
1070        StreamId stream_id = stream_.id();
1071        StreamIdType st = streamIdType(stream_id);
1072        size_t si = streamIdIndex(stream_id);
1073        switch (st) {
1074          case StreamIdType::DEFAULT:
1075            TORCH_INTERNAL_ASSERT(
1076                si == 0,
1077                "Unrecognized stream ",
1078                stream_,
1079                " (I think this should be the default stream, but I got a non-zero index ",
1080                si,
1081                ").",
1082                " Did you manufacture the StreamId yourself?  Don't do that; use the",
1083                " official API like c10::cuda::getStreamFromPool() to get a new stream.");
1084            return nullptr;
1085          case StreamIdType::LOW:
1086            return low_priority_streams[device_index][si];
1087          case StreamIdType::HIGH:
1088            return high_priority_streams[device_index][si];
1089          case StreamIdType::EXT:
1090            return reinterpret_cast<cudaStream_t>(stream_id);
1091          case StreamIdType::ATM:
1092            return atm_streams[device_index][si];
1093          default:
1094            TORCH_INTERNAL_ASSERT(
1095                0,
1096                "Unrecognized stream ",
1097                stream_,
1098                " (I didn't recognize the stream type, ",
1099                st,
1100                ")");
```

```
    }
  }

  CUDAStream getCustomCUDAStream(DeviceIndex device_index) {
    initCUDAStreamsOnce();
    if (device_index == -1)
      device_index = current_device();
    check_gpu(device_index);

    initAutoMemStreamsOnce(device_index);
    const auto stream_id = get_idx(atm_counters[device_index]);
    return CUDAStream(
        CUDAStream::UNCHECKED,
        Stream(
            Stream::UNSAFE,
            c10::Device(DeviceType::CUDA, device_index),
            makeStreamId(StreamIdType::ATM, stream_id)));
  }
```

文件名: c10/cuda/CUDACachingAllocator.cpp (修改)

```
  #include <c10/util/irange.h>
  #include <c10/util/llvmMathExtras.h>

  #include <c10/cuda/ATMConfig.h>
  #include <c10/cuda/CUDASwapQueues.h>
  #include <c10/core/ATMCommon.h>
  #include <c10/core/StorageImpl.h>

  #include <cuda_runtime_api.h>
  #include <algorithm>
  #include <bitset>

  #define CUDA_INVALID_STREAM ((cudaStream_t)-1)

  struct EntityContext {
    EntityContext() :
          limit_(0), host_allocator_(nullptr),
          stream_in_(CUDA_INVALID_STREAM),  stream_out_(CUDA_INVALID_STREAM)
  {}

    size_t              limit() { return limit_; }
    void                 set_limit(size_t limit) { limit_ = limit; }

```

```
1145     c10::Allocator* host_allocator() { return host_allocator_; }
1146     void                    set_host_allocator(c10::Allocator* host_allocator) { host_allocator_ =
1147   host_allocator; }
1148
1149     cudaStream_t       stream_in() { return stream_in_; }
1150     cudaStream_t       stream_out() { return stream_out_; }
1151     void                    set_streams(cudaStream_t out, cudaStream_t in) {
1152       if (stream_out_ == CUDA_INVALID_STREAM) {
1153         TORCH_INTERNAL_ASSERT(out != CUDA_INVALID_STREAM);
1154         stream_out_ = out; stream_in_    = in;
1155       }
1156     }
1157
1158     size_t                device_limit(int64_t current, int device) {
1159       size_t device_limit = limit_;
1160       if (device_limit == 0) {
1161         size_t available;
1162         size_t capacity;
1163         C10_CUDA_CHECK(cudaMemGetInfo(&available, &capacity));
1164         // Reserve five percent of available memory for non-tensor uses
1165         device_limit = static_cast<size_t>((available + current) * 0.95);
1166       }
1167       return device_limit;
1168     }
1169
1170     size_t                    limit_;
1171     at::Allocator*        host_allocator_;
1172     cudaStream_t           stream_in_;
1173     cudaStream_t           stream_out_;
1174   };
1175   static EntityContext entity_context;
1176
1177   // cuda-entity guarded by mutex
1178   static uint64_t get_cuda_entity_uid() {
1179     static std::mutex uid_count_mutex;
1180     static uint64_t     cuda_entity_uid_count = 0;
1181     std::lock_guard<std::mutex> lock(uid_count_mutex);
1182     return cuda_entity_uid_count++;
1183   }
1184
1185   void  CUDART_CB  __do_pageout_cb(cudaStream_t  stream,  cudaError_t  status,  void
1186   *data);
1187   void  CUDART_CB  __do_pagein_cb(cudaStream_t  stream,  cudaError_t  status,  void
1188   *data);
```

