



Epigram-HS Abstract Memory Device

Mamba: Managed Abstract
Memory Array

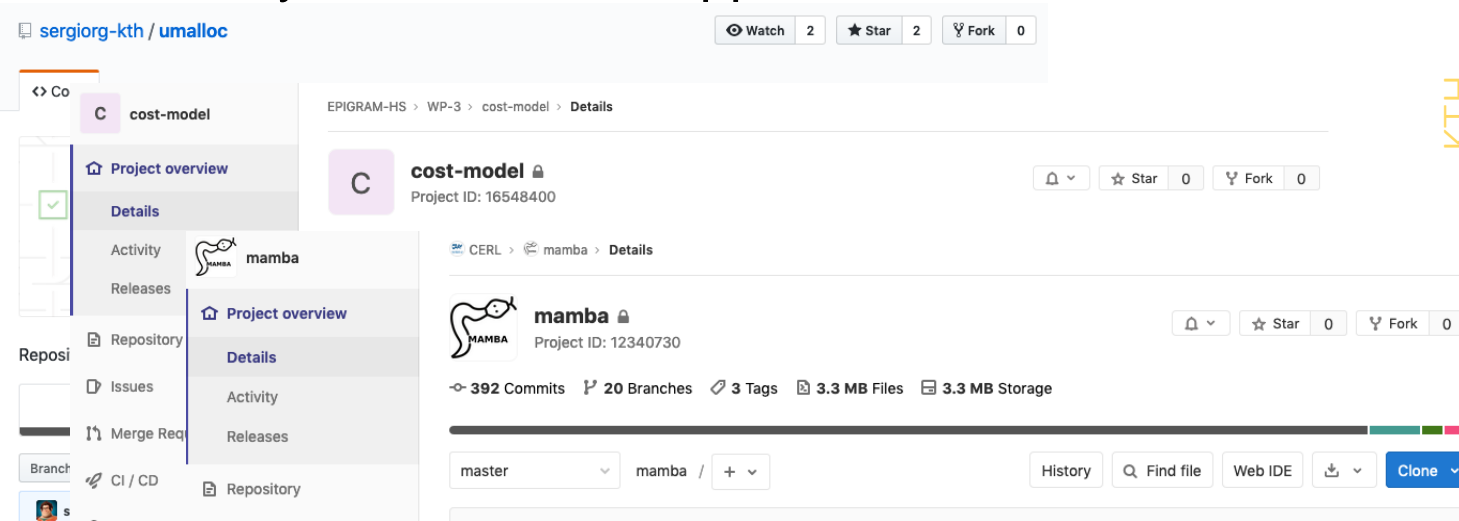
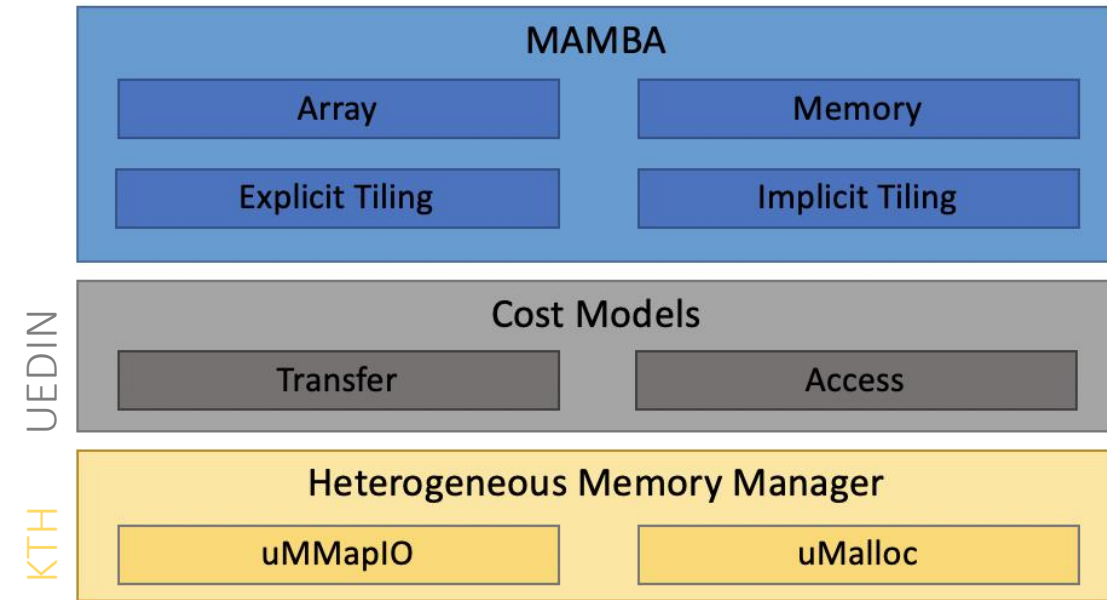


Tasks and Deliverables

- **Task T3.1: Programming and abstraction of diverse memories (M1-M30)**
 - *Leader: Cray. Contributors: KTH, UEDIN*
- **Task T3.2: Modeling of data access costs of diverse memories (M1-M30)**
 - *Leader: UEDIN. Contributors: Cray*
- **Task T3.3: Runtime for optimal data placement on diverse memories (M1-M30)**
 - *Leader: KTH. Contributors: Cray*
- **Task T3.4: Transport methods between diverse memories (M1-M36)**
 - *Leader: Cray. Contributors: KTH*
- **Deliverable D3.1: Report on current and emerging transport technologies for data movement. (M5)**
- **Deliverable D3.2: Initial design of memory abstraction device for diverse memories. (M9)**
- **Deliverable D3.3: Final design specification and prototype implementation report of API's and runtime system for data placement, migration, and access on diverse memories. (M18)**
- **Deliverable D3.4: Report on final implementation of APIs and runtime system for data placement, migration and access on diverse memories. (M30)**
- **Deliverable D3.5: Experiences and best practices on programming emerging transport technologies for data movement. (M36)**

D3.3: Final design specification and prototype implementation report of APIs and runtime system for data placement, migration and access on diverse memories.

- Updated design document
- Streamlined software stack
- Prototype implementations made available within consortium (Milestone 6) via collaborative GIT repositories
- Early feedback from application owners

