NVIDIA

RAPIDS

Numba for CUDA Programmers
Session 5 – Memory Management

Graham Markall – gmarkall@nvidia.com

This week – memory management

- Numba Memory Management Internals
- Using External Memory Management (EMM) Plugins
- EMM Plugin Development

This week we'll look in more detail at memory management with Numba. There's a reasonable amount of complexity in Numba's memory management, but I don't think it will take a whole hour to cover like the other sessions, so today is a little shorter.

There's a couple of areas we'll look at:

- First we'll cover the mechanism that Numba uses internally, that you get by default.
- And it's also got a way to replace that, so we'll look at why and how you'd replace it with some other mechanism as well.

Memory Management Internals

First of all we'll look at the internal memory management mechanisms.

Numba Memory Management

- Direct calls to Driver API cuMemAlloc, cuMemFree, etc.
- Allocations made when object constructed. E.g.:

```
# Results in immediate call to cuMemAlloc
cuda.device_array()
```

- Deallocations queued up, completed when object garbage collected:

```
# cuMemAlloc called
x = cuda.device_array(5)
# Deallocation queued, cuMemFree not called yet
del x
```


We did touch on memory management a bit in the first session, where we mentioned that when Numba needs to allocate or free memory, it does it directly using the driver API.

Whenever you call device_array, or one of the other functions that copies data to the device, it immediately makes a call to cuMemAlloc.

As the user you're not responsible for deleting memory when you've finished it - you just need to wait for the object to get garbage collected, and then Numba will eventually call cuMemFree on it.

In the example I'm forcing the x object to get garbage collected, but this isn't the point at which cuMemFree gets called. Instead, Numba adds the memory underlying x to a queue of pending deallocations. At some point later it goes through the deallocations queue and calls cuMemFree on each item in it in a batch.

Queue processing

- Deallocations processed when:
  - Queue reaches maximum entry count, or
  - Queue reaches pending size limit, or
  - An allocation needs more memory than is available.
- Controlled by config variables. Defaults:
  - `config.CUDA_MAX_PENDING_DEALLOCS_COUNT` : 10 deallocations
  - `config.CUDA_MAX_PENDING_DEALLOCS_RATIO` : 0.2 (20% of device memory)

So when, does it actually decide to run through the deallocations? There are a couple of different policies governing that behaviour.

- The first is the maximum queue length - when the queue fills up with entries, Numba empties everything out of it.
- The second is the pending size limit - that is, the queue will only hold up to a certain number of bytes' worth of deallocations before it gets forced to empty. This stops you having a small number of very large deallocations holding on to a substantial amount of device memory for too long.

If neither of these limits have been reached and you try to allocate some new memory but the allocation fails then Numba will empty the queue and try the allocation again. That way, an allocation that should succeed never gets blocked by a lack of resources caused by the pending deallocations.

You do have some control over the first policies - you can set some configuration variables to override their values.

- The default queue length is 10, which is not a lot of allocations. If you have loops that do a lot of allocation and deallocation, you'll quickly be filling the queue repeatedly so you might well want to increase the maximum queue length.
- The default size limit, which is referred to as the ratio, is 20% of device memory. If you want to change that, then you can specify it as a fraction. If you experience a lot of memory pressure, then you might want to tune it down a bit.

Deferring deallocation

- Deallocation can break asychronous execution:

```
async_func1(...) # Allocates some arrays, does async work
# async_func1's allocations are out of scope now
async_func2(...) # Does more async work
# Deferred cleanup could happen in the middle of async_func2
```

- Use `defer_cleanup()` context manager to temporarily prevent deallocations:

```
with cuda.defer_cleanup():
    async_func1(...) # Allocates some arrays, does async work
    async_func2(...) # Does more async work
# Deallocations can only happen here, outside the with block
```

- Caveat: easier to run out of memory.

Another thing to think about is that although you've got some control over the deallocation policies, you don't have explicit control over their effects. Changing the values will affect when the queue gets processed, but doesn't specify exactly when the queue will be cleared.

So, if you want to make sure that you avoid queue processing at a certain time then you need to do something else. You might want to do this if you're doing lots of asynchronous operations. When you create a sequence of async operations like kernel launches and memcopies, your python code proceeds onwards, but if your stream of operations gets interrupted by Numba clearing the queue then it breaks up the stream of async operations and forces synchronization with the device.

