MAMBA: Managed Abstract Memory Arrays

V0.1.0
March 26, 2020

Contact: tim.dykes@hpe.com

Table of Contents

1 Overview

The Mamba library is designed to facilitate the use of heterogeneous memories by providing a set of core data structures and abstractions for use by application programmers. Mamba defines an abstract memory model, implemented with an underlying memory library; on top of this model a core data structure is built called a *Mamba array*, which allows the programmer to transparently allocate, access, and transport data across heterogeneous memory tiers.

This document describes the prototype design and implementation of the Mamba library. The memory model and underlying memory manager are described in Section 2; The Mamba array and related data structures are described in Section 3, along with two expected usage models, explicit programmer utilisation via our user API and implicit utilisation via automatic, or user-guided, code transformation. Section 4 describes the current implementation, build structure, user examples, and expected future implementation steps. The prototype implementation is written in C, and so code examples and listings are predominantly also C, with discussion of language support in Section 4.3.

2 Mamba Abstract Memory Model

Mamba contains a memory management component, which provides an abstraction layer and uniform interface to heterogeneous memory. The Mamba infrastructure exploits this component for data management in Mamba arrays. This component may also be exposed to the user, to provide a uniform interface to allocate, access, and free memory, as well as for moving memory between tiers transparently, outside of the scope of a Mamba array.

In addition to providing a high level abstraction layer for the programmer and the Mamba infrastructure, this component also provides a generic memory interface to which external memory management tools can be connected. An internal memory manager is included, as well as an exploratory interface to the SICM memory manager[1].

2.1 Memory Abstraction

The Mamba library is built on the assumption there are a variety of types of memory on a single node, with different characteristics. The memory abstraction layer in Mamba allows heterogeneous memory systems to be described in a uniform manner, to simplify the process of allocating and moving data across such systems.

[1] https://github.com/lanl/SICM

Conceptually, as illustrated in Figure 1, Mamba considers all the available memory as a single abstract memory space, consisting of a group of memory sub-spaces each with a set of defined characteristics, from which data may be allocated. This concept is represented in the Mamba library by four key interacting data structures, illustrated in Figure 2. The remainder of this section describes the role and interactions between each of these data structures.

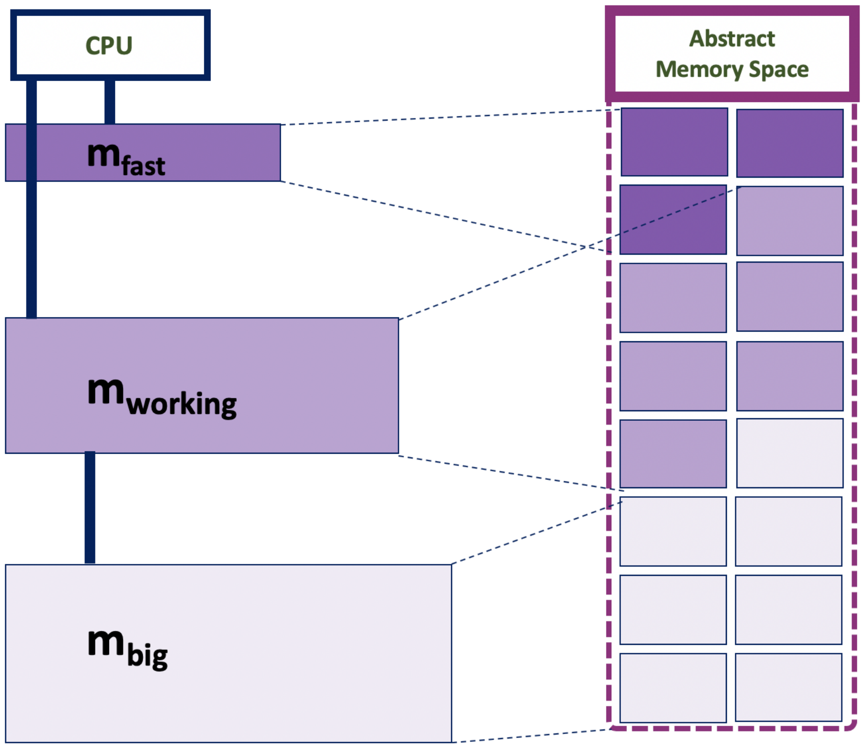

Figure 1: A conceptual illustration of memory abstraction in the Mamba library.