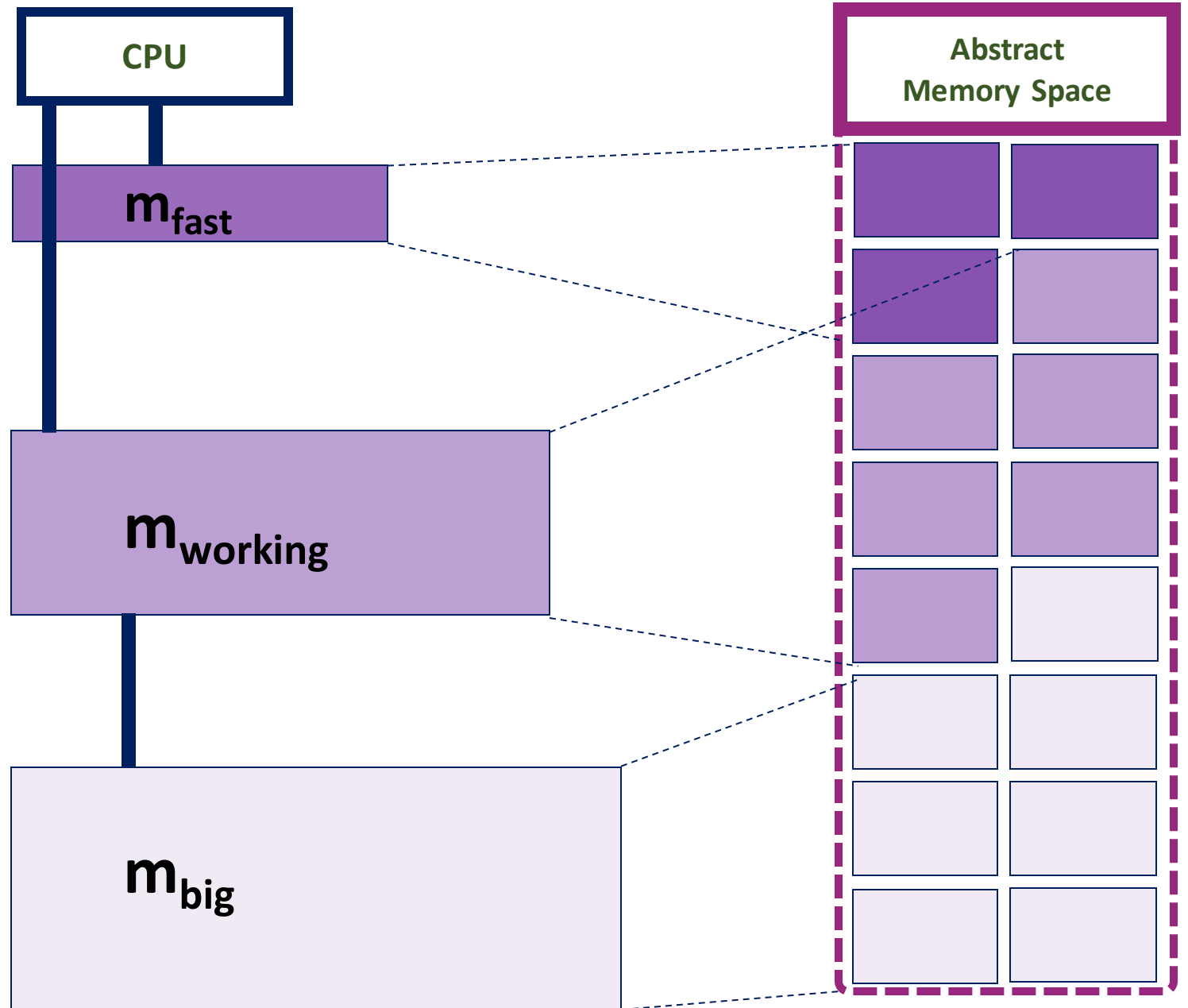


Mamba Overview

- Defines an abstract memory model
- Defines a Mamba Array object built on top of the abstract memory model
- Explicit and implicit memory management via array tiling and tile iterators
- Will include data reuse analysis and cost modelling
- Implementation in C/C++/Fortran

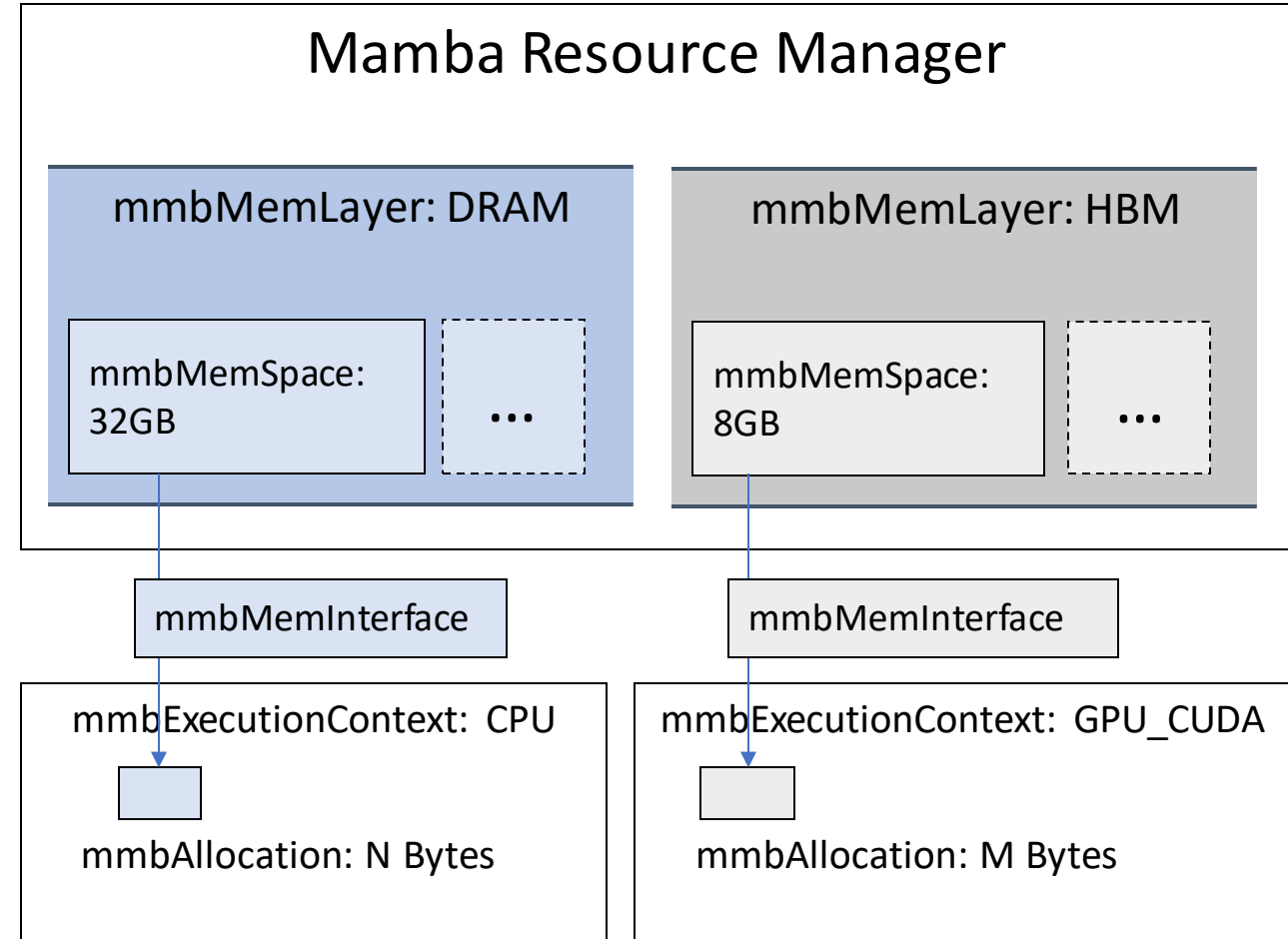
Abstract Memory Model

- Mamba is built on the assumption there are a variety of types of memory on a single node, with different characteristics
- Conceptually, these different types of memory are grouped together in an abstract memory space for use in an application.
- Users will typically access these different memories via a Mamba Array object



