HIHAT MEMORY MANAGEMENT

CJ Newburn, Jan 17, 2018 - Internal
OUTLINE

• Purpose: inform and review
• Context
• Agents and targets
• Selection
• Access enumeration
• Registration
• Requirements
ALLOCATION CONTEXT

Agents do the allocating on targets

- Allocation agents
  - \{CPU, Accel 1, Accel 2\}
- Allocation targets
  - \{CPU, Accel 1, Accel 2\}
ALLOCATION: AGENTS AND TARGETS

Requirements, enhancements

- Allocation agent
  - Execution resource where code does the allocating
  - Named since it may not be the local source thread
  - Could be a specific resource or any one of a set of resources

- Allocation target
  - Memory resource to be allocated from
  - Supports a set of possible target resources
  - Ex: managed memory allows migration between {CPU, accel 1, accel 2}

- Resources could be all or part of a physical resource
  - Use something like bitsets to specify the subset [feedback]
ALLOCATION: SELECTION

Requirements, enhancements

• Memory may have several {kinds, locations, depths}
  • Each instance is its own resource handle
• There may be several kinds of allocators for the same resource
  • {shared or unique per memory resource}
  • {shared or unique per execution resource}
  • {any size or specific size}
  • {ability to support traits} (see below)
• Select of allocator
  • By computational and memory resource
  • By chooser, which is part of ExecPol (execution policy) parameter for all actions
ALLOCATOR REGISTRATION, INVOCATION
Requirements, initial implementation

• Review of how registration works
  • User or config files register implementations of all HiHAT-prescribed functionality
  • Actions query what’s registered based on resource spec and policy, invoke it
• Registration and allocator querying share the same mapping
  • Based on compute resource agent(s), memory resource target(s), policy/selector
  • Each of these is registered to obtain an index, for a set or single entry
  • Possible implementation: 2D array [compute agent ID][memory target ID] of alloc funcs
  • If the same implementation is registered with ID values for however it’s to be referenced
    (the whole set, relevant subsets, for each individual resource), then the speed-critical lookup is fast and direct. Trade off reference complexity for speed [feedback].
ABOVE AND BELOW

• For a given set of execution and memory resources, may be several implementations
  • Can take the first, or pick a particular one using a selector
• The client/user of HiHAT sees both above and below the HiHAT dispatch layer
  • It registered the implementations, and can know what “#99” is capable of
  • It may select the implementation upon allocation, e.g. “#99”
  • Therefore it can understand semantics not explicitly exposed to HiHAT
  • HiHAT needn’t use traits/other capabilities as part of its lookup → simpler, faster
• This makes the client/user responsible for selecting a sufficient implementation
TRAITS

Some DataView properties are captured as Traits in its metadata

- A subset of traits require allocator implementation support
  - Pinned, cleared, materialized, affinitized - allocator implementation has to take action
- Traits are named with enumerated values, expressed as a bitset
- Allocator disclosed their subset of supported traits as implementation is registered
- Requested traits are expressed in alloc call
  - If debugging is enabled, that request can be checked against capability
TYPES

• The content stored for a DataView is described in two ways
  • Data layout - structure, dimensions, extent, stride, block size
    • Includes how to access elements within a data collection
  • Element types - format, how to interpret the values

• Usage of types
  • Specified at allocation, but may be mutable since same storage could be reused
  • Queried to detect a mismatch between storage format and usage that triggers conversion
  • Type info may be used to guide allocator implementation but isn’t used to select it

• Enumeration and extensibility
  • HiHAT may not architecturally name types to begin with; could leave user-extensible
  • Field that carries type is a 64b integer, which could be a bitvector
MIGRATION & MIRRORING

Requirements

- Memory object could be read and written from multiple execution resources
- Object could reside on any of multiple memory resources: many local, many remote
- Options
  - Replication of read-only copies
    - From the start or as needed
  - Migration of writable or read-only instances
    - Suppressed - preference for remote, unmigrated
    - Based on proximity, e.g. near {CPU, accel 1, accel 2}
    - Based on prioritized need for capacity, e.g. HBM, nearby bulk DRAM, SysMem, NAS
    - Based on prioritized need specialized access, e.g. HBM vs. DRAM, texture cache
MIGRATION & MIRRORING