```
1189
1190    struct CudaEntityStorageImpl : public c10::EntityStorageImpl {
1191      CudaEntityStorageImpl(c10::StorageImpl* storage) :
1192        c10::EntityStorageImpl(storage, entity_context.host_allocator()),
1193        block_(nullptr),
1194        pageout_stream_(CUDA_INVALID_STREAM),
1195        pagein_stream_ (CUDA_INVALID_STREAM),
1196        on_prefetch_(false),
1197        need_prefetch_(false) {
1198        entity_id_ = get_cuda_entity_uid();
1199      }
1200      ~CudaEntityStorageImpl() { }
1201
1202      void                set_block(Block* block) { block_ = block; }
1203      Block*              block() const { return block_; }
1204
1205      void                assign_streams(cudaStream_t out, cudaStream_t in) {
1206        if (pageout_stream_ == CUDA_INVALID_STREAM) {
1207          TORCH_INTERNAL_ASSERT(out != CUDA_INVALID_STREAM);
1208          pageout_stream_ = out;
1209          pagein_stream_   = in;
1210          compute_stream_ = block_->stream;
1211        }
1212      }
1213
1214      void                do_pageout_cb() override {
1215        std::unique_lock<std::mutex> lock(mutex_);
1216        lock.unlock();
1217        pgoutcb_cv_.notify_all();
1218      }
1219      void                do_pagein_cb() override {
1220        std::unique_lock<std::mutex> lock(mutex_);
1221        lock.unlock();
1222        pgincb_cv_.notify_all();
1223      }
1224
1225      void                do_pageout(void* dst, void* src, size_t size, bool sync) override;
1226      void                do_pagein(void* dst, void* src, size_t size, bool sync) override;
1227
1228      cudaStream_t       swap(void* dst, const void* src, size_t size, enum cudaMemcpyKind
1229    kind, cudaStream_t stream) {
1230        TORCH_INTERNAL_ASSERT(stream != CUDA_INVALID_STREAM);
1231        cudaEvent_t event = create_event();
1232        // Synchronize swap stream with compute stream
```

```
1233        if(kind == cudaMemcpyDeviceToHost) {
1234          C10_CUDA_CHECK(cudaEventRecord(event, compute_stream_));
1235          C10_CUDA_CHECK(cudaStreamWaitEvent(stream, event, 0));
1236        }
1237        // Queue copy
1238        C10_CUDA_CHECK(cudaMemcpyAsync(dst, src, size, kind, stream));
1239        // Record event to wait on copy completion
1240        C10_CUDA_CHECK(cudaEventRecord(event, stream));
1241        event_ = event;
1242        return stream;
1243      }
1244      void                pageout_internal() override {
1245        CudaEntityEvictQueue::get_evict_queue().enqueue(this);
1246      }
1247      // enqueue prefetch queue, do fetch later
1248      void                pagein_internal() override {
1249        CudaEntityFetchQueue::get_fetch_queue().enqueue(this);
1250      }
1251
1252      void                pageout_internal_sync() override;
1253      void                pagein_internal_sync() override;
1254
1255      void                ensure_data() override {
1256        ensure_data_internal(true /* reserved */);
1257      }
1258      // synchronize (true/false) ensure data
1259      void                ensure_data_internal(bool sync) override;
1260      // enqueue prefetch queue, do prefetch later
1261      void                need_prefetch_internal() override {
1262        CudaEntityFetchQueue::get_fetch_queue().enqueue(this);
1263      }
1264      void                prefetch_internal() override { }
1265      void                unsafe_wait_transfer() override { }
1266
1267      cudaEvent_t         create_event();
1268
1269      Block*                      block_; // cache block pointer for use while on reclaim list
1270      cudaStream_t                compute_stream_;
1271      cudaStream_t                pageout_stream_;
1272      cudaStream_t                pagein_stream_;
1273      cudaEvent_t                 event_;
1274
1275      std::condition_variable pgincb_cv_;
1276      std::condition_variable pgoutcb_cv_;
```