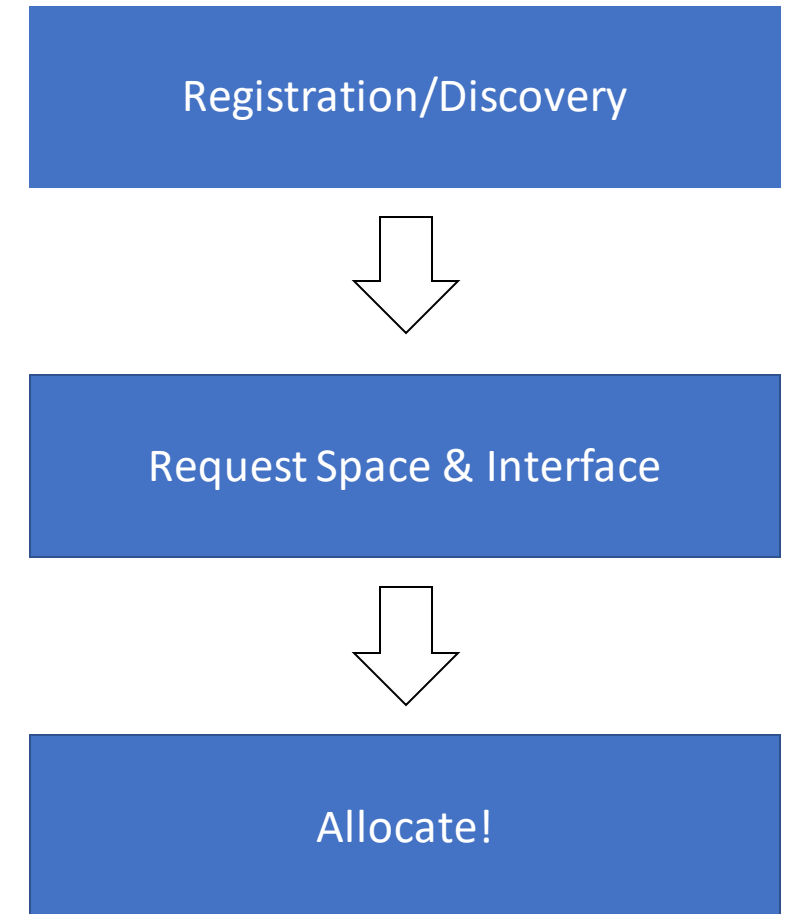
Abstract Memory Model in Mamba

- Memory Layer: `mmbMemLayer`
 - A particular type of memory with a defined set of characteristics
- Memory Space: `mmbMemSpace`
 - A size-limited addressable instantiation of a memory layer
- Memory Interface: `mmbMemInterface`
 - A space-specific memory interface for allocating blocks of memory
- Allocation: `mmbAllocation`
 - A block of allocated memory in a particular memory space
- Execution Context: `mmbExecutionContext`
 - A specifier to determine how memory should be allocated and made available for user access



General approach for memory management

- Explicitly register available memory
- OR
- Implicitly discover available memory
- THEN
- Request a memory space for a specific layer
 - Request an interface to the space
- THEN
- Explicitly allocate memory
- AND/OR
- Construct Mamba array



Managing Memory in the Mamba Memory Model

- Explicit memory registration for CPU & GPU memories
- Constructors and modifier APIs also exist for configuration options. Can typically be reused.
- Specify layer, execution context, configuration options
- Repeat for each memory layer you want to access
- Space can be returned here, or later via request_space API

```
const mmbMemSpaceConfig space_config = {  
    .size_opts = { .action = MMB_SIZE_SET, .mem_size = 8000 },  
    .interface_opts = MMB_MEMINTERFACE_CONFIG_DEFAULT,  
};
```

```
mmbMemSpace *host_space;  
stat = mmb_register_memory(MMB_DRAM, MMB_CPU, &space_config, &host_space);  
CHECK_STATUS(stat, "Failed to register dram memory for mamba\n", BAILOUT);
```

```
mmbMemSpace *device_space;  
stat = mmb_register_memory(MMB_GDRAM, MMB_GPU_CUDA, &space_config, &device_space);  
CHECK_STATUS(stat, "Failed to register dram memory for mamba\n", BAILOUT);
```


Managing Memory in the Mamba Memory Model

- Optional automatic memory discovery via hwloc library
- “--enable-discovery” configure option (enabled by default if hwloc is found)
- Requires hwloc \geq 2.0
- Pre-registers memory for all layers found in memory topology during Mamba initialization.
- Spaces may still be customised, either by changing default configuration or using modifier API

```
mmbMemSpaceConfig *config;
stat = mmb_memspace_config_create_default(&config)
CHECK_STATUS(stat, "Failed to create default space configuration \n", BAILOUT);

mmbMemSpace *host_space;
stat = mmb_request_space(MMB_DRAM, MMB_CPU, config, &host_space);
CHECK_STATUS(stat, "Failed to get DRAM space for CPU\n", BAILOUT);

mmbMemSpace *device_space;
stat = mmb_request_space(MMB_GDRAM, MMB_GPU_CUDA, config, &device_space);
CHECK_STATUS(stat, "Failed to get GDRAM space for GPU_CUDA\n", BAILOUT);
```

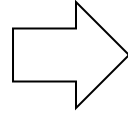
Managing Memory in the Mamba Memory Model

- Once a space is acquired, you can request a memory interface, which allows to allocate memory from the space.
- You can configure the interface at request time, or later by using a modifier API
- The interface may then be used for explicit memory allocation or Mamba Array allocation

```
const mmbMemInterfaceConfig host_conf =  
    { .provider = MMB_PROVIDER_DEFAULT, .strategy = MMB_POOLED };  
  
stat = mmb_request_interface(host_space, &host_conf, &host_interface);  
CHECK_STATUS(stat, "CPU interface request failed. (%d)\n", BAILOUT);  
  
const mmbMemInterfaceConfig device_conf =  
    { .provider = MMB_PROVIDER_DEFAULT, .strategy = MMB_STRATEGY_DEFAULT };  
  
stat = mmb_request_interface(device_space, &device_conf, &device_interface);  
CHECK_STATUS(stat, "GPU interface request failed. (%d)\n", BAILOUT);
```

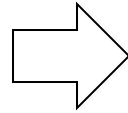
Memory API – Explicit memory allocation

- Using the host interface, we allocate in DRAM, returning an mmbAllocation



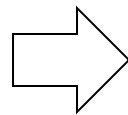
```
mmbAllocation *host_buffer;  
err = mmb_allocate(n_bytes, host_interface, &host_buffer);  
if (MMB_OK != err) {  
    ERR("Cannot allocate host buffer (%d).\n", err);  
    return EXIT_FAILURE;  
}
```

- Using the device interface, we allocate in GPU memory, returning an mmbAllocation



```
mmbAllocation *device_buffer;  
err = mmb_allocate(n_bytes, device_interface, &device_buffer);  
if (MMB_OK != err) {  
    ERR("Cannot allocate device buffer (%d).\n", err);  
    return EXIT_FAILURE;  
}
```