Implementation ideas

• Many policy choices to be made
  • Record policies as metadata in DataView
  • Can be specified at allocation, some may be mutable
  • Examples: cudaMemAdvise, mmap

• Metadata includes
  • Set of memory resource targets, pre-registration, migration policies
  • Current values for mutable characteristics
    • Current affinity - subset of execution and memory resources
    • Current access sets - subset of execution resources for each of read and write

• Can be communicated to underlying runtime via allocation request
  • Provide for that in alloc API interface using ExecPol
REGISTRATION AND QUERY APIs

- Registration and querying index by agent, target and chooser
- See below for discussion on nursery

```
hhRet hhnRegAlloc(
    void *task_ptr, // function pointer
    hhResrcHndlSet resrc_hndl, // where this func pointer can execute
    hhMemHndlSet mem_hndl, // which memory devices this func pointer is used for
    void *target_specific_config, // value or pointer to blob that selects among registered implementations
    void *chooser, // target-specific config for function
    int supported_traits, // a bit vector of supported traits, based on hhMemTraitEnum values
    size_t nursery_bytes, // size of nursery; 0 if nursery not used or supported
    void *out_nursery_base_addr, // base address obtained from initial allocation that happens at registration
);
```

```
hhRet hhnQueryRegAlloc(
    hhResrcHndlSet resrc_hndl, // where this func pointer can execute
    hhMemHndlSet mem_hndl, // which memory devices this func pointer is used for
    size_t function_idx, // index among registered tasks (another version of this API selects by chooser)
    int *out_supported_traits, // a bit vector of supported traits, based on hhMemTraitEnum values
    hhTaskHndl *out_task_hndl, // returned task handle
    size_t *out_nursery_bytes, // size of nursery; 0 if nursery not used or supported
    void *out_nursery_base_addr, // base address obtained from initial allocation that happens at registration
);
```
NURSERIES
Enabling management of aggregated data structures

• Consider a tree, where all nodes in the tree are related but allocated one by one
• Moving a node at a time is cost prohibitive, but moving the whole tree is not
  • Move normally means changing physical address or attributes, VA is much harder
• One allocator may deal with several nurseries, as distinguished by chooser
  • There’s a registration for each chooser value
  • Each of those can cause a nursery allocation of a given size
ALLOC API

- Alloc interface: specify agent(s), specify ExecPol in addition to specifying target(s), specify additional metadata - migration policies, initial affinity

```c
hhRet hhuAlloc(
    hhDataView *out_mem_hndl,  // memory handle
    void **out_addr,           // address of alloc'd address
    size_t num_bytes,          // number of bytes to allocate for view
    hhMemTrait mem_trait,      // inherent memory traits
    hhMemType mem_type,        // (bit vector of) memory types
    hhExecPol exec_pol,        // provide for a rich set of execution policies (includes chooser)
    hhExecCfg exec_cfg,        // mode, profiling, exec scope, tag set
    hhResrcHndlSet resrc_hndl, // where allocator may execute from
    hhMemHndlSet mem_hndl,     // which memory devices space is allocated from, migration scope
    hhActionHndl input_dep,    // single action this depends on
    hhActionHndl *out_action_hndl // used to obtain status, events
);
```
ALLOC API - EXTENDED VERSION

- Since reading may be shared or mutually exclusive, include readers & writers
- Consider including initial home

hhRet hhuAllocExt(
    hhDataView *out_mem_hdl,       // memory handle
    void **out_addr,              // address of alloc'd address
    size_t num_bytes,             // number of bytes to allocate for view
    hhMemTrait mem_trait,         // inherent memory traits
    hhMemType mem_type,           // (bit vector of) memory types
    hhExecPol exec_pol,           // provide for a rich set of execution policies (includes chooser)
    hhExecCfg exec_cfg,           // mode, profiling, exec scope, tag set
    hhResrcHndlSet resrc_hdl,     // where allocator may execute from
    hhResrcHndlSet resrc_hdl_readers,   // which execution resources may currently read the object
    hhResrcHndlSet resrc_hdl_writers,  // which execution resources may currently write the object
    hhMemHndlSet mem_hdl,         // which memory devices space is allocated from, migration scope
    hhMemHndlSet mem_hdl_home,    // initial home that is a subset of migration scope
    hhActionHndl input_dep,       // single action this depends on
    hhActionHndl *out_action_hdl  // used to obtain status, events
);
RESOURCE ENUMERATION