```
    // guarded by mutex
    bool                        on_prefetch_;
    bool                        need_prefetch_;
};

void CUDART_CB __do_pageout_cb(cudaStream_t stream, cudaError_t status, void
*data) {
    C10_CUDA_CHECK(status);
    EntityStorageRef_t impl = reinterpret_cast<EntityStorageRef_t>(data);
    impl->impl_->do_pageout_cb();
    delete impl; // must delete here, or entity leaks
}
void CUDART_CB __do_pagein_cb(cudaStream_t stream, cudaError_t status, void *data)
{
    C10_CUDA_CHECK(status);
    EntityStorageRef_t impl = reinterpret_cast<EntityStorageRef_t>(data);
    impl->impl_->do_pagein_cb();
    delete impl;    // must delete here, or entity leaks
}

class CachingAllocatorConfig {
  public:
    static size_t max_split_size() {
        stats.max_split_size = CachingAllocatorConfig::max_split_size();
    }
...
    cudaEvent_t create_event() {
        std::lock_guard<std::recursive_mutex> lock(mutex);
        return create_event_internal();
    }
...
    void init(int device_count, c10::Allocator* host_allocator) {
        const auto size = static_cast<int64_t>(device_allocator.size());
        if (size < device_count) {
            device_allocator.resize(device_count);
                device_allocator[i] = std::make_unique<DeviceCachingAllocator>();
            }
        }
        entity_context.set_host_allocator(host_allocator);
        entity_context.set_streams(cuda::getCustomCUDAStream().stream(),
cuda::getCustomCUDAStream().stream());
    }
...
};
```

```
1321
1322    void CudaEntityStorageImpl::pageout_internal_sync() {
1323      std::lock_guard<std::mutex> lock(mutex_);
1324      TORCH_INTERNAL_ASSERT(entity_stat_ == EntityStorageStat::kOnline);
1325      TORCH_INTERNAL_ASSERT(trans_stat_ == TransStat::kNone);
1326      set_block(caching_allocator.get_allocated_block(device_ptr()));
1327      if (block_ == nullptr) return;
1328      assign_streams(entity_context.stream_out(), entity_context.stream_in());
1329      size_t size = capacity();
1330      void* dst = host_data_ptr_.get();
1331      if (!dst) {
1332        host_data_ptr_ = host_allocator_->allocate(size);
1333        dst = host_data_ptr_.get();
1334      }
1335      entity_stat_ = EntityStorageStat::kTrans;
1336      trans_stat_ = TransStat::kPgOut;
1337      do_pageout(dst, device_ptr(), size, true);
1338      auto old_device_ptr = set_device_ptr(at::DataPtr(nullptr, device()));
1339      old_device_ptr.clear(); // Fxxk LMS :-(
1340      entity_stat_ = EntityStorageStat::kOffline;
1341      trans_stat_ = TransStat::kNone;
1342    }
1343
1344    void CudaEntityStorageImpl::pagein_internal_sync() {
1345      std::lock_guard<std::mutex> lock(mutex_);
1346      if (entity_stat_ == EntityStorageStat::kOnline && trans_stat_ == TransStat::kNone
1347        || entity_stat_ == EntityStorageStat::kTrans && trans_stat_ == TransStat::kPgIn)
1348        return;
1349      TORCH_INTERNAL_ASSERT(entity_stat_ == EntityStorageStat::kOffline);
1350      TORCH_INTERNAL_ASSERT(trans_stat_ == TransStat::kNone);
1351      size_t size = capacity();
1352      trans_stat_ = TransStat::kPgIn;
1353      entity_stat_ = EntityStorageStat::kTrans;
1354
1355      auto dst = allocator()->allocate(size);
1356      do_pagein(dst.get(), host_data_ptr_.get(), size, true);
1357      // must do move after do_pagein
1358      set_device_ptr(std::move(dst));
1359      trans_stat_ = TransStat::kNone;
1360      entity_stat_ = EntityStorageStat::kOnline;
1361    }
1362
1363    void CudaEntityStorageImpl::ensure_data_internal(bool __reserved__) {
1364      std::lock_guard<std::mutex> ensure_lock(ensure_mutex_);
```

