HETEROGENEOUS MEMORY MANAGEMENT

Linux Plumbers Conference 2018

Jérôme Glisse

EVERYTHING IS A POINTER

All data structures rely on pointers, explicitly or implicitly:

- Explicit in languages like C, C++, ...
- Implicit in languages like Java, Javascript, Python, ...

Data structures are more or less pointer intensive, from more to less:

- Graph: each node can have multiple pointers (arrows) to other nodes
- Tree: (oriented graph) each node has pointers to children
- Hash table: array with each entry being a pointer to a list
- List: each entry has a pointer to the next entry
- Array: a single pointer to the first entry in the array

VIRTUAL ADDRESS SPACE

Pointers are virtual addresses:

- CPU page table maps virtual address to physical address of memory
- Range of virtual addresses are allocated with mmap() syscall
- Memory management in all languages is built on top of virtual addresses
- Mapping from virtual address to physical address can change (migration)

Address space:

- A mapping of virtual address to physical address (page table)
- Shared between threads in a process
- Unique to a process (struct mm_struct in the kernel)

Pointers are interpreted within an address space and thus data structures are only valid within an address space.

SPLIT ADDRESS SPACE

Multiple address spaces in a single process:

- CPU page table: process address space
- Device (GPU, FPGA, ...) page table: device specific address space
- Same virtual address does not map to same physical memory

Device address space:

- One address space per device (GPU, FPGA, ...)
- Address space managed through device specific API (OpenCL, CUDA, ...)
- Data structure is meaningful within one address space only

Sharing data structures between CPU and device needs special handling, easy for array but much harder for data structures that rely on pointers (the more pointers there are, the harder it is).

DATA STRUCTURES AND MULTIPLE ADDRESS SPACES

One address space:

```
typedef struct {          list_add(list_t *entry, list_t *head) {
        void *prev;               entry→prev = head;
        void *next;               entry→next = head→next;
} list_t;                         head→next→prev = entry;
                                  head→next = entry;
                          }
```

Two address spaces:

```
typedef struct {          list_add(list_t *entry, list_t *head) {
        void *prev;               entry→prev = head;
        long gpu_prev;            entry→next = head→next;
        void *next;               entry→gpu_prev = gpu_ptr(head);
        long gpu_next;            entry→gpu_next = head→gpu_next;
} list_t;                         head→next→prev = entry;
                                  head→next→gpu_prev = gpu_ptr(entry);
                                  head→next = entry;
                                  head→gpu_next = gpu_ptr(entry);
                          }
```

CUMBERSOME AND ERROR PRONE

Multiple address spaces are cumbersome:

- Each pointer in a data structure needs a pointer for each address space
- Each address space uses a different API for virtual address allocation

Grow the data structure with pointers for each address space:

- Bad: Wastes memory when not in use by a device
- Bad: Data structure handling function must handle each address space

Duplicate the data structure in each address space:

- Good: one set of functions for data structure for each address space
- Good: uses extra memory only when in use by device
- Bad: keeping all copies synchronized needs memory bandwidth and CPU

This is all error prone and hard to maintain and debug!

SHARE VIRTUAL ADDRESS SPACE

Share the process address space with device:

- CPU page table as canonical address space
- Device (GPU, FPGA, ...) use the CPU address space
- Same virtual address maps to same physical memory

How to share CPU address space with device:

- Hardware solution: IOMMU with ATS/PASID
- Software solution: mirror CPU page table into the device page table

Both solutions are not exclusive of each other.

Terminology, it is all the same thing:

- SVA: shared virtual address, same virtual address for CPU and devices
- SVM: shared virtual memory, it is the same thing as SVA

SVA WITH HARDWARE

Shared virtual address (SVA) with hardware (IOMMU):

- ATS: Address Translation Service
- PASID: Process Address Space ID

Event flow:

1) Device requests a virtual address translation (ATS) for a given PASID
2) IOMMU maps PASID to a CPU page table
3) IOMMU maps virtual address to physical address (through CPU page table)
4) IOMMU replies to device with a physical address
5) Device uses the physical address to access the memory

Like CPUs, devices can implement a TLB cache to cache virtual address to physical address to avoid constantly making the same request.