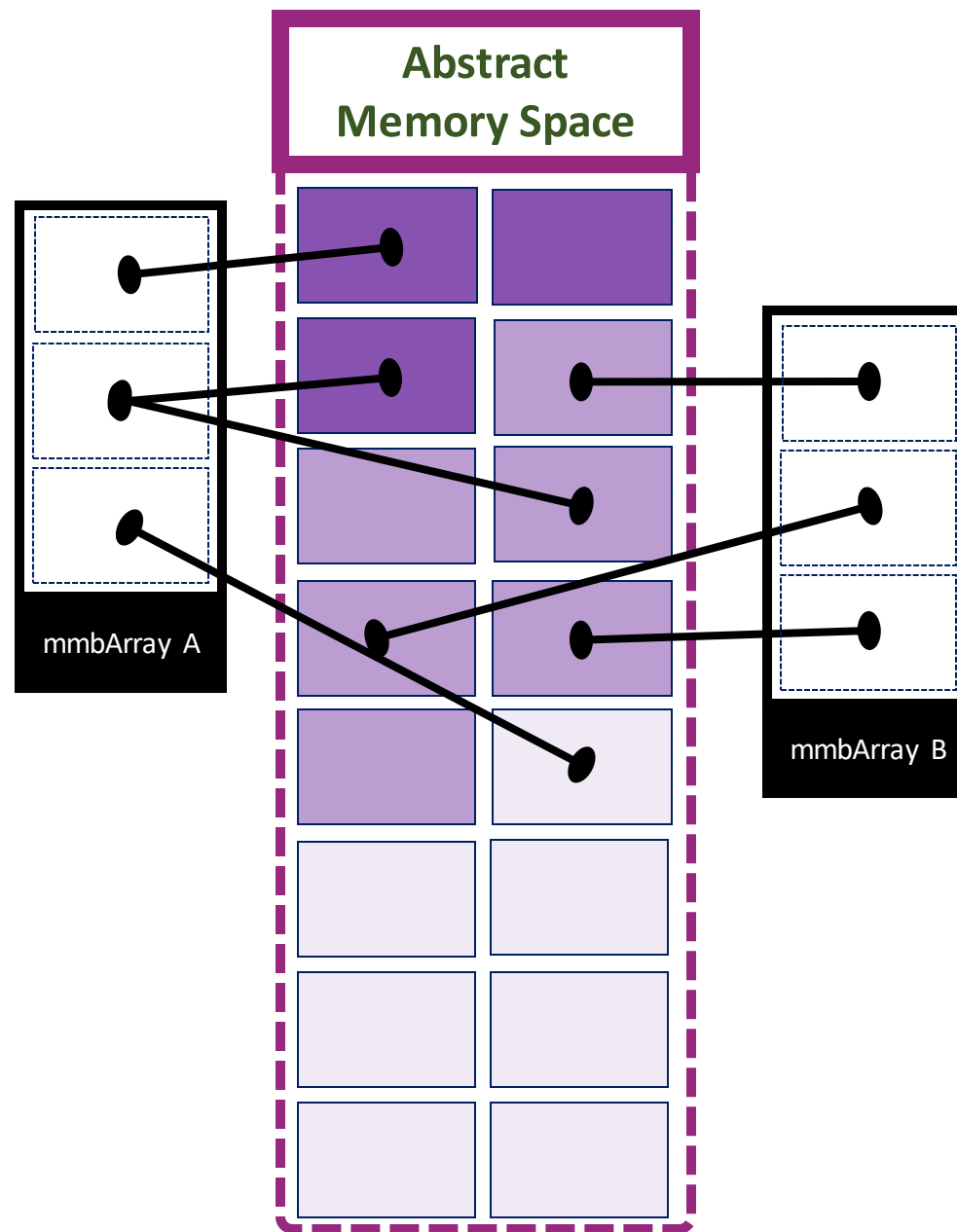
- mmb_copy copies data from one allocation to another



```
err = mmb_copy(device_buffer, host_buffer);  
if (MMB_OK != err) {  
    ERR("Cannot copy data from host to device (%d).\n", err);  
    return EXIT_FAILURE;  
}
```

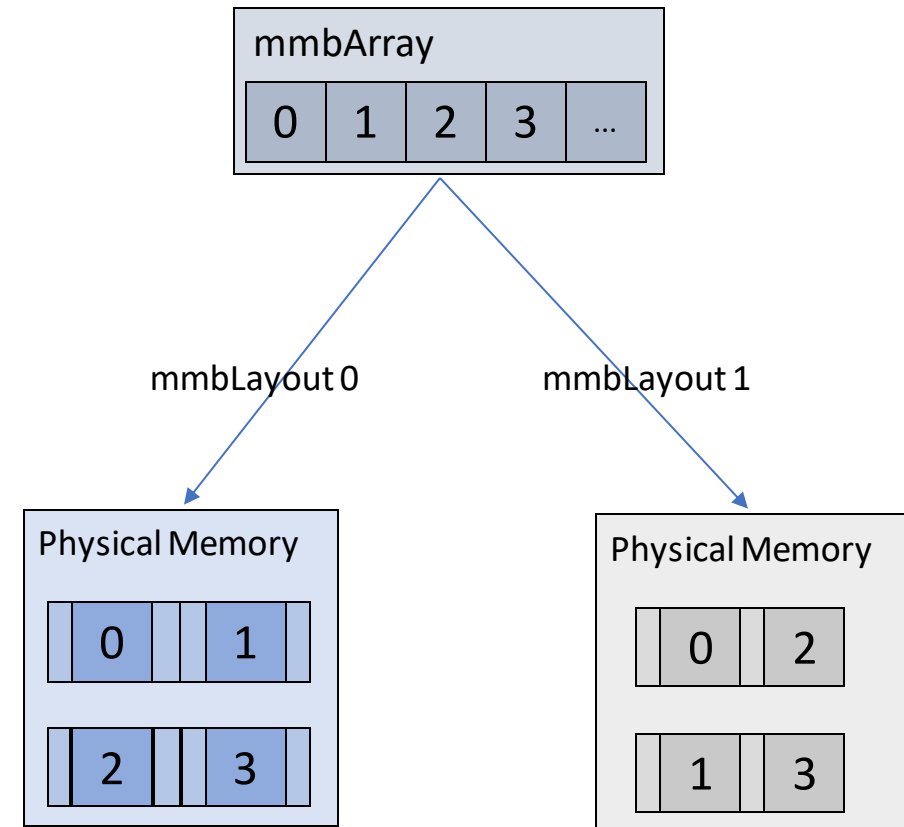
Mamba Array Concept

- **Mamba Arrays** may be spread across multiple memories
- Subsets of the array may be **duplicated**, or **moved** between memories
- Movement may be controlled **explicitly** by the user, or **implicitly** by the Mamba runtime to exploit data reuse.
- A Mamba Array can consist of several **tiles** that are suitable for movement



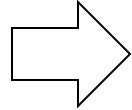
Mamba Array & Layout

- Mamba Array: `mmbArray`
 - An array object, can be internally allocated via `mmbMemInterfaces` or wrapped around an existing user pointer
 - May be spread across multiple memory spaces
 - May be tiled for loop iteration
 - Maps indices to allocations in memory spaces
 - Exploits a layout object to map to physical memory address
- Mamba Layout: `mmbLayout`
 - Defines the mapping from array indices to physical memory layout
 - Describes array characteristics such as:
 - Dimensions (2/3/etc)
 - Object type (Element/AOS/SOA)
 - Layout order (e.g. C vs Fortran ordering)
 - Layout type (e.g. regular, block cyclic, irregular)
 - Element and dimensions pre and post padding (in bytes)