Memory Layer

A Memory Layer, defined in the Mamba API as `mmbMemLayer`, represents a particular *type* of memory with a defined set of characteristics. The memory layers available to a user are defined by the available hardware in the system, and may have characteristics such as high bandwidth or low latency. Such layers may be defined by the user, or discovered automatically where possible during library initialisation.

Memory Space

A Memory Space, defined in the Mamba API as `mmbMemSpace`, represents a size-limited, addressable, instantiation of space corresponding to a specific memory layer. The size may be artificially limited by the user, or hardware-limited. Users will typically

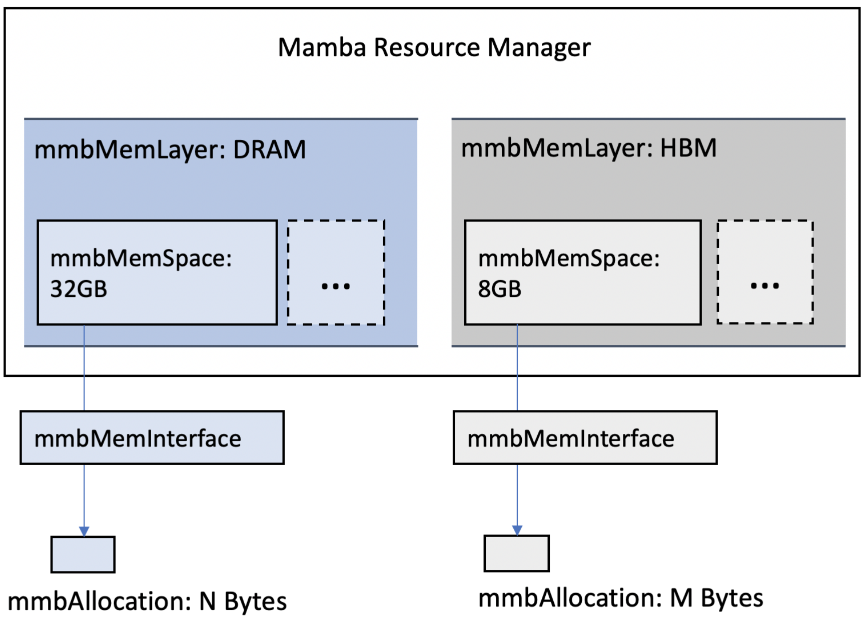

Figure 2: An illustration of the four key interacting data structures that form the memory abstraction in the Mamba library. Two memory layers exist, each with a size-limited memory space. Each space has a corresponding memory interface, from which an allocation object may be obtained.

obtain a memory space by specifying a layer and a size, and allocate a Mamba array in this space as described in Section 3.3. A memory space has an associated *execution context*, `mmbExecutionContext`, that determines how memory will be allocated and made available in that space. The execution context provides the user a means to choose the programming model through which memory must be made available. For example, NVIDIA GPU device memory could be provided within a CUDA, HIP, or OpenACC execution context.

Memory Interface

A Memory Interface, defined in the Mamba API as `mmbMemInterface`, acts as an interface to a specific memory space, providing a mechanism to allocate, move, copy, and free memory. A memory interface may have a specific *strategy*, that defines the behaviour of the interface. For example, this may enforce thread safety during allocation, or define the type of memory allocator used (e.g. pooled vs. slab).

Allocation

An Allocation object, defined in the Mamba API as `mmbAllocation`, provides an abstract container for a memory allocation in a specific memory space. Allocation objects are provided by memory interfaces, and passed into generic allocation, copy, and free routines. Allocation objects contain, if feasible, a real data pointer, along with provenance information such as the interface through which it was allocated, and whether the underlying data is owned by the allocation object or not. The concept of ownership allows, for example, sub-allocations to be created as slices of existing allocations.

2.2 Memory Management

The Mamba API for memory management is based on the abstractions detailed in Section 2.1, and is utilised for managing Mamba array objects detailed in Section 3. A brief overview of the memory management API is presented here, for full API details see the API documentation in the Mamba library.

2.2.1 Memory System Discovery

In order to allocate memory either via a Mamba Array object or explicitly using the allocation API, an appropriate memory space is required. Mamba is able to automatically discover the available memory layers on the system and create an appropriate memory space for each supported layer. This behaviour is enabled by default when an appropriate version (>=2.0) of the `hwloc` library is found during configuration, which is used to analyse system hardware topology during a discovery phase of initialisation. Listing 1 demonstrates the request of an existing memory space from the DRAM layer in the default execution space.

