Unified Memory Framework

Presenter: Igor Chorazewicz (igor.chorazewicz@intel.com)

Authors: Sergei Vinogradov, Igor Chorazewicz, Piotr Balcer, Rafal Rudnicki

Agenda

- Heterogenous memory systems challenges
- Solving the challenges using UMF
- UMF architecture overview
- Status and plans
- Summary

Heterogenous memory systems

- Increased demand for data processing leads to memory subsystems of modern server platforms becoming heterogeneous
- A single application can leverage multiple types of memory
  - Local DRAM
  - HBM
  - CXL-attached memory
  - GPU memory
- Utilizing heterogenous memory requires:
  - A way to discover available memory resources
  - Deciding where to place the data and how to migrate it between memory types
  - Interacting with different APIs for allocation & data migration

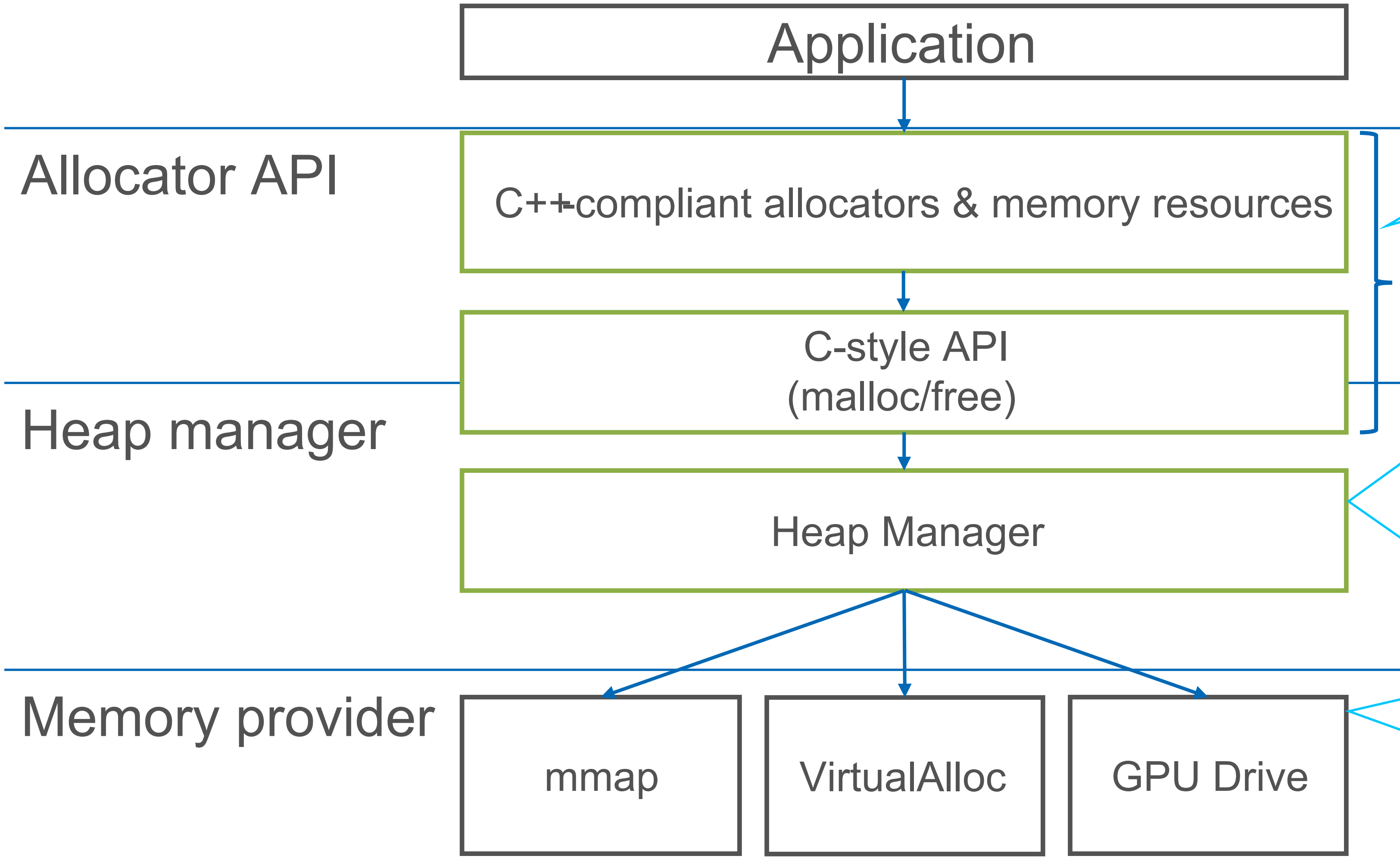
Unified Memory Framework (UMF)