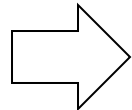
Basic Layout & Array construction

- Create a 1d array layout, specifying element size with no padding



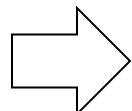
```
mmbLayout *layout;
stat = mmb_layout_create_regular_1d(sizeof(float), MMB_PADDING_NONE, &layout);
CHECK_STATUS(stat, "Failed to create regular 1d layout\n", BAILOUT);
```

- Define array dimensions, and construct array, specifying dimensions, layout, memory interface, and expected read/write policy



```
mmbArray* mba;
size_t M = 128;
mmbDimensions dims = {1, &M};
stat = mmb_array_create(&dims, layout, dram_interface, MMB_READ_WRITE, &mba);
CHECK_STATUS(stat, "Failed to create mamba array\n", BAILOUT);
```

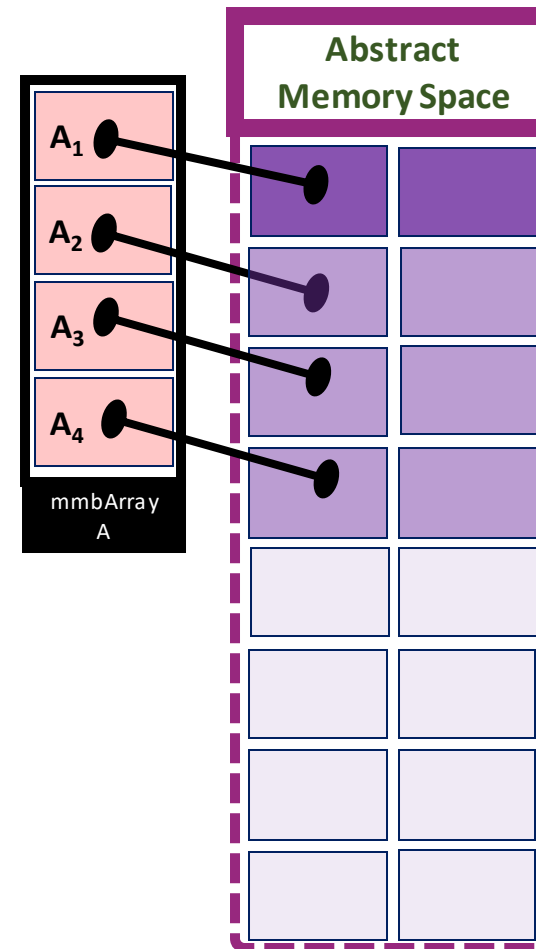
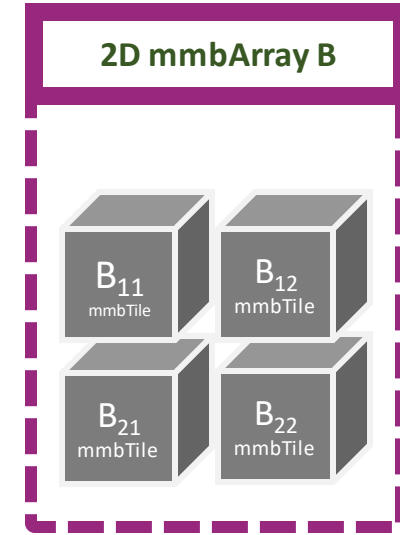
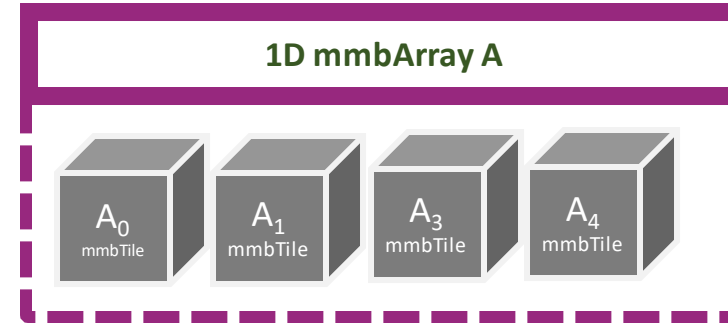
- Can also wrap existing pointer. Still required to specify a memory interface so Mamba knows where pointer resides and how to copy it.



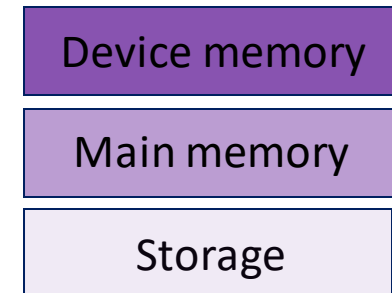
```
float* buffer0 = malloc(sizeof(float) * M);
stat = mmb_array_create_wrapped(buffer0, &dims, layout,
                                dram_interface, MMB_READ_WRITE, &mmba0);
CHECK_STATUS(stat, "Failed to create mamba array\n", BAILOUT);
```

Mamba Array Tiling

- A decomposition of the Mamba array into tiles (otherwise known as... chunks, blocks, pieces, subsets, etc)
- Currently required user-provided tile size in each dimension
- Being tiled is a property of the array – therefore only a single tiling may be valid on an array at any one time.
- Tiles may be iterated over by using a tile iterator object
- Tiles may be duplicated, or migrated, to alternative memory tiers
- Individual array elements are accessed using indexing macros

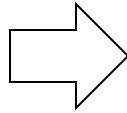


Memory types present in abstract space (for example):



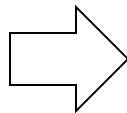
Creating a 1D Array Tiling

- Choose a tile size, and tile the array 'mba'



```
size_t tile_size = 8;
size_t chunkdims[1] = {tile_size};
mmbDimensions chunks = {1, chunkdims};
stat = mmb_array_tile(mba, chunks);
CHECK_STATUS(stat, "Failed to tile mamba array\n", BAILOUT);
```

- Shortcut to create a single tile covering whole array

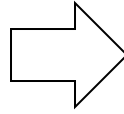


```
mmbError stat = mmb_array_tile(mba, mba->dims);
CHECK_STATUS(stat, "Failed to tile mamba array\n", BAILOUT);
```


Iterating over 1D Array Tiles and Accessing Elements

- Explicit iteration

- User specified iteration
- Tiles are accessed by index in the array tiling
- E.g. `tile_at(index)`



```
mmbArrayTile *tile;
for (size_t ti = 0; ti < tiling_dims->d[0]; ++ti) {

    stat = mmb_index_set(idx,ti);
    CHECK_STATUS(stat, "Failed to set mmb index\n", BAILOUT);

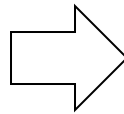
    stat = mmb_tile_at(mba, idx, &tile);
    CHECK_STATUS(stat, "Failed to get tile at index\n", BAILOUT);

    ...

}
```