Goals and Expectations

• Goals
  • What’s there - enumerate it once, avoid double coverage
  • How it’s connected - number and kinds and characteristics of links
  • Cost models - access characteristics, for unloaded and shared use
• Expectations
  • Core set of basic enumerations of what’s there
  • Extended, target-specific enumeration of add’l features, e.g. connectivity, costs, order
  • Enumeration informs cost models, cost models are specialized for each scheduler
ENUMERATION
Requirements, enhancements

• Enumeration of compute resources
• Enumeration of memory resources
  • Hang off of compute resources
  • Enumerated once even if it’s associated with multiple compute resources (ok’d for hwloc)
• Partial overlap is supportable
  • Example: >1/4 of resource X could go to each of A and B, and they could fight over 2/4
• Enumeration of connectivity between compute and memory resources [feedback]
  • Design and implementation not yet complete
  • Unlikely for the community to agree, so offer a flexible & extensible interface
  • Priority among equivalence classes - guide selection via priority, could include “whether”
  • Cost - could depend on direction, read/write mix, granularity, queue depth
RESOURCE ENUMERATION

Device and memory hierarchy

DevKind  DevHndl  MemKind  MemHndl
RESOURCE ENUMERATION

Both one to many and many to one

DevKind  DevHndl  MemKind  MemHndl
ENumeration

- Enumeration of connectivity between compute and memory resources
  - Not yet supported: in API, not implementation
  - Unlikely for the community to agree, so offer a flexible & extensible interface
  - Priority among equivalence classes - guide selection via priority, could include “whether”
  - Cost - could depend on direction, read/write mix, granularity, queue depth

/// Exactly two of the four Resrc/MemHndlSets must be non-null
hhRet hhnQueryConnectionBasic(
    hhResrcHndlSet resrc_hndl_from,  ///< execution resource the transfer is coming from (could be empty)
    hhResrcHndlSet resrc_hndl_to,   ///< execution resource the transfer is going to (could be empty)
    hhMemHndlSet  mem_hndl_from,    ///< memory resource the transfer is coming from (could be empty)
    hhMemHndlSet  mem_hndl_to,      ///< memory resource the transfer is going to (could be empty)
    size_t   *out_read_bandwidth,   ///< bytes/s of typical read bandwidth
    size_t   *out_write_bandwidth,  ///< bytes/s of typical write bandwidth
    size_t   *out_read_latency,     ///< nanoseconds for typical read latency
    size_t   *out_write_latency    ///< nanoseconds for typical write latency
);

hhRet hhnQueryConnectionExtended(
    hhResrcHndlSet resrc_hndl_from,  ///< execution resource the transfer is coming from (could be empty)
    hhResrcHndlSet resrc_hndl_to,   ///< execution resource the transfer is going to (could be empty)
    hhMemHndlSet  mem_hndl_from,    ///< memory resource the transfer is coming from (could be empty)
    hhMemHndlSet  mem_hndl_to,      ///< memory resource the transfer is going to (could be empty)
    int selector,                   ///< selector among potential cost structures, default 0
    size_t cost_structure_bytes,   ///< max permissible bytes to copy in
    void *out_ConnectionCost        ///< user-defined object that describes connectivity/costs to copy into
);
THINGS TO PONDER

• Support copy on write?
  • Could add that support as a trait

• Support is limited to least common denominator for multiple targets?
  • Maybe. Can use different allocators for different subsets of resources

• How is exclusive access managed?
  • The implementation of the HiHAT dispatch code itself will not track where instances are, nor which are valid or close by. That’d be left to what’s layered above or below.
  • Could be managed either in implementations plugged in from below, or functionality layered on top of HiHAT. We’ll need to carefully choose between those two.