Goal: Unify path for heterogeneous memory allocations and resource discovery among higher-level runtimes (SYCL, OpenMP, Unified Runtime, MPI, oneCCL, etc.) and external libs/applications.

What it is:

- A single project to accumulate technologies related to memory management.
- Flexible mix-and-match API allows tuning for a particular use case.
- Complement (not compete with) OS capabilities.
  - OS - page-size granularity; Applications - object-level abstraction.

Common Memory Allocation Structure



Allocator API

Heap manager

Memory provider

Application

C++-compliant allocators & memory resources

C-style API
(malloc/free)

Heap Manager

mmap

VirtualAlloc

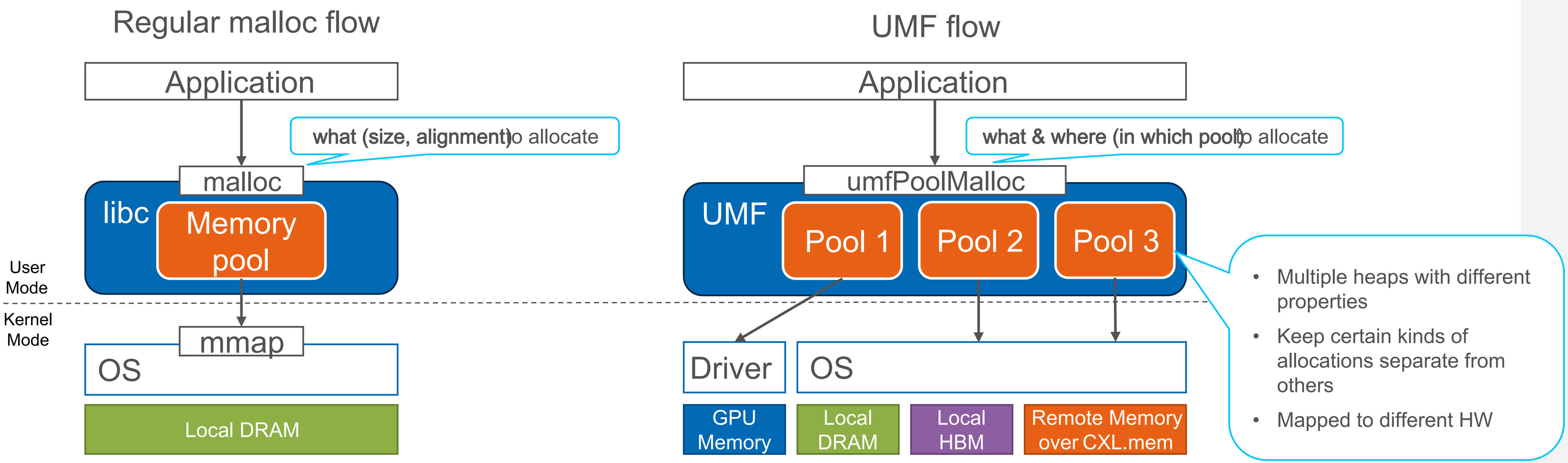
GPU Drive

Application-level interfaces:
- Object-size granularity
- Fine grain allocations

Memory pooling/caching:
- Pools big chunks from memory provider
- Service app allocations
- Various implementations optimized for different use cases
  - (concurrency, fragmentation, etc.)

System-level interface:
- Page-size granularity.
- Might be expensive.