```
1365     std::unique_lock<std::mutex> lock(mutex_);
1366     switch (entity_stat_) {
1367       case EntityStorageStat::kOnline     : return;
1368       case EntityStorageStat::kOffline    : {
1369         CudaEntityFetchQueue::get_fetch_queue().enqueue_front(this);
1370         pgincb_cv_.wait(lock);
1371         return;
1372       }
1373       case EntityStorageStat::kTrans      : {
1374         if (trans_stat_ == TransStat::kPgIn) {
1375           pgincb_cv_.wait(lock);
1376         } else if (trans_stat_ == TransStat::kPgOut) {
1377           pgoutcb_cv_.wait(lock);
1378           CudaEntityFetchQueue::get_fetch_queue().enqueue_front(this);
1379           pgincb_cv_.wait(lock);
1380         }
1381         return;
1382       } }
1383   }
1384
1385   inline cudaEvent_t CudaEntityStorageImpl::create_event() {
1386     return caching_allocator.device_allocator[device().index()]->create_event();
1387   }
1388
1389   void CudaEntityStorageImpl::do_pageout(void* dst, void* src, size_t size, bool sync) {
1390     #ifdef ATM_DEBUG_STORAGE
1391     c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
1392                                           "CudaEntityStorageImpl::do_pageout",
1393                                           "CUDA Pageout Memory" + std::to_string(size));
1394     #endif
1395
1396     swap(dst, src, size, cudaMemcpyDeviceToHost, pageout_stream_);
1397
1398     if (!sync) {
1399       EntityStorageRef_t entity_ptr = new EntityStorageRef(this->storage_->entity());
1400       C10_CUDA_CHECK(cudaStreamAddCallback(pageout_stream_,  __do_pageout_cb,
1401   (void*)entity_ptr, 0));
1402     } else {
1403       C10_CUDA_CHECK(cudaEventSynchronize(event_));
1404       #ifdef ATM_DEBUG_STORAGE
1405       c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
1406                                             "CudaEntityStorageImpl::do_pageout",
1407                                             "Done CUDA Pageout Memory" +
1408   std::to_string(size));
```

