Heterogeneous Memory Management (HMM)

Provide infrastructure and helpers to integrate non-conventional memory (device memory like GPU on board memory) into regular kernel path, with the cornerstone of this being specialized struct page for such memory (see sections 5 to 7 of this document).

HMM also provides optional helpers for SVM (Share Virtual Memory), i.e., allowing a device to transparently access program addresses coherently with the CPU meaning that any valid pointer on the CPU is also a valid pointer for the device. This is becoming mandatory to simplify the use of advanced heterogeneous computing where GPU, DSP, or FPGA are used to perform various computations on behalf of a process.

This document is divided as follows: in the first section I expose the problems related to using device specific memory allocators. In the second section, I expose the hardware limitations that are inherent to many platforms. The third section gives an overview of the HMM design. The fourth section explains how CPU page-table mirroring works and the purpose of HMM in this context. The fifth section deals with how device memory is represented inside the kernel. Finally, the last section presents a new migration helper that allows leveraging the device DMA engine.

Problems of using a device specific memory allocator

Devices with a large amount of on board memory (several gigabytes) like GPUs have historically managed their memory through dedicated driver specific APIs. This creates a disconnect between memory allocated and managed by a device driver and regular application memory (private anonymous, shared memory, or regular file backed memory). From here on I will refer to this aspect as split address space. I use shared address space to refer to the opposite situation: i.e., one in which any application memory region can be used by a device transparently.

Split address space happens because devices can only access memory allocated through a device specific API. This implies that all memory objects in a program are not equal from the device point of view which complicates large programs that rely on a wide set of libraries.

Concretely, this means that code that wants to leverage devices like GPUs needs to copy objects between generically allocated memory (malloc, mmap private, mmap share) and memory allocated through the device driver API (this still ends up with an mmap but of the device file).

For flat data sets (array, grid, image, ...) this isn't too hard to achieve but for complex data sets (list, tree, ...) it's hard to get right. Duplicating a complex data set needs to re-map all the pointer relations between each of its elements. This is error prone and programs get harder to debug because of the duplicate data set and addresses.

Split address space also means that libraries cannot transparently use data they are getting from the core program or another library and thus each library might have to duplicate its input data set using the device specific memory allocator. Large projects suffer from this and waste resources because of the various memory copies.

Duplicating each library API to accept as input or output memory allocated by each device specific allocator is not a viable option. It would lead to a combinatorial explosion in the library entry points.

Finally, with the advance of high level language constructs (in C++ but in other languages too) it is now possible for the compiler to leverage GPUs and other devices without programmer knowledge. Some compiler identified patterns are only do-able with a shared address space. It is also more reasonable to use a shared address space for all other patterns.

I/O bus, device memory characteristics

I/O buses cripple shared address spaces due to a few limitations. Most I/O buses only allow basic memory access from device to main memory; even cache coherency is often optional. Access to device memory from a CPU is even more limited. More often than not, it is not cache coherent.

If we only consider the PCIE bus, then a device can access main memory (often through an IOMMU) and be cache coherent with the CPUs. However, it only allows a limited set of atomic operations from the device on main memory. This is worse in the other direction: the CPU can only access a limited range of the device memory and cannot perform atomic operations on it. Thus device memory cannot be considered the same as regular memory from the kernel point of view.

Another crippling factor is the limited bandwidth (~32GBytes/s with PCIE 4.0 and 16 lanes). This is 33 times less than the fastest

GPU memory (1 TBytes/s). The final limitation is latency. Access to main memory from the device has an order of magnitude higher latency than when the device accesses its own memory.

Some platforms are developing new I/O buses or additions/modifications to PCIE to address some of these limitations (OpenCAPI, CCIX). They mainly allow two-way cache coherency between CPU and device and allow all atomic operations the architecture supports. Sadly, not all platforms are following this trend and some major architectures are left without hardware solutions to these problems.

So for shared address space to make sense, not only must we allow devices to access any memory but we must also permit any memory to be migrated to device memory while the device is using it (blocking CPU access while it happens).

Shared address space and migration

HMM intends to provide two main features. The first one is to share the address space by duplicating the CPU page table in the device page table so the same address points to the same physical memory for any valid main memory address in the process address space.