UMF: High-Level Idea

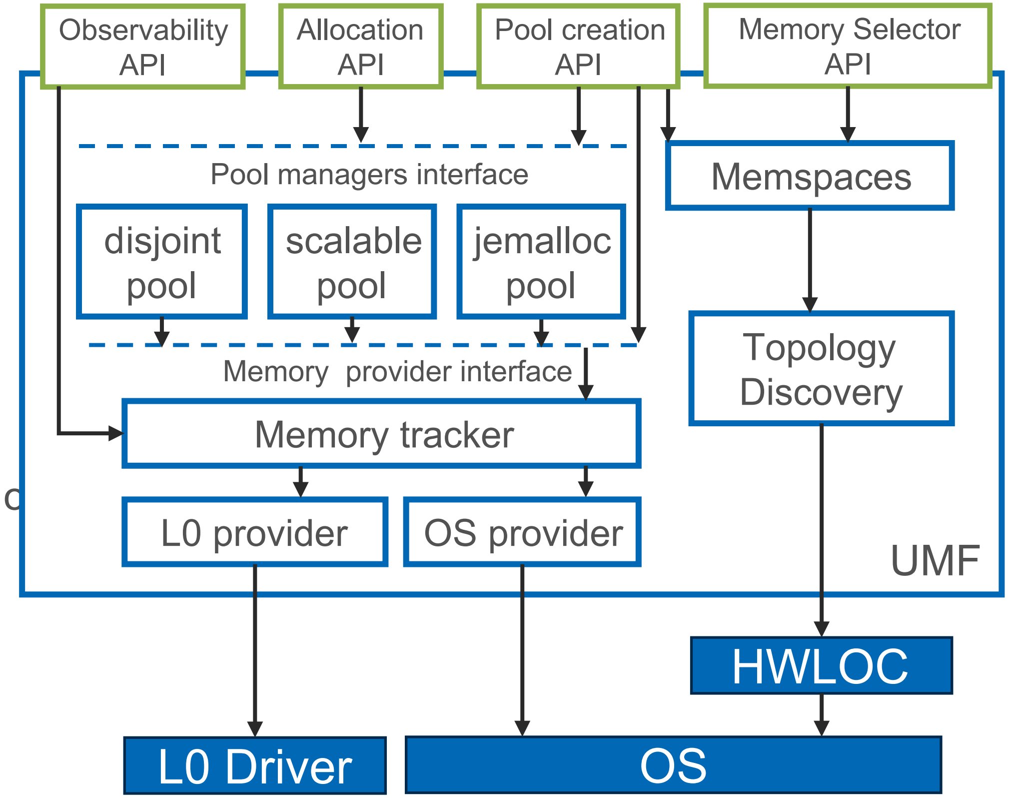


- Multiple heaps with different properties
- Keep certain kinds of allocations separate from others
- Mapped to different HW

- Expose different kinds of memory as pools/heaps with different properties and behavior. For example:
  - Pool 1 resides on GPU.
  - Pool 2 relies on OS memory tiering – do the same as regular malloc.
  - Pool 3 is bound to DRAM & CXL.mem (allows OS to migrate pages between DRAM and CXL.mem but prohibits migration to HBM). Heap manager can do page monitoring (like Linux DAMON) and make advice to OS (madvise).

UMF Architecture

- UMF is a framework to build allocators and organize memory pools.
- Pool is a combination of pool manager and memory provider.
  - Memory provider does actual memory (coarse-grain) allocations.
  - Heap manager manages the pool and services fine-grain malloc/free request.
- UMF defines heap manager and memory provider interfaces.
  - Provides implementations (disjoint pool, scalable pool, OS provider) o heap managers and memory providers.
  - Heap managers and Memory provider implementations are static libraries that can be linked on demand.
  - External heap managers and memory providers are allowed.
  - Users can choose existing ones or provide their own.