You can stop any queue processing and deallocation temporarily using the defer_cleanup context manager, so that any code inside its with block can run asynchronously in an unbroken stream.

In the example on the slide, the first function allocates some memory that goes out of scope and gets garbage collected, leaving some pending deallocations. Then, inside the second function, Numba goes through the pending deallocation list, causing a synchronization.

If we put both those calls inside the with block, then the deallocations can't happen until after async_func2 has returned.

This can be handy, but you do need to be aware when you're using it that it makes it easier to run out of memory because you're effectively preventing the freeing of any device memory at all.

External Memory Management Plugins

That’s covered the internal memory management mechanisms - now we'll continue by looking at using external plugins for memory management.

Recap – CUDA Array Interface

- A standard for different libraries to exchange / use each others' on-device data.
- Objects implement `__cuda_array_interface__`. Returns a dict:

```
class MyArray:
   @property
   def __cuda_array_interface__(self):
       return {
           'shape': self._shape,
           'typestr': self._typestr,
           'data': (self._data, False),
           'version': 2
       }
```

- Supported in Numba, CuPy, PyTorch, PyArrow, mpi4py, ArrayViews, JAX, RAPIDS.

Before we get into the detail of that, this is a recap of the CUDA Array Interface that we talked about in session 3.

If you have a library that allocates CUDA memory, you can share your objects with other cuda libraries by implementing the interface. That interface provides a dict with a pointer to the data and some other information, like its type and shape.

There's quite a few libraries that implement it now – these include Numba, CuPy, PyTorch, cuDF, and some other examples on the slide.

Example – Numba on CuPy data

Calling a Numba Kernel on a CuPy array:

```
import cupy
from numba import cuda

@cuda.jit
def add(x, y, out):
    start = cuda.grid(1)
    stride = cuda.gridsize(1)
    for i in range(start, x.shape[0], stride):
        out[i] = x[i] + y[i]

a = cupy.arange(10)
b = a * 2
out = cupy.zeros_like(a)

add[1, 32](a, b, out)
```


This is one of the examples from before, as a reminder that the interface is transparent to the user.

We create a Numba kernel and a CuPy array, and then we can call the kernel on the array without having to do anything manually to exchange data between the two libraries.

Numba sees an object that's not one of its own device arrays, and it checks for the CUDA Array Interface. It then uses that information to prepare the kernel arguments for the launch.

Library memory management

Problem: Combining libraries oversubscribes GPU memory

- Numba: Deferred Deallocation
- RAPIDS: Rapids Memory Manager device pool
- CuPy: Device and Pinned Memory Pools
- PyTorch: Caching memory allocator

This transparent sharing of memory between different libraries is nice and convenient, and it works well as all the libraries you're using have fairly basic memory management strategies like Numba does.

In more complex scenarios, it can be problematic. Several libraries implement pool or caching allocators, that allocate a very large chunk of memory. Then when the library needs some more device memory, it doesn't use a CUDA driver or runtime function to allocate memory, but instead it just slices up its pool or cache to provide chunks of memory to the application.

That makes for a much more efficient implementation because you can effectively allocate and deallocate memory without synchronizing with the
device and without using any CUDA calls, but there's a downside when you combine two libraries that both do this.

To start up, a library with a memory pool might allocate a really substantial chunk of memory. For example, RMM might start up by creating a pool that uses half the device memory. So if you use that, and you use a device memory pool from CuPy, you can find that before any application code is running, nearly all your device memory is allocated.

That doesn't leave you any for your own purposes outside the libraries, and it can also mean that the libraries can run out of memory in their pools, then be unable to grow the pool because another library is taking up the rest of the device memory with its own pool.

So you end up with this situation where you've got a lot of memory allocated, but not a lot in use, and not being able to allocate more.

There's some links on the slide to some of the memory allocation strategies used in some libraries - apart from Numba they're all using some kind of memory pool.

Using one strategy

- **Solution:** Configure all libraries to use one implementation
- Example:

```
import cupy
import rmm

cupy.cuda.set_allocator(rmm.rmm_cupy_allocator)

# Allocated using RMM
a = cupy.arange(5)
```


We can overcome this problem by getting all the libraries to use the allocator from just one of them. Several of the libraries we've just mentioned support this.