```
1409        #endif
1410      }
1411    }
1412
1413    void CudaEntityStorageImpl::do_pagein(void* dst, void* src, size_t size, bool sync) {
1414      #ifdef ATM_DEBUG_STORAGE
1415      c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
1416                                            "CudaEntityStorageImpl::do_pagein",
1417                                            "CUDA Pagein Memory" + std::to_string(size));
1418      #endif
1419      block_ = caching_allocator.get_allocated_block(dst);
1420      pagein_stream_ = entity_context.stream_in();
1421      swap(dst, src, size, cudaMemcpyHostToDevice, pagein_stream_);
1422
1423      if (!sync) {
1424        EntityStorageRef_t entity_ptr = new EntityStorageRef(this->storage_->entity());
1425        C10_CUDA_CHECK(cudaStreamAddCallback(pagein_stream_,      __do_pagein_cb,
1426    (void*)entity_ptr, 0));
1427      } else {
1428        C10_CUDA_CHECK(cudaEventSynchronize(event_));
1429        #ifdef ATM_DEBUG_STORAGE
1430        c10::cuda::get_debug_log()->add_debug(c10::cuda::ATMLogLevel::DEBUG,
1431                                              "CudaEntityStorageImpl::do_pagein",
1432                                              "Done      CUDA      Pagein      Memory"      +
1433    std::to_string(size));
1434        #endif
1435      }
1436    }
1437
1438    struct CudaCachingAllocator : public Allocator {
1439    ...
1440      c10::EntityStorageImpl* as_entity(c10::StorageImpl* storage) {
1441        return new CudaEntityStorageImpl(storage);
1442      }
1443    };
1444
1445    void init(int device_count, c10::Allocator* host_allocator) {
1446      caching_allocator.init(device_count, host_allocator);
1447    }
1448
1449    // cuda entity methods
1450    void createSwapEnv() {
1451      CudaEntityEvictQueue::get_evict_queue().start_actions();
1452      CudaEntityFetchQueue::get_fetch_queue().enable_queue();
```

```
}
void closeSwapEnv() {
   CudaEntityEvictQueue::get_evict_queue().wait_and_stop_actions();
   CudaEntityFetchQueue::get_fetch_queue().wait_and_stop_actions();
}
// clear prefetch queue
void prefetchInit() {
   CudaEntityFetchQueue::get_fetch_queue().wait_and_stop_actions();
   CudaEntityFetchQueue::get_fetch_queue().enable_queue();
}
void prefetchAll() {
   beforPrefetchWaitAll();
   CudaEntityFetchQueue::get_fetch_queue().start_actions();
}
void beforPrefetchWaitAll() {
   CudaEntityEvictQueue::get_evict_queue().wait_actions();
   CudaEntityFetchQueue::get_fetch_queue().wait_actions();
}
```

文件名: c10/cuda/CUDASwapQueues.h (新增)

```
#pragma once
#include <c10/cuda/CUDAException.h>
#include <c10/cuda/CUDAFunctions.h>
#include <c10/cuda/CUDAGuard.h>
#include <c10/util/UniqueVoidPtr.h>
#include <c10/util/flat_hash_map.h>
#include <c10/util/irange.h>
#include <c10/util/llvmMathExtras.h>

#include <c10/core/EntityStorageImpl.h>
#include <c10/core/StorageImpl.h>

#include <mutex>
#include <deque>
#include <thread>

namespace c10 {
namespace cuda {
struct CudaEntityTransferQueue {
  public:
   CudaEntityTransferQueue() :
            enable_flag_(false),
            active_flag_(false),
```

```
                unique_flag_(false) {}
    void                    enqueue(EntityStorageImpl* impl);
    int                     erase(EntityStorageImpl* impl);
    EntityStorageRef_t dequeue();

    virtual void          start_actions() = 0;
    virtual void          wait_and_stop_actions() = 0;
    virtual void          wait_actions() = 0;

  protected:
    std::mutex                          action_mutex_;
    // guarded by action_mutex
    std::deque<EntityStorageRef_t> actions_;
    std::atomic_bool                    enable_flag_;
    std::atomic_bool                    active_flag_;
    std::atomic_bool                    unique_flag_;

    std::condition_variable           not_empty_cv_;
    std::condition_variable           empty_cv_;
  };

  struct CudaEntityEvictQueue final : public CudaEntityTransferQueue {
   public:
    CudaEntityEvictQueue() = default;

    static CudaEntityEvictQueue& get_evict_queue();

    void                    start_actions() override;
    void                    wait_and_stop_actions() override;
    void                    wait_actions() override;