High-level API: memspaces

- Memspace is an abstraction over memory resources: it's a collection of memory targets.
- Memspace can be used as a means of discovery or for pool creation
- Memory target represents a single memory source (numa node, memory-mapped file, etc.) and can have certain properties (e.g. latency, bandwidth, capacity)
- UMF exposes predefined memspaces (HOST_ALL, HBM, LOWEST_LATENCY, etc.)

Basic Example

Pool creation flow

```
// Create memory pool of HBM memory from predefined memspace
umf_memory_pool_handle_t hbmPool = NULL;
umf_memspace_handle_t MEMSPACE_HBW = umfMemspaceHighestBandwidthGet();
umfPoolCreateFromMemspace(MEMSPACE_HBW, NULL, &hbmPool);

// Create memory pool on top of the highest capacity memory
umf_memory_pool_handle_t highCapPool = NULL;
umf_memspace_handle_t MEMSPACE_HIGH_CAP = umfMemspaceHighestCapacityGet();
umfPoolCreateFromMemspace(MEMSPACE_HIGH_CAP, NULL, &highCapPool);
```

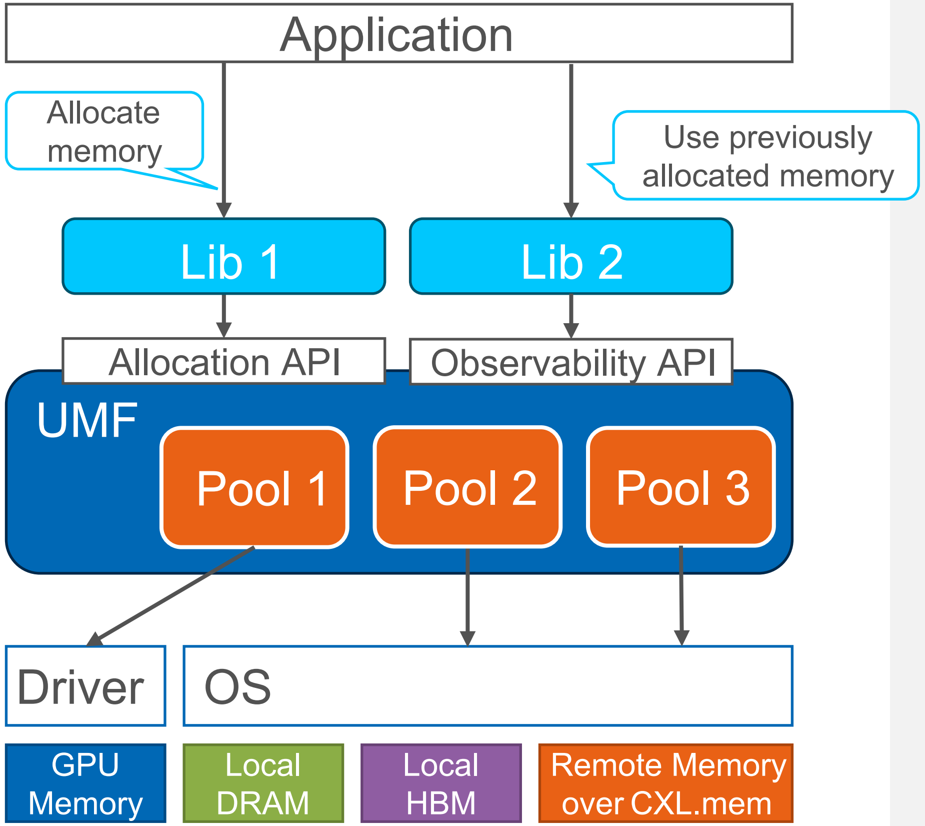
malloc/free flow

```
// Allocate HBM memory from the pool
void* ptr1 = umfPoolMalloc(hbmPool, 1024);

// Allocate memory from the highest capacity pool
void* ptr2 = umfPoolMalloc(highCapPool, 1024);

umfFree(ptr1); // Pool is found automatically
umfFree(ptr2); // Pool is found automatically
```

UMF: Interop capabilities

Memory is a key for efficient interoperability

- Modern applications are complex.
  - Multiple libraries/runtimes might be used by a single application.
  - Memory allocated by one library might be used by another library.
- UMF aggregates data about allocations.
  - Can provide memory properties of allocated regions.
- **Example:** Memory allocated by OpenMP/SYCL is used by MPI for scale-out. UMF can tell:
  - Whether it is OS-managed or GPU driver-managed memory.
  - Which NUMA node is used.
  - MPI can get IPC handle to map memory to another process.