Listing 1: Requesting a memory space in Mamba

```
mmbError err;
mmbMemSpace *dram_space;
err = mmb_request_space(MMB_DRAM, MMB_DEFAULT_EXECUTION_CONTEXT,
  &dram_space);
CHECK_STATUS(err, "Failed to retrieve a dram memory space\n");
```

Once a memory space is acquired, it may be passed to a Mamba Array during construction as shown in Section 3, or may be used to allocate memory explicitly via the API as shown in Section 2.2.2.
Automatic discovery may be disabled, either by default when the `hwloc` library is not found or is at version <2.0, or explicitly by the user via configuration argument `--enable-discovery[=yes|no|default]`. In this case, the available memory layers must be registered by the user. When registering a memory layer, as demonstrated in Section 2.2.2, the memory library will attempt to initialise an appropriate memory interface and will return a memory space if successful.

The benefit of automatic discovery is that the user may query the existing memory system and choose from available memories, whilst memory registration requires the user to know which layers are available and specify them explicitly.

2.2.2 Explicit Management

The Mamba memory API is also available to the user for explicit memory management. The typical process a user, or the Mamba implementation, must follow to **manually** utilise the memory API is:

1. Acquire a memory space S_M for memory layer M, via registration or request.
2. Request an interface I_M from space S_M.
3. Request a memory allocation A_M from interface I_M.

Listing 2 illustrates this process using the prototype Mamba API. This example demonstrates the allocation of a memory buffer both on host and device in a heterogeneous CPU-GPU system, and the transparent copy between buffers.

Listing 2: Memory management API utilisation in Mamba. Error checking is omitted for brevity.

```
size_t n_bytes = 1024;
mmbMemSpace *host_space;
mmbMemSpace *device_space;

/* Manually register two existing memory layers, returning two size limited
  memory spaces for host and device. */
mmb_register_memory(MMB_DRAM, 8000, &host_space);
mmb_register_memory(MMB_GDRAM, 8000, &device_space);

/* Acquire memory interface for host memory */
mmbMemInterface *host_interface;
mmb_memspace_request_interface(host_space, MMB_DEFAULT_MEMINTERFACE,
                               MMB_POOLED, &host_interface);

/* Allocate 1KB of memory on the host */
mmbAllocation *host_buffer;
mmb_allocate(n_bytes, host_interface, &host_buffer);

/* Acquire memory interface for device memory */
mmbMemInterface *device_interface;
mmb_memspace_request_interface(device_space, MMB_DEFAULT_MEMINTERFACE,
                               MMB_POOLED, &device_interface);
/* Allocate 1KB of memory on the device */
mmbAllocation *device_buffer;
```

```
mmb_allocate(n_bytes, device_interface, &device_buffer);

/* Omitted: fill host memory with useful data */

/* Copy data from host memory allocation to device memory allocation */
mmb_copy(device_buffer, host_buffer);
```

3 Mamba Array

A Mamba array is an array-like data structure that forms the core abstraction of the Mamba library. It is built on top of the abstract memory model described in 2, and the underlying data in a Mamba array may be transparently distributed across multiple types of memory as illustrated in Figure 3. Subsets of the array may be duplicated or moved between memories, either explicitly by the user or implicitly by the Mamba runtime. The remainder of this section describes the process to construct, access, and move Mamba arrays.

3.1 Array construction

The constructor for a Mamba array requires four key arguments as shown in `mmb_array_create` signature in Listing 3.

Listing 3: The API to construct and destroy a Mamba array.

```
mmbError mmb_array_create(mmbDimensions *in_dims, mmbLayout *in_layout,
                          mmbMemSpace *in_space, mmbAccessType in_access,
                          mmbArray **out_mba);

mmbError mmb_array_create_wrapped(void *in_data, mmbDimensions *in_dims,
                                  mmbLayout *in_layout, mmbMemSpace *in_space,
                                  mmbAccessType in_access, mmbArray
    **out_mba);

mmbError mmb_array_destroy(mmbArray *in_mba);
```

The mmbDimensions argument, `in_dims`, specifies array size in each dimension, and data will be appropriately allocated during array construction.
The second argument, `in_layout`, defines the layout of the array in physical memory, represented in the Mamba API as an `mmbLayout` data structure. This object defines the mapping of array indices to physical memory layout, and encapsulates array characteristics such as:

- Number of dimensions

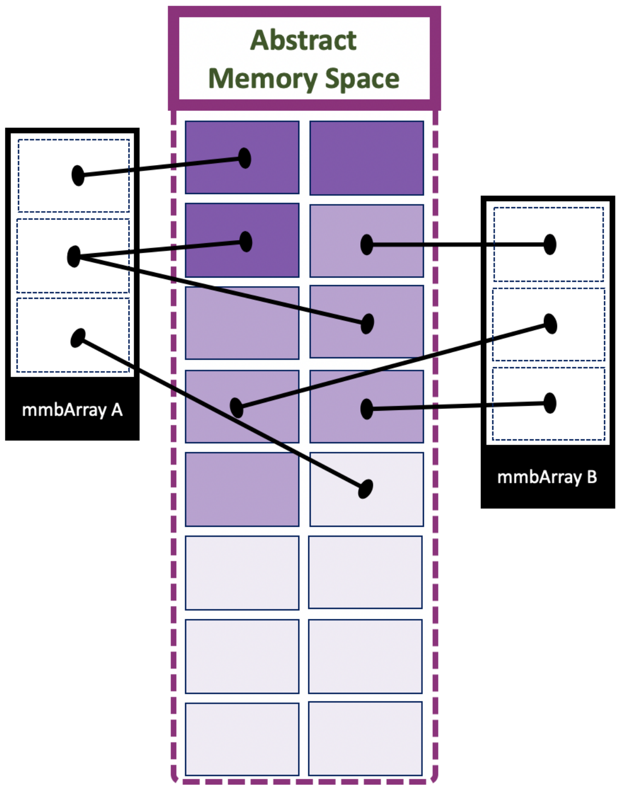

Figure 3: A conceptual illustration of Mamba arrays distributed across the different types of memory in an abstract memory space representing a heterogeneous memory system. Array subsets may reside in one or more memory spaces, explained further in Section 3.2.

- Object type (Element/AOS/SOA) and size in bytes
- 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)

Figure 4 represents how different layouts may map conceptual indices in an `mmbArray` to physical indices in memory, and a series of constructors are provided for typical types of layout as shown in Listing 4, although the prototype library implementation only supports regular layouts at this time.

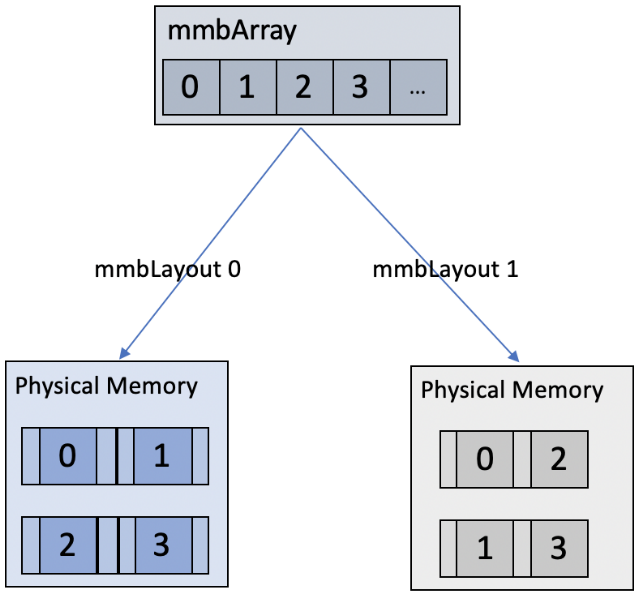

Figure 4: Multiple mappings from `mmbArray` indices to physical memory locations, due to varying `mmbLayout` configurations.

Listing 4: Various constructors for an `mmbLayout` object.

```
mmbError mmb_layout_create_regular_1d(size_t element_size_bytes,
                                       mmbLayoutPadding *pad,
                                       mmbLayout **out_layout);

mmbError mmb_layout_create_regular_nd(size_t element_size_bytes, size_t
    n_dims,
                                       mmbLayoutOrder element_order,
                                       mmbLayoutPadding *pad,
                                       mmbLayout **out_layout);

mmbError mmb_layout_create_block(size_t n_dims, size_t element_size_bytes,
                                  mmbLayoutOrder element_order,
                                  mmbDimensions *block_size,
                                  mmbDimensions *block_order,
                                  mmbLayoutPadding *pad,
                                  mmbLayout **out_layout);

mmbError mmb_layout_create_irregular_1d(size_t element_size_bytes,
                                         size_t *offsets,
                                         size_t *sizes,
                                         mmbLayout **out_layout);
```

The third argument for Mamba array construction, `in_space`, provides the memory space in which the array should be allocated. This may be obtained through

memory registration or discovery, as demonstrated in Section 2.2.
The fourth argument, `in_access`, indicates the *access type* for the array, represented by `mmbAccessType` in the Mamba API. This describes how the array is intended to be used and may be, for example:

- `MMB_READ`
- `MMB_WRITE`
- `MMB_READ_WRITE`
- ...