   private:
    static void thread_do_entity_evict(CudaEntityEvictQueue& evict_queue);
    std::thread thread_do_entity_evict_;
  };

  struct CudaEntityFetchQueue final : public CudaEntityTransferQueue {
   public:
    CudaEntityFetchQueue() = default;

    static CudaEntityFetchQueue& get_fetch_queue();

    void                    enqueue_front(EntityStorageImpl* impl);
```

```
    void                    enable_queue();
    void                    start_actions() override;
    void                    wait_and_stop_actions() override;
    void                    wait_actions() override;
  private:
    static void thread_do_entity_fetch(CudaEntityFetchQueue& fetch_queue);
    std::thread thread_do_entity_fetch_;
};

} // namespace cuda
} // namespace c10
```

文件名: c10/cuda/CUDASwapQueues.cpp (新增)

```
#include <c10/cuda/CUDASwapQueues.h>

namespace c10 {
namespace cuda {

void CudaEntityTransferQueue::enqueue(EntityStorageImpl* impl)
{
    std::unique_lock<std::mutex> lock(action_mutex_);
    if (enable_flag_) {
        actions_.emplace_back(new EntityStorageRef(impl->storage_->entity()));
        if (active_flag_) {
            lock.unlock(); not_empty_cv_.notify_all();
        }
    }
}

EntityStorageRef_t CudaEntityTransferQueue::dequeue()
{
    std::lock_guard<std::mutex> lock(action_mutex_);
    if (actions_.empty())
        return nullptr;
    auto impl_ref = actions_.front();
    actions_.pop_front();
    return impl_ref;
}

int CudaEntityTransferQueue::erase(EntityStorageImpl* impl)
{
    std::lock_guard<std::mutex> lock(action_mutex_);
    for (auto i = actions_.begin(); i != actions_.end(); i++)
```

```
1585          if ((*i)->impl_->entity_id_ == impl->entity_id_) {
1586             actions_.erase(i); return 0;
1587          }
1588       return 1;
1589    }
1590
1591    CudaEntityEvictQueue& CudaEntityEvictQueue::get_evict_queue()
1592    {
1593       static CudaEntityEvictQueue evict_queue_;
1594       return evict_queue_;
1595    }
1596
1597    void                  CudaEntityEvictQueue::thread_do_entity_evict(CudaEntityEvictQueue&
1598    evict_queue)
1599    {
1600       std::unique_lock<std::mutex> lock(evict_queue.action_mutex_);
1601       // unique working thread allowed
1602       if (evict_queue.unique_flag_) return;
1603          else evict_queue.unique_flag_ = true;
1604       while (true) {
1605          lock.unlock();
1606          auto impl_ref = evict_queue.dequeue();
1607          lock.lock();
1608          while (impl_ref == nullptr && evict_queue.actions_.empty()) {
1609             evict_queue.empty_cv_.notify_all();
1610             evict_queue.not_empty_cv_.wait(lock);
1611             lock.unlock();
1612             impl_ref = evict_queue.dequeue();
1613             lock.lock();
1614             if (!evict_queue.active_flag_) goto post_evict_thread;
1615          }
1616          lock.unlock();
1617          if (impl_ref->impl_.use_count() > 1 && !impl_ref->impl_->dirty_) {
1618             impl_ref->impl_->pageout_internal_sync();
1619             impl_ref->impl_->do_pageout_cb();
1620          }
1621          delete impl_ref;
1622          lock.lock();
1623       }
1624       post_evict_thread:
1625       // allow new unique thread to create
1626       evict_queue.unique_flag_ = false;
1627    }
1628
```