Current Status and Plans for 2024

- First release as internal component of oneAPI 2025.0 in 2024Q3.
- Open-source repo is created for open development.
- Key stakeholders:
    - **Unified Runtime:** USM memory pooling (used by SYCL and OpenMP offload).
    - **Intel MPI:** interop with SYCL and OpenMP based on Observability & IPC API.
    - **oneCCL:** memory pooling for big allocations and IPC functionality.
    - **libiomp:** build OpenMP 6.0 support on top of UMF.
    - **CAL:** malloc/free intercept based on UMF

Summary

- UMF unifies interfaces to work with memory hierarchies.
- UMF improves efficiency by code/technology reuse.
  - Set of building blocks to adapt to particular needs.
- UMF handles interop between runtimes by aggregating data about all allocations.

- **Call to action:**
  - Try out UMF when dealing with heterogenous memory or building a custom memory allocator
- **Extra resources:**
  - https://oneapi-src.github.io/unified-memory-framework/introduction.html

intel®

Value Proposition

- **For developers:**
  - unified interfaces to work with memory kinds.
  - drive efficiency across teams.
- **For customers:**
  - better interoperability between runtimes by aggregating data about all allocations.
- **For industry:**
  - public open-source project to simplify the adoption of heterogeneous memory technologies.

Vision

- Enable application performance and scalability with the use of memory kinds/hierarchy by and across XPUs.
- Provide simple consistent mechanisms for SW developers to work with memory hierarchies and functions that operate on memory.
- Provide appropriate abstraction layers for HW innovation in the areas of memory technology, memory locality, and memory offloads.

UMF Structure

- Single repository. Single source base. Single shared library (libumf.so)
- Provides different sets of APIs:
  - **Pool creation API.**
    - A low-level API to explicitly build memory allocators/pools. Users explicitly choose heap manager and memory provider.
      - Targeted allocator developers.
    - A high-level API (Memkind replacement). Predefined pools based on the memory topology.
      - Targeted application developers.
  - **Allocation API.**
    - malloc-like API to allocate from a particular pool.
  - **Memory Selector API**
    - Choose a memory device based on user constraints.
      - E.g. High bandwidth memory, Lowest latency, Highest capacity, etc.
  - **Observability API**
    - Allows to retrieve memory properties of a memory allocated via UMF.
    - Provides an ability to create IPC handles.

Observability and IPC APIs

Process 1:

Library A:

```
// Some library creates the pool and allocates from it
umf_memory_pool_handle_t somePool = …;
// Allocate memory from some pool
void* ptr = umfPoolMalloc(somePool, 1024);
```

Library B:

```
// Another library fetches the pool the pointer belongs to
umf_memory_pool_handle_t retrievedPool = umfPoolByPtr(ptr);

// Work in progress!!!
// UMF allows to get properties of a particular allocation
// E.g. NUMA nodes, device (CPU, GPU), if GPU – device, context
umf_alloc_properties_t allocProperties;
umfGetAllocProperties(ptr, &allocProperties);


// For scale-out UMF allows to get IPC handles
umf_ipc_handle_t ipcHandle;
size_t handleSize;
umfGetIPCHandle(ptr, &ipcHandle, &handleSize);

send_to_another_process(ipcHandle, handleSize);
```

Process 2:

Library B:

```
// Another library fetches the pool the pointer belongs to
umf_ipc_handle_t ipcHandle;

receive_from_another_process(&ipcHandle);

// Create memory pool to open IPC handle
umf_memory_pool_handle_t somePool = …;

// Mmap memory pointed by IPC handle to the current process
void* ptr = NULL;
umfOpenIPCHandle(somePool, ipcHandle, &ptr);
```