SVA WITH SOFTWARE

Shared virtual address (SVA) with software (HMM in linux kernel):

- Copy CPU page table to the device page table
- Keep CPU and device page tables synchronized at all times
- Same virtual address must point to same physical address at all times

Event flow:

1) Device faults on virtual address with empty mapping in page table
2) Device driver requests HMM to mirror the faulting virtual address
3) HMM snapshots the CPU page table and IOMMU map page to device
4) Device driver populates the device page table from HMM snapshot
5) Device driver resumes device activities (like CPU after a CPU page fault)

Continuous synchronization by monitoring all changes to CPU page table (using mmu_notifier inside the linux kernel).

SVA: HARDWARE OR SOFTWARE ?

SVA (Shared Virtual Address) with hardware:

- Good: only one page table (CPU page table) avoids wasting memory
- Good: same virtual address points to same physical address at all times
- Good: only needs couple of dozen lines in device driver (it’s all in hardware)
- Bad: only CPU accessible memory can be used
- Bad: limit on number of processes because of PASID limit

SVA (Shared Virtual Address) with software:

- Good: can use device memory (PCIE is not cache coherent)
- Good: no limit on number of processes
- Bad: multiple page tables
- Bad: synchronization is cumbersome
- Bad: needs a lot of driver code

SVA: HARDWARE AND SOFTWARE

SVA (Shared Virtual Address) mix of hardware and software:

- Allow use of device memory (only device can access that memory)
- Reserve a range of virtual addresses for device use only (Vulkan, OpenGL)
- Only need to populate device page table for VA range using device memory

How to mix hardware and software:

- Device uses device page table first
- Flag per device page table entry to select device or CPU page table
- Granularity needs to match kernel PAGE_SIZE

DEVICE MEMORY: YOU WANT IT!

Device memory why you want it:

- High bandwidth (800GB/s versus 30GB/s or 60GB/s for main memory)
- Lower latency (PCIE can have 10 times higher latency than device memory)
- Free PCIE bandwidth for background DMA

Issues with device memory:

- CPU might not be able to access it because of PCIE BAR SIZE
- PCIE does not support cache coherency (unlike CAPI,CCIX,NVlink,XGMI)

MIGRATING MEMORY

Migrating memory:

- Virtual address to physical address can change over process lifetime
- Migrate virtual address from one physical memory to another physical memory
- Common today on NUMA architectures (each CPU socket has its local memory)
- Consolidate CPU threads and physical memory on same socket to avoid cross talk

Memory placement is either explicit or automatic (or a mix):

- Explicit memory selection by the application (mbind() syscall)
- Explicit memory selection by sysadmin (memory cgroup)
- Automatic selection by the kernel (autonuma track CPU access)

PLACING DATA STRUCTURE

Use best memory (device or main) for each data structure is hard:

- Changes over time depending on where computation happens (CPU or device)
- Same data structure can be used concurrently by both device and CPU
- NUMA for CPU complicates the overall picture (memory topology)
- When there are multiple devices it can be hard to choose which device to use
- Not all applications will provide hints for explicit placement
- Monitoring access patterns needs resources (memory, CPU, or HW counters)
- Automatic placement can lag (decide to migrate too late to help)

Nonetheless we will need to make it work as well as possible, both explicit placement for application developer that can optimize memory placement, and automatic placement in other cases.