```
1629    void CudaEntityEvictQueue::start_actions()
1630    {
1631        std::lock_guard<std::mutex> lock(action_mutex_);
1632        if (active_flag_ || enable_flag_ || unique_flag_) return;
1633        active_flag_ = true;
1634        enable_flag_ = true;
1635        thread_do_entity_evict_ = std::thread(thread_do_entity_evict, std::ref(*this));
1636        thread_do_entity_evict_.detach();
1637    }
1638
1639    void CudaEntityEvictQueue::wait_and_stop_actions()
1640    {
1641        std::unique_lock<std::mutex> lock(action_mutex_);
1642        enable_flag_ = false;
1643        if (!active_flag_) return;
1644        // there's running working thread, wait
1645        if (!actions_.empty()) empty_cv_.wait(lock);
1646        active_flag_ = false;
1647        lock.unlock();
1648        // must be a working thread wait not_empty_cv
1649        not_empty_cv_.notify_all();
1650    }
1651
1652    void CudaEntityEvictQueue::wait_actions()
1653    {
1654        std::unique_lock<std::mutex> lock(action_mutex_);
1655        if (!active_flag_) return;
1656        enable_flag_ = false;
1657        if (!actions_.empty()) empty_cv_.wait(lock);
1658        enable_flag_ = true;
1659    }
1660
1661
1662    void CudaEntityFetchQueue::enqueue_front(EntityStorageImpl* impl)
1663    {
1664        std::unique_lock<std::mutex> lock(action_mutex_);
1665        if (enable_flag_) {
1666            actions_.emplace_front(new EntityStorageRef(impl->storage_->entity()));
1667            if (active_flag_) {
1668                lock.unlock(); not_empty_cv_.notify_all();
1669            }
1670        }
1671    }
1672
```

```
1673  CudaEntityFetchQueue& CudaEntityFetchQueue::get_fetch_queue()
1674  {
1675     static CudaEntityFetchQueue fetch_queue_;
1676     return fetch_queue_;
1677  }
1678
1679  void                CudaEntityFetchQueue::thread_do_entity_fetch(CudaEntityFetchQueue&
1680  fetch_queue)
1681  {
1682     std::unique_lock<std::mutex> lock(fetch_queue.action_mutex_);
1683     // unique working thread allowed
1684     if (fetch_queue.unique_flag_) return;
1685        else fetch_queue.unique_flag_ = true;
1686     while (true) {
1687        lock.unlock();
1688        auto impl_ref = fetch_queue.dequeue();
1689        lock.lock();
1690        while (impl_ref == nullptr && fetch_queue.actions_.empty()) {
1691           fetch_queue.empty_cv_.notify_all();
1692           fetch_queue.not_empty_cv_.wait(lock);
1693           lock.unlock();
1694           impl_ref = fetch_queue.dequeue();
1695           lock.lock();
1696           if (!fetch_queue.active_flag_) goto post_fetch_thread;
1697        }
1698        lock.unlock();
1699        if (impl_ref->impl_.use_count() > 1 && !impl_ref->impl_->dirty_) {
1700           impl_ref->impl_->pagein_internal_sync();
1701           impl_ref->impl_->do_pagein_cb();
1702        }
1703        delete impl_ref;
1704        lock.lock();
1705     }
1706     post_fetch_thread:
1707     // allow new unique thread to create
1708     fetch_queue.unique_flag_ = false;
1709  }
1710
1711  void CudaEntityFetchQueue::enable_queue()
1712  {
1713     std::lock_guard<std::mutex> lock(action_mutex_);
1714     enable_flag_ = true;
1715  }
1716
```