To achieve this, HMM offers a set of helpers to populate the device page table while keeping track of CPU page table updates. Device page table updates are not as easy as CPU page table updates. To update the device page table, you must allocate a buffer (or use a pool of pre-allocated buffers) and write GPU specific commands in it to perform the update (unmap, cache invalidations, and flush, ...). This cannot be done through common code for all devices. Hence why HMM provides helpers to factor out everything that can be while leaving the hardware specific details to the device driver.

The second mechanism HMM provides is a new kind of ZONE_DEVICE memory that allows allocating a struct page for each page of device memory. Those pages are special because the CPU cannot map them. However, they allow migrating main memory to device memory using existing migration mechanisms and everything looks like a page that is swapped out to disk from the CPU point of view. Using a struct page gives the easiest and cleanest integration with existing mm mechanisms. Here again, HMM only provides helpers, first to hotplug new ZONE_DEVICE memory for the device memory and second to perform migration. Policy decisions of what and when to migrate is left to the device driver.

Note that any CPU access to a device page triggers a page fault and a migration back to main memory. For example, when a page backing a given CPU address A is migrated from a main memory page to a device page, then any CPU access to address A triggers a page fault and initiates a migration back to main memory.

With these two features, HMM not only allows a device to mirror process address space and keeps both CPU and device page tables synchronized, but also leverages device memory by migrating the part of the data set that is actively being used by the device.

Address space mirroring implementation and API

Address space mirroring's main objective is to allow duplication of a range of CPU page table into a device page table; HMM helps keep both synchronized. A device driver that wants to mirror a process address space must start with the registration of a mmu_interval_notifier:

```
int mmu_interval_notifier_insert(struct mmu_interval_notifier *interval_sub,
                                 struct mm_struct *mm, unsigned long start,
                                 unsigned long length,
                                 const struct mmu_interval_notifier_ops *ops);
```

During the ops->invalidate() callback the device driver must perform the update action to the range (mark range read only, or fully unmap, etc.). The device must complete the update before the driver callback returns.

When the device driver wants to populate a range of virtual addresses, it can use:

```
int hmm_range_fault(struct hmm_range *range);
```

It will trigger a page fault on missing or read-only entries if write access is requested (see below). Page faults use the generic mm page fault code path just like a CPU page fault.

Both functions copy CPU page table entries into their pfns array argument. Each entry in that array corresponds to an address in the virtual range. HMM provides a set of flags to help the driver identify special CPU page table entries.

Locking within the sync_cpu_device_pagetables() callback is the most important aspect the driver must respect in order to keep things properly synchronized. The usage pattern is:

```
int driver_populate_range(...)
{
     struct hmm_range range;
     ...

     range.notifier = &interval_sub;
     range.start = ...;
     range.end = ...;
     range.hmm_pfns = ...;

     if (!mmget_not_zero(interval_sub->notifier.mm))
```

```
                return -EFAULT;

    again:
         range.notifier_seq = mmu_interval_read_begin(&interval_sub);
         mmap_read_lock(mm);
         ret = hmm_range_fault(&range);
         if (ret) {
             mmap_read_unlock(mm);
             if (ret == -EBUSY)
                    goto again;
             return ret;
         }
         mmap_read_unlock(mm);

         take_lock(driver->update);
         if (mmu_interval_read_retry(&ni, range.notifier_seq) {
             release_lock(driver->update);
             goto again;
         }

         /* Use pfns array content to update device page table,
          * under the update lock */

         release_lock(driver->update);
         return 0;
    }
```

The driver->update lock is the same lock that the driver takes inside its invalidate() callback. That lock must be held before calling mmu_interval_read_retry() to avoid any race with a concurrent CPU page table update.

Leverage default_flags and pfn_flags_mask

The hmm_range struct has 2 fields, default_flags and pfn_flags_mask, that specify fault or snapshot policy for the whole range instead of having to set them for each entry in the pfns array.

For instance if the device driver wants pages for a range with at least read permission, it sets:

```
range->default_flags = HMM_PFN_REQ_FAULT;
range->pfn_flags_mask = 0;
```

and calls hmm_range_fault() as described above. This will fill fault all pages in the range with at least read permission.