The access type may influence where the array is placed in a particular memory space, such as using fast read-only memory where applicable, and is further exploited for array subsets as discussed in Section 3.2.
Finally, a Mamba array may also be constructed wrapped around existing user data, using the constructor `mmb_create_wrapped` shown in Listing 3. This allows a user to pass in pre-allocated data for management by the Mamba library, using the additional initial argument `in_data`.

3.2 Array Tiling

In the Mamba library, arrays may be decomposed into subsets for iteration or movement between memory spaces, as illustrated in Figure 5. These subsets are known as *array tiles*, and are represented by the `mmbArrayTile` data structure. The process of decomposing an array into these subsets is known as *tiling an array*, and the set of array tiles that constitute the full array is known as an *array tiling*.
Listing 5 shows the API provided to tile, and untile, a Mamba array. The only argument required to tile an array, beyond the array itself, is a dimensions object `in_dims`, which defines the size of a single tile. The full array will then be decomposed into tiles with these dimensions by the Mamba runtime. As indicated by the API, a tiling is an action applied to an array object, and only a single tiling may be active on an array at any one time. Repeated calls to `mmb_array_tile` with new dimensions will result in removal of any previous tiling applied to the provided array.

Listing 5: Mamba API to tile an Array

```
mmbError mmb_array_tile(mmbArray *in_mba, mmbDimensions *in_dims);
mmbError mmb_array_untile(mmbArray *in_mba);
```

Array tiles may be accessed either directly by requesting a tile by index in the array tiling, or indirectly by requesting and using an iterator over the set of array tiles.

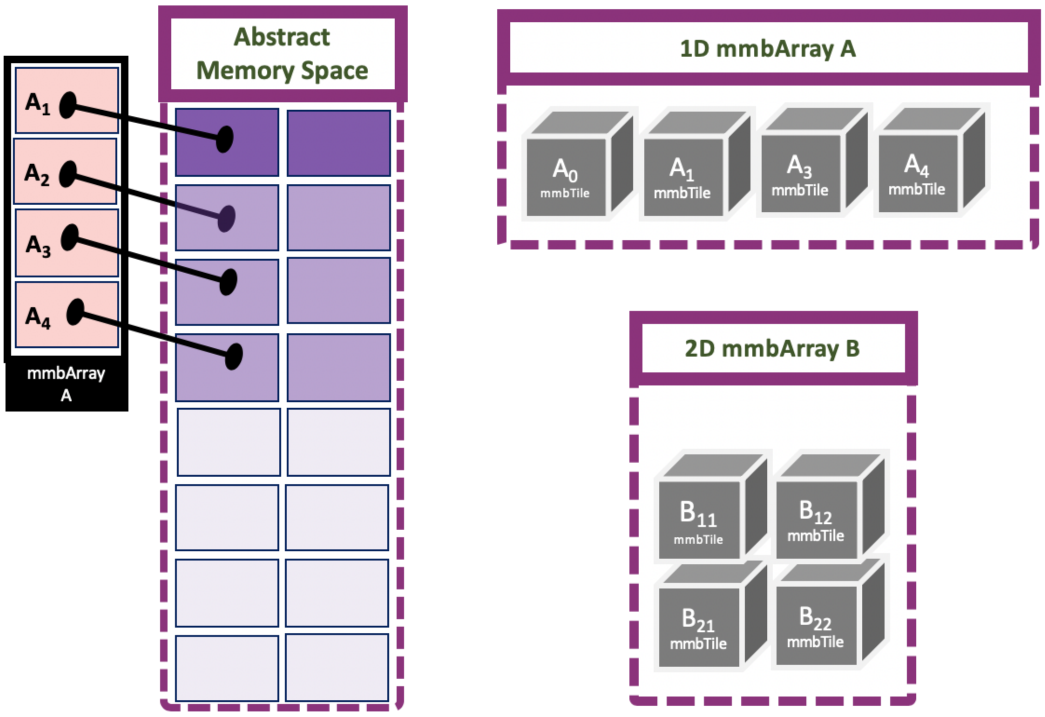

Figure 5: An illustration of the concept of array tiling in one and two dimensions, each tile may reside in one or more memory spaces.