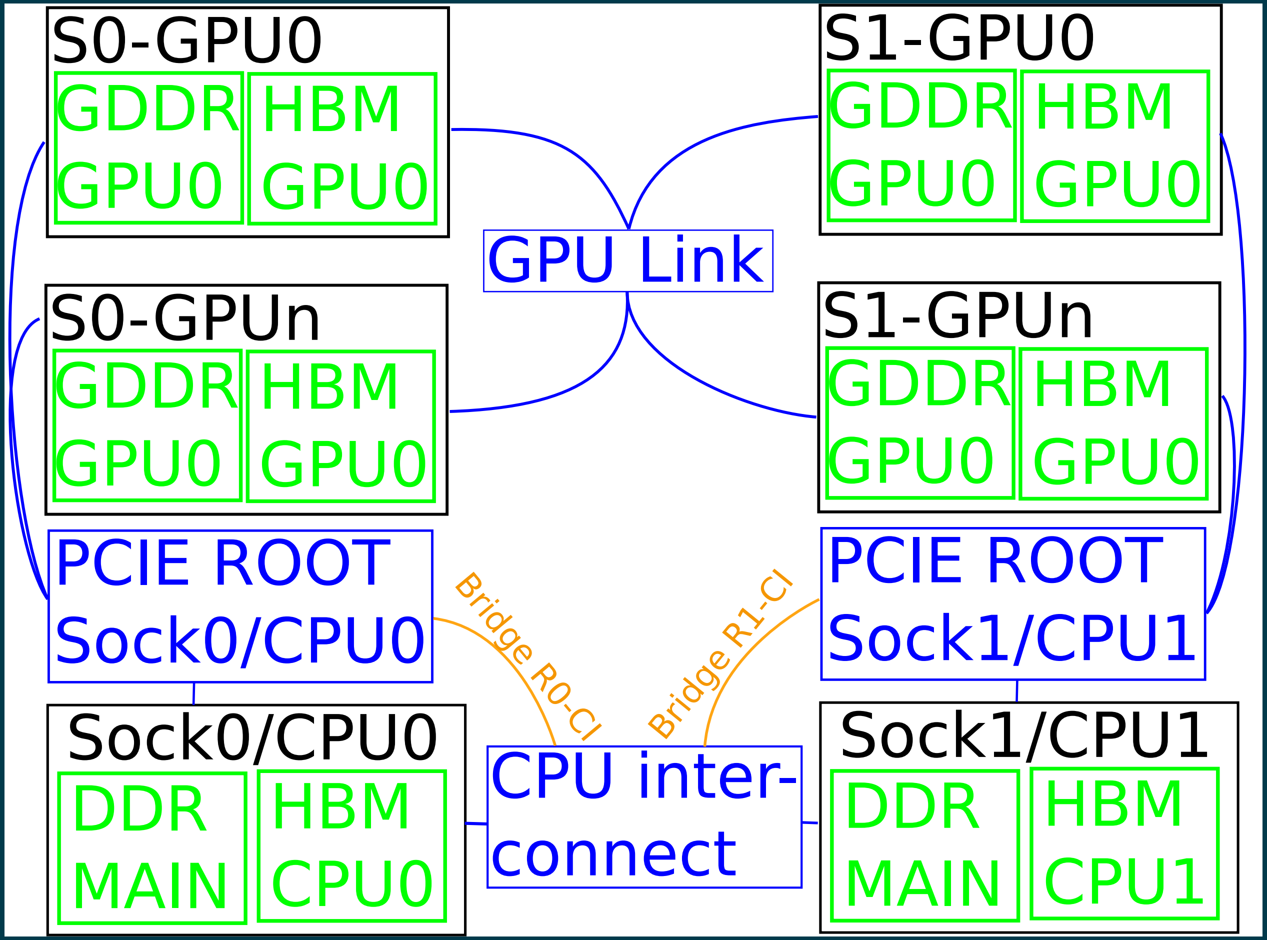
We will need a new API to support device memory.

IT ALL GETS HARDER!

New technology piles up to make all this harder:

- HBM (High Bandwidth Memory) for CPU
- Main memory (beloved DDR DIMM) average bandwidth and latency
- Persistent memory (lower bandwidth and higher latency)
- New memory technology with higher latency but gigantic size (TB)
- NUMA (Non Uniform Memory Architecture) on top of everything else
- System memory topology can become as complex as big network topology
- Device memory and to which socket the device is connected is part of topology
- Device specific link (NVlink, XGMI) is part of topology

SYSTEM MEMORY TOPOLOGY



WHERE WE ARE, WHERE TO GO

Where we are:

- IOMMU with ATS and PASID (AMD, ARM, Intel, PowerPC)
- HMM (software solution) mirror CPU page table into device page table
- HMM for CPU inaccessible device memory (DEVICE_PRIVATE)
- HMM helpers for memory migration

Where to go:

- New API for memory placement (mbind() just does not cut it)
- New way to expose system topology to userspace
- Allow different devices to access each other’s memory (RDMA and GPU)
- New automatic memory placement that also handles device memory

We are blurring the boundary between CPU and device computation!

redhat.

THANK YOU

plus.google.com/+RedHat

facebook.com/redhatinc

linkedin.com/company/red-hat

twitter.com/RedHat

youtube.com/user/RedHatVideos