Now let's say the driver wants to do the same except for one page in the range for which it wants to have write permission. Now driver set:

```
range->default_flags = HMM_PFN_REQ_FAULT;
range->pfn_flags_mask = HMM_PFN_REQ_WRITE;
range->pfns[index_of_write] = HMM_PFN_REQ_WRITE;
```

With this, HMM will fault in all pages with at least read (i.e., valid) and for the address == range->start + (index_of_write << PAGE_SHIFT) it will fault with write permission i.e., if the CPU pte does not have write permission set then HMM will call handle_mm_fault().

After hmm_range_fault completes the flag bits are set to the current state of the page tables, ie HMM_PFN_VALID | HMM_PFN_WRITE will be set if the page is writable.

Represent and manage device memory from core kernel point of view

Several different designs were tried to support device memory. The first one used a device specific data structure to keep information about migrated memory and HMM hooked itself in various places of mm code to handle any access to addresses that were backed by device memory. It turns out that this ended up replicating most of the fields of struct page and also needed many kernel code paths to be updated to understand this new kind of memory.

Most kernel code paths never try to access the memory behind a page but only care about struct page contents. Because of this, HMM switched to directly using struct page for device memory which left most kernel code paths unaware of the difference. We only need to make sure that no one ever tries to map those pages from the CPU side.

Migration to and from device memory

Because the CPU cannot access device memory directly, the device driver must use hardware DMA or device specific load/store instructions to migrate data. The migrate_vma_setup(), migrate_vma_pages(), and migrate_vma_finalize() functions are designed to make drivers easier to write and to centralize common code across drivers.

Before migrating pages to device private memory, special device private `struct page` need to be created. These will be used as special "swap" page table entries so that a CPU process will fault if it tries to access a page that has been migrated to device private memory.

These can be allocated and freed with:

```
struct resource *res;
struct dev_pagemap pagemap;

res = request_free_mem_region(&iomem_resource, /* number of bytes */,
                              "name of driver resource");
pagemap.type = MEMORY_DEVICE_PRIVATE;
pagemap.range.start = res->start;
pagemap.range.end = res->end;
pagemap.nr_range = 1;
pagemap.ops = &device_devmem_ops;
memremap_pages(&pagemap, numa_node_id());

memunmap_pages(&pagemap);
release_mem_region(pagemap.range.start, range_len(&pagemap.range));
```

There are also devm_request_free_mem_region(), devm_memremap_pages(), devm_memunmap_pages(), and devm_release_mem_region() when the resources can be tied to a `struct device`.

The overall migration steps are similar to migrating NUMA pages within system memory (see :ref:`Page migration <page_migration>`) but the steps are split between device driver specific code and shared common code:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\vm\[linux-master][Documentation][vm]hmm.rst`, line 306); ***backlink***

Unknown interpreted text role "ref".

1. `mmap_read_lock()`

   The device driver has to pass a `struct vm_area_struct` to migrate_vma_setup() so the mmap_read_lock() or mmap_write_lock() needs to be held for the duration of the migration.

2. `migrate_vma_setup(struct migrate_vma *args)`

   The device driver initializes the `struct migrate_vma` fields and passes the pointer to migrate_vma_setup(). The `args->flags` field is used to filter which source pages should be migrated. For example, setting `MIGRATE_VMA_SELECT_SYSTEM` will only migrate system memory and `MIGRATE_VMA_SELECT_DEVICE_PRIVATE` will only migrate pages residing in device private memory. If the latter flag is set, the `args->pgmap_owner` field is used to identify device private pages owned by the driver. This avoids trying to migrate device private pages residing in other devices. Currently only anonymous private VMA ranges can be migrated to or from system memory and device private memory.

   One of the first steps migrate_vma_setup() does is to invalidate other device's MMUs with the `mmu_notifier_invalidate_range_start(()` and `mmu_notifier_invalidate_range_end()` calls around the page table walks to fill in the `args->src` array with PFNs to be migrated. The `invalidate_range_start()` callback is passed a `struct mmu_notifier_range` with the `event` field set to `MMU_NOTIFY_MIGRATE` and the `owner` field set to the `args->pgmap_owner` field passed to migrate_vma_setup(). This is allows the device driver to skip the invalidation callback and only invalidate device private MMU mappings that are actually migrating. This is explained more in the next section.