Listing 6 shows the API for direct access by index, which will be demonstrated in Section 3.3.

Listing 6: Mamba API to directly access a specific tile of an array via index.

```
mmbError mmb_tile_at(mmbArray *in_mba, mmbIndex *in_idx,
                     mmbArrayTile **out_tile);
```

Tile iterators, represented by the `mmbTileIterator` data structure, are a more convenient way of iterating over a full array tiling. An iterator object contains an internal schedule over tiles, and provides typical iterator operations (first, next, etc) to traverse the array tiling. The current API for the prototype implementation of array tile iterators is shown in Listing 7. It is expected that a full implementation of the tile iterator objects would allow the Mamba runtime to automatically move array tiles based on knowledge of the iteration schedule, for example pre-fetching one or more tiles.

Listing 7: Mamba prototype API to construct and use an array tile iterator object.

```
typedef struct mmbTileIterator {
  mmbIteratorSchedule schedule;
  mmbIndex idx;
  mmbArrayTile* tile;
  mmbArray* src;
} mmbTileIterator;

mmbError mmb_tile_iterator_create(mmbArray *in_mba, mmbTileIterator** out_it);
mmbError mmb_tile_iterator_first(mmbTileIterator * in_it);
mmbError mmb_tile_iterator_next(mmbTileIterator * in_it);
mmbError mmb_tile_iterator_count(mmbTileIterator * in_it, size_t* count);
mmbError mmb_tile_iterator_destroy(mmbTileIterator * in_it);
```

Once a tile is acquired, either by index or via iterator, it may be used to access the underlying array data. There are two mechanisms to do this, direct access or assisted indexing via convenience macros defined by the Mamba library.
Direct access allows the user to retrieve a raw pointer from the array tile, and index it directly using the indexing bounds stored directly in the tile structure as shown in Listing 8. Alternatively, one may use the indexing macros provided by Mamba to simplify the indexing calculation for common cases, as shown in Listing 9.

Listing 8: A partial list of the accessible members of a tile structure.

```

typedef struct mmbArrayTile {

  // Lower and upper indices relative to the tile allocation pointer
  size_t *lower;
  size_t *upper;
  // Tile dimensions
  size_t *dim;

  // Global array lower and upper indices, relative to the array allocation
    pointer
  size_t *alower;
  size_t *aupper;

  // Absolute dimensions, required when tile is not a contiguous allocation
  size_t *abs_dim;
  // ...
} mmbArrayTile;
```

Listing 9: Looping over a 2D array tile and accessing members using a convenience indexing macro to zero-initialise the tile.

```
for(size_t i = tile->lower[0]; i < tile->upper[0]; i++)
  for(size_t j = tile->lower[1]; j < tile->upper[1]; j++){
    MMB_IDX_2D(tile, i, j, float) = 0;
  }
```

Array tiles are the mechanism by which arrays may be segmented and transported between physical memory tiers. The Mamba library provides features for three types of movement:

- Array tile duplication
- Array tile merging
- Array tile migration

Array tile *duplication* provides a means of creating a duplicate of an array tile in another (or even the same) memory space. A new block of memory is allocated (or taken from an internal cache) to store the new tile, and data is copied from the original tile. A reference to the original tile is maintained, however this tile is not accessible to the user.

Array tile *merging* provides a means of merging a duplicated tile back to the original tile from which it was duplicated. A strategy, `mmbMergeStrategy`, defines how this merge will take place. In the prototype library implementation, the only strategy supported is to overwrite the original data. Other strategies are envisioned such as typical reduction operators or the provision of a custom user function to merge array indices.

Array *tile migration* provides a means of migrating an array tile to a different memory space. A new block of memory is allocated (or taken from an internal cache) to store the new tile, and data is copied from the original tile. The new tile replaces the original tile, and the original tile reference is dropped, releasing associated memory where possible.

