ACTS

Implementation of an improved Memory Manager

Gabriel Carmona

gabriel.carmonat@sansano.usm.cl
Universidad Técnica Federico Santa María

July 9, 2021

Índice

Index

Device and Host memory Resource

This memory manager has an interface to use differents memory resources depending of the problem that you are solving.

Interfaces:

DeviceMemoryResource: it is an interface for the memory resources that use the device

HostMemoryResource: it is an interface for the memory resources that use the host

Both has virtual functions **doAllocate** and **doDeallocate**, and both have to be implemented by the classes that inherit this interfaces.

And each interface have a callable function called **allocate** and **deallocate**, that use the implemented function of **doAllocate** and **doDeallocate**

Some code

Methods of HostMemoryResource

```
void* allocate(std::size_t bytes, std::size_t alignment = alignof(std::max_align_t)) {
  return doAllocate(bytes, alignment);
}
void deallocate(void* p, std::size_t alignment = alignof(std::max_align_t)) {
  doDeallocate(p, alignment);
}
```

Methods of DeviceMemoryResource

```
void* allocate(std::size_t bytes, CudaStreamView stream = CudaStreamView{}) {
  return doAllocate(Nmm::detail::align_up(bytes, 8), stream);
}
void deallocate(void* p, std::size_t bytes, CudaStreamView stream = CudaStreamView{}) {
  doDeallocate(p, Nmm::detail::align_up(bytes, 8), stream);
}
```

Index

CPUMemoryResource and PinnedMemoryResource

In the case of the Host Memory Resource, two memory resources have been implemented that are at least used in the actual Cuda Plugin.

CPUMemoryResource: use **malloc** and **free** directly

PinnedMemoryResource: use **cudaMallocHost** and **cudaMallocFree**

doAllocate in CPUMemoryResource

```
void *doAllocate(std::size_t bytes, std::size_t alignment = alignof(std::max_align_t))
    override {
  // don't allocate anything if the user requested zero bytes
  if (0 == bytes) { return nullptr; }

  // If the requested alignment isn't supported, use default
  alignment = (Nmm::detail::isSupportedAlignment(alignment)) ? alignment : alignof(std::
     max_align_t);
  return Nmm::detail::alignedAllocate(bytes, alignment, [](std::size_t size) {
    void *p = malloc(size);
    if (nullptr == p) { throw std::bad_alloc{}; }
    return p;
  });
}
```

doDeallocate in CPUMemoryResource

```
void doDeallocate(void *p, std::size_t alignment = alignof(std::max_align_t)) override {
  if (nullptr == p) { return; }
  Nmm::detail::alignedDeallocate(
    p, alignment, [](void *q) { free(q); });
}
```

doAllocate in PinnedMemoryResource

```
void *doAllocate(std::size_t bytes, std::size_t alignment = alignof(std::max_align_t))
    override {
  // don't allocate anything if the user requested zero bytes
  if (0 == bytes) { return nullptr; }

  // If the requested alignment isn't supported, use default
  alignment =
    (Nmm::detail::isSupportedAlignment(alignment)) ? alignment : alignof(std::max_align_t);
  return Nmm::detail::alignedAllocate(bytes, alignment, [](std::size_t size) {
    void *p{nullptr};
    auto status = cudaMallocHost(&p, size);
    if (cudaSuccess != status) { throw std::bad_alloc{}; }
    return p;
  });
}
```

doDeallocate in PinnedMemoryResource

```
void doDeallocate(void *p, std::size_t alignment = alignof(std::max_align_t)) override {
  if (nullptr == p) { return; }
  Nmm::detail::alignedDeallocate(
    p, alignment, [](void *q) { cudaFreeHost(q); });
}
```

Index

Cuda Memory Resource

For an instance there is created a class called CudaMemoryResource, that will help other memory resources in case it needs more memory, such as expand the actual memory.

This will implement the virtual functions of device memory resource with simples **cudaMalloc** and **cudaFree**.

```
void* doAllocate(std::size_t bytes, CudaStreamView) override {
  void* p{nullptr};
  cudaMalloc(&p, bytes);
  return p;
}

void doDeallocate(void* p, std::size_t, CudaStreamView) override {
  cudaFree(p);
}
```

Arena Memory Resource

This memory uses a system of blocks and arenas that control those blocks, to reduce the fragmentation of the allocations in execution.