   While walking the page tables, a `pte_none()` or `is_zero_pfn()` entry results in a valid "zero" PFN stored in the `args->src` array. This lets the driver allocate device private memory and clear it instead of copying a page of zeros. Valid PTE entries to system memory or device private struct pages will be locked with `lock_page()`, isolated from the LRU (if system memory since device private pages are not on the LRU), unmapped from the process, and a special migration PTE is inserted in place of the original PTE. migrate_vma_setup() also clears the `args->dst` array.

3. The device driver allocates destination pages and copies source pages to destination pages.

   The driver checks each `src` entry to see if the `MIGRATE_PFN_MIGRATE` bit is set and skips entries that are not migrating. The device driver can also choose to skip migrating a page by not filling in the `dst` array for that page.

   The driver then allocates either a device private struct page or a system memory page, locks the page with `lock_page()`, and fills in the `dst` array entry with:

   ```
   dst[i] = migrate_pfn(page_to_pfn(dpage));
   ```

   Now that the driver knows that this page is being migrated, it can invalidate device private MMU mappings and copy device private memory to system memory or another device private page. The core Linux kernel handles CPU page table invalidations so the device driver only has to invalidate its own MMU mappings.

   The driver can use `migrate_pfn_to_page(src[i])` to get the `struct page` of the source and either copy the source page to the destination or clear the destination device private memory if the pointer is `NULL` meaning the source page was not populated in system memory.

4. `migrate_vma_pages()`

This step is where the migration is actually "committed".

If the source page was a `pte_none()` or `is_zero_pfn()` page, this is where the newly allocated page is inserted into the CPU's page table. This can fail if a CPU thread faults on the same page. However, the page table is locked and only one of the new pages will be inserted. The device driver will see that the `MIGRATE_PFN_MIGRATE` bit is cleared if it loses the race.

If the source page was locked, isolated, etc. the source `struct page` information is now copied to destination `struct page` finalizing the migration on the CPU side.

5. Device driver updates device MMU page tables for pages still migrating, rolling back pages not migrating.

   If the `src` entry still has `MIGRATE_PFN_MIGRATE` bit set, the device driver can update the device MMU and set the write enable bit if the `MIGRATE_PFN_WRITE` bit is set.

6. `migrate_vma_finalize()`

   This step replaces the special migration page table entry with the new page's page table entry and releases the reference to the source and destination `struct page`.

7. `mmap_read_unlock()`

   The lock can now be released.

Exclusive access memory

Some devices have features such as atomic PTE bits that can be used to implement atomic access to system memory. To support atomic operations to a shared virtual memory page such a device needs access to that page which is exclusive of any userspace access from the CPU. The `make_device_exclusive_range()` function can be used to make a memory range inaccessible from userspace.

This replaces all mappings for pages in the given range with special swap entries. Any attempt to access the swap entry results in a fault which is resovled by replacing the entry with the original mapping. A driver gets notified that the mapping has been changed by MMU notifiers, after which point it will no longer have exclusive access to the page. Exclusive access is guranteed to last until the driver drops the page lock and page reference, at which point any CPU faults on the page may proceed as described.

Memory cgroup (memcg) and rss accounting

For now, device memory is accounted as any regular page in rss counters (either anonymous if device page is used for anonymous, file if device page is used for file backed page, or shmem if device page is used for shared memory). This is a deliberate choice to keep existing applications, that might start using device memory without knowing about it, running unimpacted.

A drawback is that the OOM killer might kill an application using a lot of device memory and not a lot of regular system memory and thus not freeing much system memory. We want to gather more real world experience on how applications and system react under memory pressure in the presence of device memory before deciding to account device memory differently.

Same decision was made for memory cgroup. Device memory pages are accounted against same memory cgroup a regular page would be accounted to. This does simplify migration to and from device memory. This also means that migration back from device memory to regular memory cannot fail because it would go above memory cgroup limit. We might revisit this choice latter on once we get more experience in how device memory is used and its impact on memory resource control.

Note that device memory can never be pinned by a device driver nor through GUP and thus such memory is always free upon process exit. Or when last reference is dropped in case of shared memory or file backed memory.