- Tile iterators

- Simpler array iteration
- Contains a list of all tiles to iterate over
- Contains a schedule over the list
- Provides typical operations, e.g. `next()`



```
mmbError stat = mmb_array_tile(mba_a, mba_a->dims);
CHECK_STATUS(stat, "Failed to tile mamba array\n", BAILOUT);

mmbTileIterator *it;
stat = mmb_tile_iterator_create(mba_a, &it);
CHECK_STATUS(stat, "Failed to create tile iterator\n", BAILOUT);

stat = mmb_tile_iterator_first(it);
CHECK_STATUS(stat, "Failed to set tile iterator to first\n", BAILOUT);

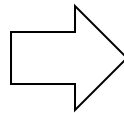
mmbArrayTile* tile = it->tile;
for (size_t i = tile->lower[0]; i < tile->upper[0] ;++i) {
    MMB_IDX_1D(tile, i, float) = 0.f;
}

stat = mmb_tile_iterator_destroy(it);
CHECK_STATUS(stat, "Failed to free tile iterator\n", BAILOUT);

stat = mmb_array_untile(mba_a);
CHECK_STATUS(stat, "Failed to untile mamba array\n", BAILOUT);
```

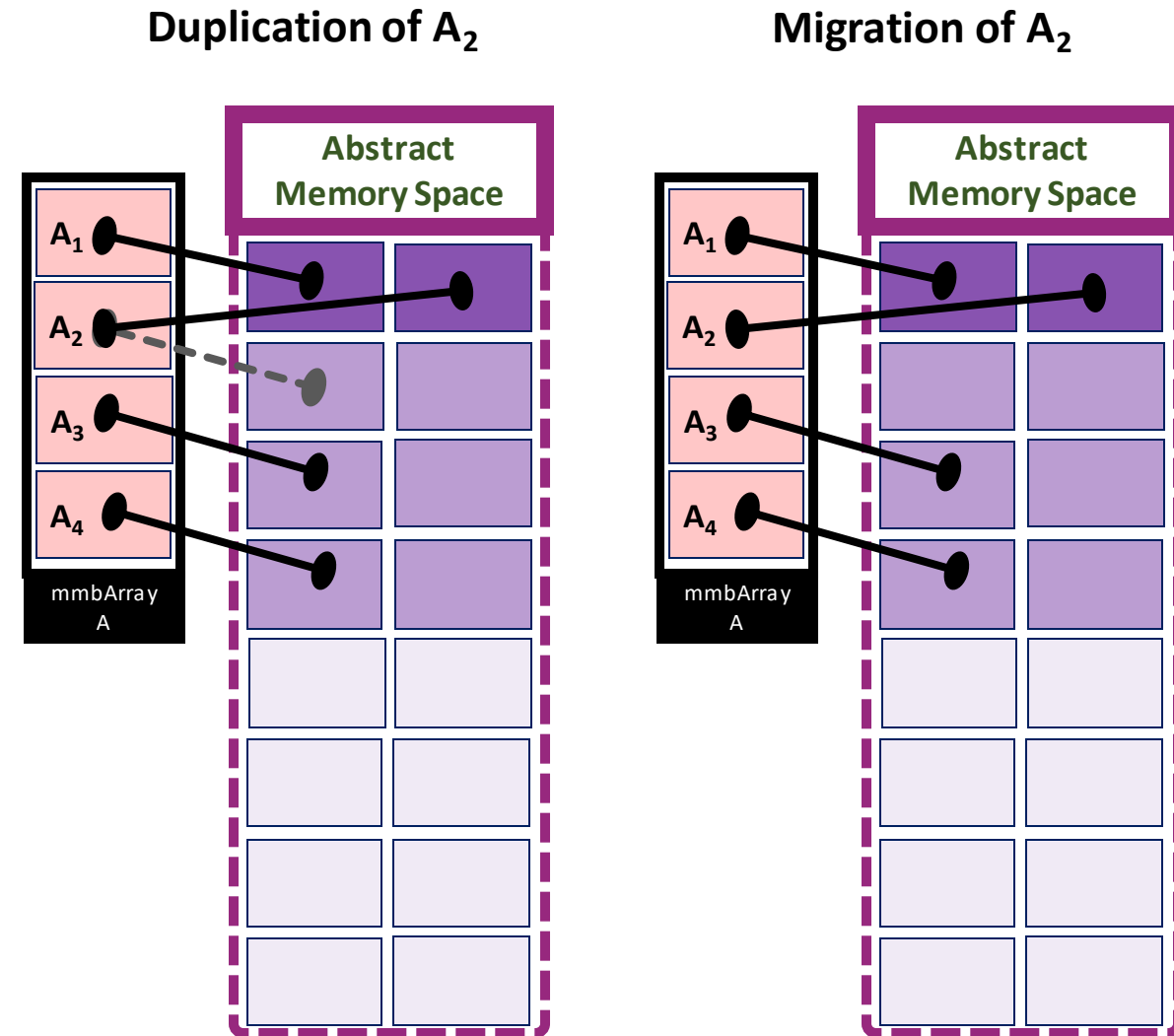
- Element accessors

- Accessor macro per dimension
- Evaluates to regular 1D C array access



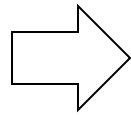
Duplicating, Migrating, and Merging Array Tiles

- Array tiles may be duplicated on other memories
 - 2 copies of tile memory exist, only one of which is legally accessible at any time
 - Duplicates may be merged back into parent tile
- Array tiles may be migrated to other memories
 - 1 copy of tile memory exists, which is moved to a different memory
- User can choose to duplicate/migrate, or runtime may internally duplicate/migrate
- Tiles may have different access types to array as a whole
 - E.g. duplicate an array tile as a read-only object
- Tile metadata must be available in the expected execution space



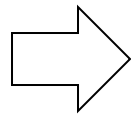
Array Initialisation and Tiling, followed by Duplicating, Writing to, and Merging 1d Array Tiles on GPU

- Initialise Mamba



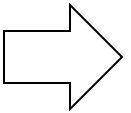
```
mmb_init(MMB_INIT_DEFAULT);
```

- Request memory space & interface to space, for array allocation



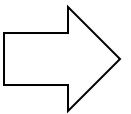
```
mmbMemSpace *dram_space;  
mmb_request_space(MMB_DRAM, MMB_EXECUTION_CONTEXT_DEFAULT, NULL, &dram_space);  
mmbMemInterface *dram_interface;  
mmb_request_interface(dram_space, NULL, &dram_interface);
```

- Create regular 1 dimensional array layout



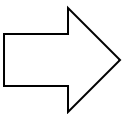
```
mmbLayout *layout;  
mmb_layout_create_regular_1d(sizeof(float), MMB_PADDING_NONE, &layout);
```

- Create array of size dims, with specified layout, allocate data using dram interface, will be read/writeable



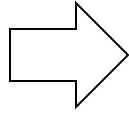
```
mmbArray *mba;  
size_t arrdims[1] = {32};  
mmbDimensions dims = {1, arrdims};  
mmb_array_create(&dims, layout, dram_interface, MMB_READ_WRITE, &mba);
```

- Non mamba function— initialize array data



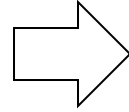
```
init_array(mba);
```

- Request a different space & interface, this time on GPU



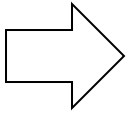
```
mmbMemSpace *gpu_space;  
mmb_request_space(MMB_GDRAM, MMB_EXECUTION_CONTEXT_DEFAULT, NULL, &gpu_space);  
mmbMemInterface *gpu_interface;  
mmb_request_interface(gpu_space, NULL, &gpu_interface);
```