There's an example on the slide, where you might want to use the RAPIDS Memory Manager with CuPy. You can do that by importing them both, then before you do any allocations, you can call set_allocator in CuPy to tell it to use RMM. RMM includes a function called rmm_cupy_allocator that CuPy can call on for allocations.

Now when you create CuPy arrays, the underlying memory comes from the RMM pool.

Using RMM with Numba

```
from numba import cuda
import rmm

cuda.set_memory_manager(rmm.RMMNumbaManager)

# Allocated using RMM
a = cuda.device_array(5)
```

- CuPy + Numba using RMM pool:

```
from numba import cuda
import rmm
import cupy

cupy.cuda.set_allocator(rmm.rmm_cupy_allocator)
cuda.set_memory_manager(rmm.RMMNumbaManager)
```


You can do a similar thing to use RMM with Numba. The interface is a tiny little bit different, but the idea is still the same. In this case we call cuda.set_memory_manager with the RMMNumbaManager class, then all Numba allocations get made using RMM. We'll look in a bit more detail at the RMMNumba manager class later.

If you're going to be using CuPy and Numba both together and want to use RMM for everything, then you can just import everything and set the allocator for both.

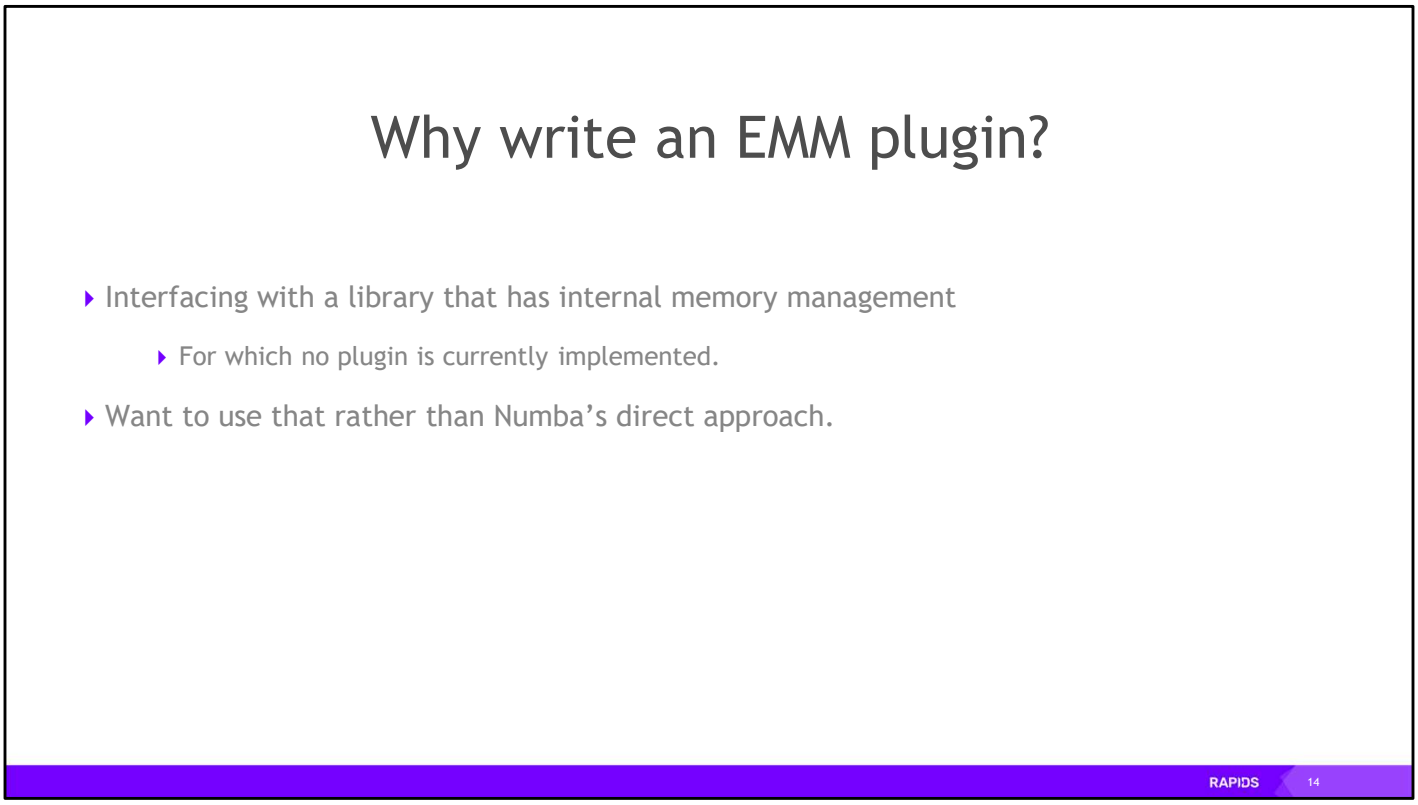
So, that's covered the API for using external memory management plugins. If you're using a library that implements an external memory management plugin for Numba then you can just set Numba up to use it by calling cuda.set_memory_manager with the class.