Figure 6 illustrates the difference between duplicating an array tile, and migrating an array tile, whilst Listing 10 shows the API available for tile movement. For duplication and migration, a target memory space is required, along with an access type and layout for the new tile. The access type may be used for optimised memory placement of the new tile, or to optimise the data movement requirements as illustrated in Figure 7. For a subset of layouts, it is expected that a new layout may be provided here and an automatic transformation will occur during duplication/migration. The prototype library so far only supports passing a tile layout equivalent to the existing array layout. In each case, the tile returned will be accessible in the execution context of the memory space requested.

Listing 10: Mamba API for the three types of tile movement.

```
mmbError mmb_tile_duplicate(mmbArrayTile *in_tile, mmbMemSpace *in_space,
                            mmbAccessType in_access, mmbLayout *in_layout,
                            mmbArrayTile **out_tile);

mmbError mmb_tile_migrate(mmbArrayTile *in_tile, mmbMemSpace *in_space,
                          mmbAccessType in_access, mmbLayout *in_layout);

mmbError mmb_tile_merge(mmbArrayTile *in_tile, mmbMergeType in_merge);
```

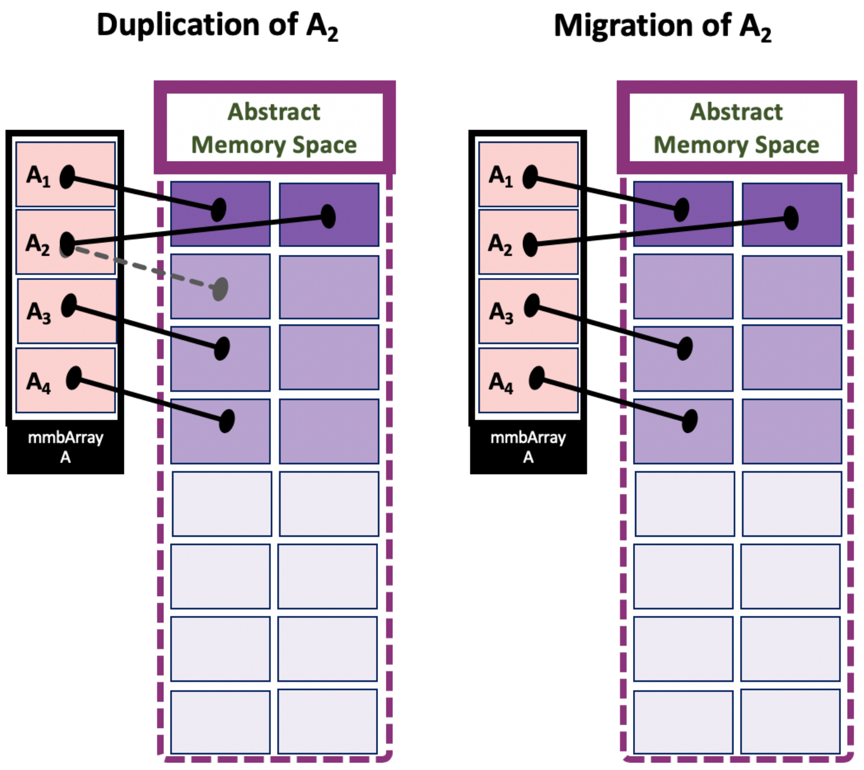

Figure 6: An illustration of the difference between duplicating and migrating array tiles. In the case of duplication, a copy of the original tile still exists, whereas in the case of migration the original tile is discarded.

Figures 8 and 9 document tile behaviour when duplicating, migrating, and merging for synchronous, and asynchronous versions of the tile movement API via state machine. The Mamba prototype library does not yet support the asynchronous API version.