- Tile Mamba array with 1d 8-element tiles



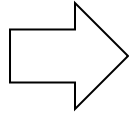
```
size_t chunkdims[1] = {8};  
mmbDimensions chunks = {1, chunkdims};  
mmb_array_tile(mba, &chunks);
```

- Loop over array tiles (aiming to reduce verbosity of this part of API)



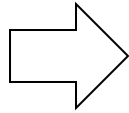
```
mmbArrayTile *tile;  
mmbIndex *idx;  
mmbDimensions *tiling_dims;  
mmb_index_create(1, &idx);  
mmb_tiling_dimensions(mba, &tiling_dims);  
for (size_t ti = 0; ti < tiling_dims->d[0]; ++ti) {
```

- Access tile at specified index



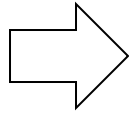
```
    mmb_index_set(idx, ti);  
    mmb_tile_at(mba, idx, &tile);
```

- Duplicate array tile using GPU interface (copies data from CPU tile to GPU tile)



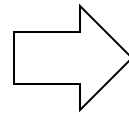
```
    mmbArrayTile *duplicate_tile;  
    mmb_tile_duplicate(tile, gpu_interface, MMB_READ_WRITE, layout, &duplicate_tile);
```

- Ask for device-local copy of tile metadata (accessible from within cuda kernel)



```
    mmb_tile_get_space_local_handle(duplicate_tile, &dev_tile);  
    write_to_tile_cuda_ker<<<block_grid_width, threads_per_block>>>(dev_tile);
```

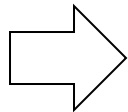
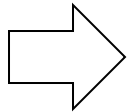
- Merge duplicate tile into parent tile, copies data from gpu tile to cpu tile, overwriting parent tile data.



```
    mmb_tile_merge(duplicate_tile, MMB_OVERWRITE);  
}
```

Multidimensional arrays: ND array construction

- A 2D Mamba array
- Create a regular 2d layout object, specifying element size, dimensions, ordering, and (lack of) padding.
- Create mamba array, specifying the memory interface to use for allocation, and the type of access expected (read-only, read-write, write-only)
- Construction and tiled array access currently works for ND arrays, still working on optimized transport to support tile duplication etc.



```
mmbArray *mba;
mmbLayout *layout;
stat = mmb_layout_create_regular_nd(sizeof(float), 2, MMB_COLMAJOR,
| | | | | | | | | | | | | | | | MMB_PADDING_NONE, &layout);
CHECK_STATUS(stat, "Failed to create 2d array layout\n", BAILOUT);

size_t dims[2] = {array_size_M, array_size_N};
mmbDimensions array_dims = {2, dims};
mmb_array_create(&array_dims, layout, dram_interface, MMB_READ_WRITE, &mba);
CHECK_STATUS(stat, "Failed to create mamba array\n", BAILOUT);
```

Advanced Layout Construction: 2D Block-cyclic

Regular contiguous element indices

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

2x2 block cyclic
contiguous element indices

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

- Continuous elements of array are organized into 2D blocks
- Improve locality, for example in dense matrix computations
- Combine with array tiling to optimize transport for tiled movement
- Each tile is now a contiguous block of memory from the outset
- Tradeoff – indexing for tiling that doesn't match block cyclic layout is more expensive

Construction of a 2D Block-cyclic Mamba Array

- A 2D Mamba array
- Create a block 2d layout object, specifying element size, dimensions, ordering, block size, ordering, and (lack of) padding.
- Create mamba array with specified layout, in DRAM, with the type of access expected (read-only, read-write, write-only).
- Would also use block_dims during tiling for optimized transport
- Block-cyclic indexing used when tile does not match block size

```
mmbLayout *layout;
size_t bdims[2] = {block_size_M, block_size_N};
mmbDimensions block_dims = {2, bdims};
stat = mmb_layout_create_block(sizeof(float), 2, MMB_COLMAJOR,
| | | | | | | | | | | | | | | | &block_dims, NULL, MMB_PADDING_NONE, &layout);
CHECK_STATUS(stat, "Failed to create block cyclic layout\n", BAILOUT);

mmbArray *mba;
size_t adims[2] = {array_size_M, array_size_N};
mmbDimensions arr_dims = {2, adims};
stat = mmb_array_create(&arr_dims, layout, dram_interface,
| | | | | | | | | | | | | | | | MMB_READ_WRITE, &mba);
CHECK_STATUS(stat, "Failed to create wrapped array\n", BAILOUT);
```

New Ease-of-use features

- Initialisation options
 - Configure logging infrastructure
 - Configure memory management infrastructure
 - Set default behavior
 - Custom allocator settings
- Logging infrastructure
 - Custom logging options
 - Override library logging with user logging
- Proper release cycle & versioning
- Unit testing
 - CHEAT

Fortran Interface

- A set of derived datatypes that match or augment the C API.
- Some examples provided to mirror the C examples
- To have a Fortran array object for tiles programmer will have to provide an appropriately typed/dimensioned pointer.
- Will also provide similar indexing macros from the preprocessor.
- Work is ongoing, expect to have version available in July.

C++ Interface

- Very early implementation stages, not released yet
- Three phase approach:
 - 1: Utility: simple wrapper of C API
 - 2: Ease-of-use: extensions to API using C++ syntactic sugar
 - 3: Advanced features: enabled through use of higher level abstractions available in C++

Distributed Mamba Arrays

- Early design stages for connection with WP2
- Extend the Mamba array concept to multiple processes.
- Distributed Mamba Array built on top of local mamba arrays on each node
- Will rely on transport library UDJ for inter-node movement & redistribution
- MPI transport in UDJ will explore use of EPCC work on persistent operations where possible.

Summary: Current status and Next Steps

Current

- C interface
- Resource manager
 - Can allocate and move memory on CPU, GPU (CUDA) using memory API
 - Generic memory API with custom allocators, not yet connected to e.g. uMMapIO
- Array construction, copy
 - N dimensional regular arrays
- Array Tiling
 - Arrays can be tiled/untiled with explicit tile sizes
 - Indexed tile access, simple iterator access
 - Duplication and merging of contiguous blocks in DRAM & GPU (CUDA) (i.e. 1D arrays & contiguous blocks of ND arrays)
- Cost model
 - Cost model API implemented, not yet integrated
- Loop analysis
 - Loop description and analysis API implemented, not yet exploited

In progress / planning

- Advanced memory allocation and movement
 - Cost model based movement
 - Tile migration and merge policies
 - Efficient ND movement
 - Flash, storage, other execution contexts
 - FPGA ?
- Irregular arrays
 - Irregular element size or blocking
- Non-contiguous tile transport
 - Efficient transport of non-contiguous tile data
- Layout transformation
 - Changing array layouts for efficient computation or data transport
- Fortran interface, C++ interface
- Asynchronous API
- Loop analysis
- Distributed Mamba Arrays

Summary: High level capabilities

Early:

- Allow programmer to allocate arrays across heterogeneous memories
 - Initial locations/execution contexts supported are: DRAM and GPU (cuda)
- Allow programmer to split array and transport chunks across memories
- Parameterised tiling
- General out of core:
 - Storage to DRAM
 - DRAM to device memory
 - Caching in fast/high bandwidth memory
 - Splitting array across multiple GPUs per node