At the moment I'm only aware of RMM including a Numba memory management plugin within its source.

EMM Plugin Development

You might want to use another library's memory manager, and to do that you can write a plugin that lives separately from the library that you want to use for allocation.

So that's what we'll look at next - we'll see how the EMM plugins like the RMM Numba Manager work, and how we can implement one.



Why write an EMM Plugin?

- Perhaps you're working with some library that has some complex memory management implementation that you'd like to use.
- Maybe implementing your own memory allocation strategy would be more efficient for your use case.

Implementing a plugin

- Implement the `BaseCUDAMemoryManager` interface.

```
class BaseCUDAMemoryManager:
    def memalloc(self, size):
        """Allocate on-device memory in the current context."""

    def memhostalloc(self, size, mapped, portable, wc):
        """Allocate pinned host memory."""

    def mempin(self, owner, pointer, size, mapped):
        """Pin a region of host memory that is already allocated."""

    def get_ipc_handle(self, memory):
        """Return an IPC handle from a GPU allocation."""

    def defer_cleanup(self):
        """Context manager than ensures that cleanup is deferred
           while it is active."""
```


So how do you go about implementing an EMM plugin? You need to write a class that implements the BaseCUDAMemoryManager interface. We'll have a look at the methods in that interface, and what they need to do. Then, we'll have a look at a couple of examples of implementations.

On this slide are some of the methods of the interface that are directly related to memory management.

When the plugin is in use, Numba calls the methods as needed.

- If it needs an allocation of device memory, it'll call memalloc.
- If it needs pinned host memory, it'll call memhostalloc.
- mempin is used for pinning already allocated memory.
- get_ipc_handle is needed in a multi-gpu setup when numba needs an IPC handle for one of your allocations.
- Finally, in defer_cleanup you should make sure that your plugin won't do any deallocations or operations that would synchronize with the device.

BaseCUDAMemoryManager continued

- Housekeeping methods:

```
def initialize(self):
    """Do anything needed for the EMM plugin instance to be ready."""

def get_memory_info(self):
    """ Returns ``(free, total)`` memory in bytes in the context.
        Can throw NotImplementedError."""

def reset(self):
    """Clears up all memory allocated in this context."""

@property
def interface_version(self):
    """Version the plugin implements. Should be 1."""
```


There's also some housekeeping methods you might need to implement.

- In initialize, you can do anything you need to get your memory management ready to use in the current context.
- get_memory_info should return the free and total memory in bytes. It is possible that that's not an easy thing to do if you've got a complex memory management strategy and it might not make sense to just reduce the state down to a simple count like that, so you can throw NotImplementedError instead if you want.
- reset is called when Numba resets the current context. If this method gets called, then you should clean up everything in the current context.
- Also, the interface is versioned just in case it changes in future. There’s only one version for now, so you should always return 1.

Device-only plugins

- Use the `HostOnlyCUDAMemoryManager` class as a base.

```
class MyEMMPlugin(GetIpcHandleMixin, HostOnlyCUDAMemoryManager):
    def memalloc(self, size):
        # Return MemoryPointer to device memory of `size`.

    def initialize(self):
        pass

    def get_memory_info(self):
        raise NotImplementedError

    @property
    def interface_version(self):
        return 1
```

- Using the plugin:

```
cuda.set_memory_manager(MyEMMPlugin)
```

RAPIDS 17

There is a fair bit to implement for the BaseCUDAMemoryManager that you might not want to have to do. For example, you might only want to implement a device memory allocation strategy, if you're trying to use a library that doesn't manage host memory.