This class uses a globalArena to control the arenas of this memory resource. Then each arena has blocks at the disposition of the program that can be used or not. Also this class is thread safe.

```
void* doAllocate(std::size_t bytes, CudaStreamView stream) override {
  if(bytes <= 0) return nullptr;
  bytes = detail::Arena::alignUp(bytes);
  return getArena(stream).allocate(bytes);
}
void doDeallocate(void* p, std::size_t bytes, CudaStreamView stream) override {
  if(p == nullptr || bytes <= 0) return;
  bytes = detail::Arena::alignUp(bytes);
  if(!getArena(stream).deallocate(p, bytes, stream)) {
    deallocateFromOtherArena(p, bytes, stream);
  }}
```

Allocation in the Memory Resource

```
void* doAllocate(std::size_t bytes, CudaStreamView stream) override {
  if(bytes <= 0) return nullptr;
  bytes = detail::Arena::alignUp(bytes);
  return getArena(stream).allocate(bytes);
}
```

Deallocation in the Memory Resource

```
void doDeallocate(void* p, std::size_t bytes, CudaStreamView stream) override {
  if(p == nullptr || bytes <= 0) return;
  bytes = detail::Arena::alignUp(bytes);
  if(!getArena(stream).deallocate(p, bytes, stream))
    deallocateFromOtherArena(p, bytes, stream);
}
void deallocateFromOtherArena(void* p, std::size_t bytes, CudaStreamView stream) {
  stream.synchronize(); readLock lock(mtx_);
  if(usePerThreadArena(stream)) {
    auto const id = std::this_thread::get_id();
    for(auto& kv : threadArenas_)
      if(kv.first != id && kv.second->deallocate(p, bytes)) return;
  } else {
    for(auto& kv : streamArenas_)
      if(stream != kv.first && kv.second.deallocate(p, bytes)) return;
  }
  globalArena_.deallocate({p, bytes});
}
```

GlobalArena

As we saw before GlobalArena has methods of allocation deallocation, in case that they need to use anotherArena and not the same arena.

```
Block allocate(std::size_t sizeOfbytes) {
  lockGuard lock(mtx_);
  return getBlock(sizeOfbytes);
}
void deallocate(Block const& b) {
  lockGuard lock(mtx_);
  coalesceBlock(freeBlocks_, b);
}
```

Arena

The Arena has the implementation of methods to deallocate and allocate.

```
void* allocate(std::size_t bytes) {
  lockGuard lock(mtx_);
  auto const b = getBlock(bytes);
  allocatedBlocks_.emplace(b.pointer(), b);
  return b.pointer();
}
bool deallocate(void* p, std::size_t bytes) {
  lockGuard lock(mtx_);

  auto const b = freeBlock(p, bytes);
  if(b.isValid()) {
    globalArena_.deallocate(b);
  }

  return b.isValid();
}
```

And in case the deallocation is for a stream there is another definition of the method deallocate.

```
bool deallocate(void* p, std::size_t bytes, CudaStreamView stream) {
  lockGuard lock(mtx_);
  auto const b = freeBlock(p, bytes);
  if(b.isValid()) {
    auto const merged = coalesceBlock(freeBlocks_, b);
    shrinkArena(merged, stream);
  }
  return b.isValid();
}
```

So, we can the Arena is in the end the class that control the calls for allocations and deallocation to make a correct management of the memory.

This, also have some methods that appears that help to know what block is the given block for the memory or if it is other.

And when there is a deallocation the blocks have methods to know where put this free block.

Blocks

As we saw before, there is methods that can manage the blocks to deliver the correct block to use for an allocation or know where put a block that is being deallocated.

```
inline Block firstFit(std::set<Block>& freeBlocks, std::size_t size){
  auto const iter = std::find_if(freeBlocks.cbegin(), freeBlocks.cend(),   [size](auto const&
      b) { return b.fits(size); });
  if(iter == freeBlocks.cend()) return {};
  else {
    auto const b = *iter;
    auto const i = freeBlocks.erase(iter);
    if(b.size() > size) {
      auto const split = b.split(size);
      freeBlocks.insert(i, split.second);
      return split.first;
    } else {
      return b;
    }
  }
}
```

```
inline Block coalesceBlock(std::set<Block>& freeBlocks, Block const&b){
   if(!b.isValid()) return b;
   auto const next = freeBlocks.lower_bound(b);
   auto const previous = next == freeBlocks.cend() ? next : std::prev(next);
   bool const mergePrev = previous->isContiguousBefore(b);
   bool const mergeNext = next != freeBlocks.cend() && b.isContiguousBefore(*next);
   Block merged{};
   if(mergePrev && mergeNext) {
      merged = previous->merge(b).merge(*next); freeBlocks.erase(previous);
      auto const i = freeBlocks.erase(next); freeBlocks.insert(i, merged);
   } else if(mergePrev) {
      merged = previous->merge(b); auto const i = freeBlocks.erase(next);
      freeBlocks.insert(i, merged);
   } else if(mergeNext) {
      merged = b.merge(*next); auto const i = freeBlocks.erase(next);
      freeBlocks.insert(i, merged);
   } else {
      freeBlocks.emplace(b);
      merged = b;
   }
   return merged;
}
```

Index

Conclusion and future work

In the case of vecmem, the concept of Device Memory Resource and Host Memory Resource like an interface for multiple types of memory resource are very similar, so join this two codes will not be really difficult.

The use of this code applies a layer of control and complete management of the memory that is in use.

For the future, will be necesary to make studies of spaces uses, to make sure that the effect of Arena Memory Resource can reduce the fragmentation of the program or maybe find that another type of memory resource that can be usefull for other cases.