Upcoming:

- Integration with demonstrator applications
- Integration with other partners work
- Wider range of supported memories / devices
 - Flash, storage, remote, ...?
- Wider range of supported execution contexts
 - OpenACC, OpenMP, HIP, etc
- Optimized & overlapped data movement
- Distributed arrays

Research area:

- Optimising data movement for/as well as loop execution order

Example codes

- Array allocation and copy in DRAM
- 1d tile duplication in DRAM (from DRAM)
- 1d tile duplication in GPU (from DRAM)
- Entire array allocation on GPU
- 2D tiled matrix multiply in DRAM
- User defined logging override
- 2D tiled matrix multiply on GPU
- Multi-GPU offload
- Tile duplication in DRAM (from storage)
- Asynchronous duplication
-

Available

Available but requires refactor/improvement

Under development

Key interactions with partners

- KTH
 - Benchmarking
 - Demonstrator applications
 - Memory provision
- EPCC
 - Loop analysis/cost modelling
 - Distributed Mamba Arrays
- ECMWF
 - Demonstrator applications
 - First technical collaboration meeting earlier this year
- Maestro project
 - Demonstrator application



Fortran Interface

Mamba Fortran Interface

- Designed to mirror that of the C interface as much as possible
- Assumes Fortran 2008 compiler
- Provides a module named mamba
- The module provides:
 - Type definitions
 - The mamba API
 - Supporting KIND values

Module definitions

- Opaque types

mmbMemSpace

mmbMemInterface

mmbLayout

mmbLayoutPadding

mmbArray

mmbTiling

mmbTileIterator

mmbIndex

- These are passed around the API but not accessed in Fortran

Module definitions: constants

- For many enums provided in the C API and the LOG levels
 - **mmb_error** is an example
- KIND values for INTEGERS
 - **mmbSizeKind** (matches size_t in the C interface)
 - **mmbIndexKind** (matches size_t in the C interface)
 - Various kind values for enums used in Fortran user defined types
 - mmbProviderKind**
 - mmbStrategyKind**
 - mmbAccessTypeKind**

Module definitions: user defined types

```
TYPE, BIND(C) :: mmbAllocation
```

```
    TYPE(C_PTR) :: ptr
```

```
    INTEGER(mmbSizeKind) :: n_bytes
```

```
    TYPE(mmbMemInterface) :: interface
```

```
    LOGICAL(c_bool) :: owned
```

```
END TYPE mmbAllocation
```

```
TYPE, BIND(C) :: mmbMemInterfaceConfig
```

```
    INTEGER(mmbProviderKind) :: provider
```

```
    INTEGER(mmbStrategyKind) :: strategy
```

```
END TYPE mmbMemInterfaceConfig
```

```
TYPE, BIND(C) :: mmbSizeConfig
```

```
    INTEGER(kind(MMB_SIZE_INVALID)) ::  
    action
```

```
    INTEGER(c_size_t) :: mem_size
```

```
END TYPE mmbSizeConfig
```

```
TYPE, BIND(C) :: mmbMemSpaceConfig
```

```
    TYPE(mmbSizeConfig) :: size_opts
```

```
    TYPE(mmbMemInterfaceConfig) ::  
    interface_opts
```

```
END TYPE mmbMemSpaceConfig
```

Module definitions: mmbArrayTile type

```
TYPE :: mmbArrayTile
  TYPE(C_mmbArrayTile) :: c_tile ! The C tile structure
  INTEGER :: rank
  TYPE(C_PTR) :: ptr
  ! Lower and upper indices in tile allocation
  INTEGER(mmbIndexKind), allocatable :: lower(:)
  INTEGER(mmbIndexKind), allocatable :: upper(:)
  ! Array lower and upper indices used for slicing and may be useful
  ! for other applications such as halo management and multi-level tiling
  INTEGER(mmbIndexKind), allocatable :: alower(:)
  INTEGER(mmbIndexKind), allocatable :: aupper(:)
  ! Tile dimensions
  INTEGER(mmbIndexKind), allocatable :: dim(:)
  ! Dimension of allocation containing tile
  ! Absolute dimensions, required when tile is not a contiguous allocation
  INTEGER(mmbIndexKind), allocatable :: abs_dim(:)
  LOGICAL is_contiguous
END TYPE mmbArrayTile
```

General comments on Fortran interface

- API calls are SUBROUTINES with an optional final argument returning the error code, eg.

```
INTEGER(mmbErrorKind) :: err  
call mmb_array_tile(mba, chunks)  
call mmb_array_tile(mba, err=err)
```

- Some API calls (as above) use optional arguments where the C function takes a NULL, may need to use keyword arguments.
- The Fortran API uses 1-based indexing for array and tile index space
- Fortran examples are provided in examples/fortran
- The next slides outline other differences

Dimensions object

- The C API uses a Dimensions object which cannot be interfaced directly in Fortran. It is used in array creation, tiling and for tile dimension enquiry.
- The Fortran interface just uses an array ...

```
integer(mmbIndexKind), dimension(2) :: chunks
```

```
chunks = [tile_size_M, tile_size_N]  
call mmb_array_tile(mba_a, chunks)
```

- There is no mmb_dimensions_destroy() in the Fortran API

Iterators

- Mamba has been changed so that iterator positioning functions also return the array tile, the Fortran interface supports this..

call mmb_tile_iterator_first(iterator, tile)

call mmb_tile_iterator_next(iterator, tile)

- Note that tiles can be accessed in tile space via mmb_tile_at_* routines:

eg

mmb_tile_at(mmb_arr, tid, tile)

mmb_tile_at_2d(mmb_arr, tid_i, tid_j, tile)

Accessing tile 'data'

- Metadata is directly accessible in the tile eg tile%rank.
- A pointer to tile array data is set as follows

real, pointer, dimension(:,:) :: tp

call mmb_tile_get_pointer(tile, tp)

tp(tile%lower(1):tile%upper(1),tile%lower(2):tile%upper(2))=0.0

- Note that the pointer may actually point to a larger space than just the tile

Indexing support

- The C API provides index macro access to array elements
- We expect Fortran programmers only to use this for irregular layouts
- To do this in the Fortran API
 - #include “mambaf.h”
 - Pass a **rank 1** array into mmb_tile_get_pointer(tile)
- The index macro should be passed the **tile** and the **pointer**...

```
MMB_IDX_2D(tile_c, tp_c, ei, ej) = &  
MMB_IDX_2D(tile_c, tp_c, ei, ej) + &  
MMB_IDX_2D(tile_a, tp_a, ei, ek) * MMB_IDX_2D(tile_b, tp_b, ek, ej)
```

Caveats and next steps

- Note that the support for wrapping arrays is not standard Fortran
- Next steps
 - Implement C benchmarks in Fortran and check overheads
 - Memory space and programming model support for GPU

Funding for the work is received from the European Commission H2020 program
Grant Agreement No. 801039 (<https://epigram-hs.eu/>)



European
Commission

Horizon 2020
European Union funding
for Research & Innovation



Fraunhofer



tim.dykes@hpe.com
harvey.richardson@hpe.com