```
1717  void CudaEntityFetchQueue::start_actions()
1718  {
1719     std::lock_guard<std::mutex> lock(action_mutex_);
1720     if (active_flag_ || unique_flag_) return;
1721     active_flag_ = true;
1722     thread_do_entity_fetch_ = std::thread(thread_do_entity_fetch, std::ref(*this));
1723     thread_do_entity_fetch_.detach();
1724  }
1725
1726  void CudaEntityFetchQueue::wait_and_stop_actions()
1727  {
1728     std::unique_lock<std::mutex> lock(action_mutex_);
1729     enable_flag_ = false;
1730     if (!active_flag_) return;
1731     if (!actions_.empty()) empty_cv_.wait(lock);
1732     active_flag_ = false;
1733     lock.unlock();
1734     not_empty_cv_.notify_all();
1735  }
1736
1737  void CudaEntityFetchQueue::wait_actions()
1738  {
1739     std::unique_lock<std::mutex> lock(action_mutex_);
1740     if (!active_flag_) return;
1741     enable_flag_ = false;
1742     if (!actions_.empty()) empty_cv_.wait(lock);
1743     enable_flag_ = true;
1744  }
1745
1746  } // cuda
1747  } // c10
1748
1749  文件名: aten/src/ATen/EntityTensorImpl.h (新增)
1750
1751  #pragma once
1752
1753  #include <atomic>
1754  #include <memory>
1755  #include <numeric>
1756  #include <random>
1757
1758  #include <c10/core/Backend.h>
1759  #include <c10/core/MemoryFormat.h>
1760  #include <c10/core/Storage.h>
```

```
#include <c10/core/TensorOptions.h>
#include <c10/core/DispatchKeySet.h>
#include <c10/core/impl/LocalDispatchKeySet.h>
#include <c10/core/CopyBytes.h>

#include <c10/util/Exception.h>
#include <c10/util/Optional.h>
#include <c10/util/Flags.h>
#include <c10/util/Logging.h>
#include <c10/util/python_stub.h>
#include <c10/core/TensorImpl.h>
#include <ATen/Tensor.h>
#include <ATen/ATen.h>

#define likely(x)       __builtin_expect(!!(x), 1)
#define unlikely(x)     __builtin_expect(!!(x), 0)
// #define TORCH_CHECK(a, ...) // profile mode
```

文件名: aten/src/ATen/EntityTensorImpl.cpp (新增)

```
#include <ATen/EntityTensorImpl.h>
#include <c10/cuda/CUDACachingAllocator.h>

#include <chrono>
#include <string>
#include <random>
#include <cmath>

namespace at {

namespace native {
bool pageout_manual(const Tensor& t) {
    t.unsafeGetTensorImpl()->storage().unsafeGetStorageImpl()->pageout_manual();
    return true;
}

bool pagein_manual(const Tensor& t) {
    t.unsafeGetTensorImpl()->storage().unsafeGetStorageImpl()->pagein_manual();
    return true;
}
bool need_prefech(const Tensor& t) {
    t.unsafeGetTensorImpl()->storage().unsafeGetStorageImpl()->need_prefetch();
    return true;
}
```

```
int64_t get_pointer(const Tensor&t) {
   return
t.unsafeGetTensorImpl()->storage().unsafeGetStorageImpl()->entity().impl_->entity_id_;
}
} // namespace native
} // namespace at
```

文件名: aten/src/ATen/native/native_functions.yaml (修改)

```
# representing ScalarType's. They are now superseded by usage of
# `aten::to()`. The ops remain here for backward compatibility purposes.
- func: pageout_manual(Tensor self) -> bool
   variants: method

- func: pagein_manual(Tensor self) -> bool
   variants: method

- func: need_prefech(Tensor self) -> bool
   variants: method

- func: get_pointer(Tensor self) -> int
   variants: method
```

文件名: c10/cuda/CMakeLists.txt (修改)

```
set(C10_CUDA_SRCS
      ATMConfig.cpp
      CUDAStream.cpp
      CUDAFunctions.cpp
      CUDAMiscFunctions.cpp
      CUDACachingAllocator.cpp
      CUDASwapQueues.cpp
      impl/CUDAGuardImpl.cpp
      impl/CUDATest.cpp
)
set(C10_CUDA_HEADERS
      ATMConfig.h
      CUDAException.h
      CUDAGuard.h
      CUDAMacros.h
      CUDAStream.h
      CUDAFunctions.h
      CUDAMiscFunctions.h
```

```
1849            CUDASwapQueues.h
1850            impl/CUDAGuardImpl.h
1851            impl/CUDATest.h
1852        )
```