So, there is a base class that can simplify things a bit, which is the HostOnlyCUDAMemoryManager. It takes care of the host-side memory management, so your derived class needs to only look after the device memory.

That way, you can have a fairly minimal implementation that looks something like this on the slide. Here I'm also using the GetIpcHandleMixin to save implementing get_ipc_handle - that should work in most cases, so in general you probably want to use it for implementing a plugin.

So for a minimal implementation you only need to implement memalloc, provide a stub implementation of get_memory_info, and give the interface version. Then, you should be able to use it with cuda.set_memory_manager.

Return values

```
class MyEMMPlugin:
    def memalloc(self, size):
        ptr = my_alloc(size) # Returns e.g. a ctypes c_uint64
        ctx = self.context
        finalizer = make_finalizer(ptr.value)
        return MemoryPointer(ctx, ptr, size, finalizer=finalizer)

def make_finalizer(ptr):
    def finalizer():
        my_free(ptr)

    return finalizer
```


So far we've glossed over what goes in the implementation of the memalloc function, so let's have a look at it now.

In this example, we'll assume there's some function called my_alloc that somehow allocates device memory of a given size, and returns a pointer to that device memory.

Then we need to put together arguments for constructing a MemoryPointer object. We'll look at the MemoryPointer and similar classes shortly. For now, we'll look at what's needed to construct one:

- We need to get the current context. That's always available in self.context in an EMM Plugin implementation.
- We also need a finalizer. The finalizer is a function that gets called when the MemoryPointer is garbage collected. The finalizer should do whatever's necessary to clean up the allocation. In this example, our finalizer function calls a function called my_free, that we'll assume just frees the pointer immediately.

So with all the above, we can construct the MemoryPointer and return it.

MemoryPointer class

- Used by Numba internally to represent allocations.
- Construct with:

```
MemoryPointer(context, pointer, size, owner=None, finalizer=None)
```

- `context`: The context in which the memory is allocated
- `pointer`: The address of the memory.
- `size`: The size of the allocation in bytes.
- `owner`: Don't set this in an EMM Plugin.
- `finalizer`: Function to be called when the object is GC'd

We've just constructed a MemoryPointer. This is the class that's used internally in Numba to represent allocations of device memory.

We've already seen most of the arguments and how to use them. The only other one is owner, which Numba sometimes sets, but it's not something you need to set from an EMM plugin, so you can just leave it out the argument list.

Other memory classes

- `MemoryPointer`: Returned from `memalloc`.
- `MappedMemory`: Return from `memhostalloc` or `mempin` when host memory mapped into device space.
- `PinnedMemory`: Return from `memhostalloc` or `mempin` when host memory NOT mapped into device space

The MemoryPointer class is for device memory only. If you're implementing a plugin that manages host memory too, then there's a couple of other classes to be aware of:

- The MappedMemory class is used for host memory mapped into device space,
- The PinnedMemory class is for host memory that's not mapped into the device space.

All of these objects are constructed in the same way as the MemoryPointer class, so we'll not go over the arguments for constructing them again.

Examples

- Link to RMMNumbaManager
- Link to NumbaCUDAMemoryManager

We've covered quite a lot of the specifications and details for an EMM Plugin.

This slide likes to two EMM Plugin implementations that you can read to understand the implementation of an EMM Plugin in a real-world context.

Summary

Key points:

- Numba uses immediate alloc / deferred dealloc
- Use EMM plugins to replace internal with third-party allocator
- Write EMM plugins to interact with other allocators

Doc links:

- Numba Deallocation Behaviour
- External Memory Management

This session we've:

- Examined Numba's memory allocation strategy that defers deallocations to increase performance, and looked at how to modify its cleanup policies as well as suspending deallocation to support asynchronous use cases,
- Looked at how to replace Numba's memory allocation strategy with an external mechanism,
- And how to implement an External Memory Management plugin to provide our own memory management mechanism.

Documentation that is useful to follow when looking at Numba's memory management or implementing your own memory management is linked on the slide.

Thankyou! / Questions?


RAPIDS