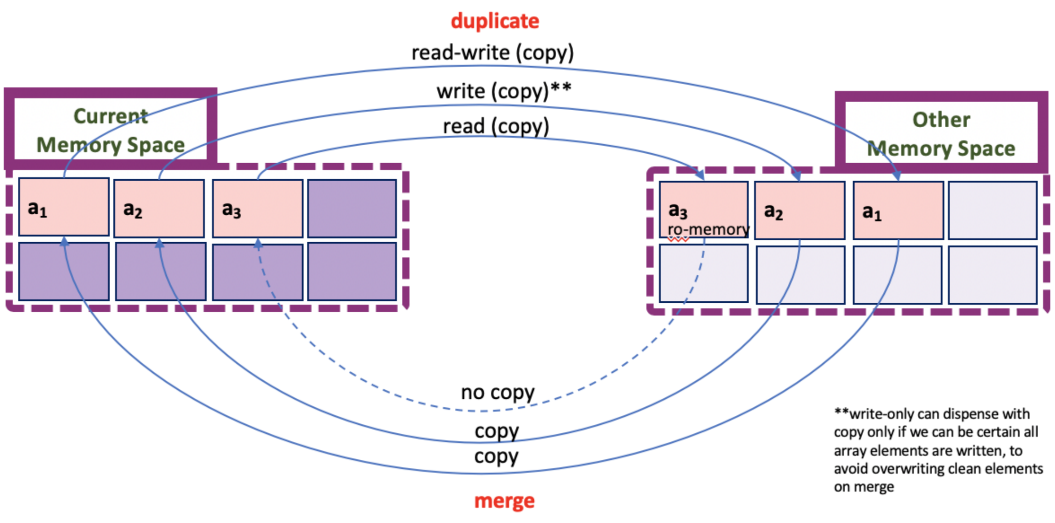
duplicate
read-write (copy)
write (copy)**
read (copy)
Current Memory Space
Other Memory Space
a1
a2
a3
a3
ro-memory
a2
a1
no copy
copy
copy
merge
**write-only can dispense with copy only if we can be certain all array elements are written, to avoid overwriting clean elements on merge

Figure 7: An illustration of the potential for simple automatic data movement optimisation based on the provision of access types during tile duplication.

3.3 Explicit Tiling

As discussed in Section 1, two use cases of the Mamba library are envisioned, *explicit* and *implicit* tiling. The explicit use case requires the user to directly modify their code using the Mamba API, re-implementing part of their application to exploit the Mamba library. Using this approach, the user may:

- Allocate and transport data across heterogeneous memories in a uniform and performance-portable manner.
- Implement general out-of-core algorithms using Mamba data structures.
- Experiment with tile-based kernel optimisation in a parameterised way.
- Benefit from limited automatic data movement via tile iterator objects.

The steps a user is required to take are, at least:

1. Initialise the Mamba library
2. Register/discover available memory
3. Acquire one or more memory space handles
4. Construct a Mamba array
5. Tile the array

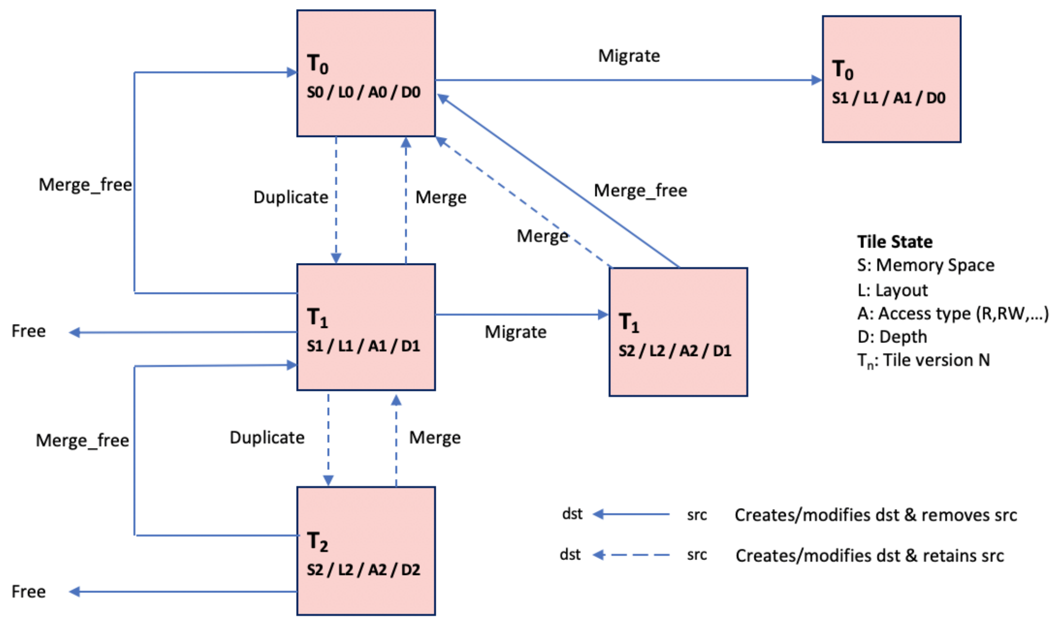

Figure 8: A state machine describing array tiles during duplication, migration, and merging. Only the maximum depth tile is accessible at any one time.

6. Iterate over tiles
7. Duplicate/migrate if required
8. Access tile data
9. Free tile
10. Destroy array

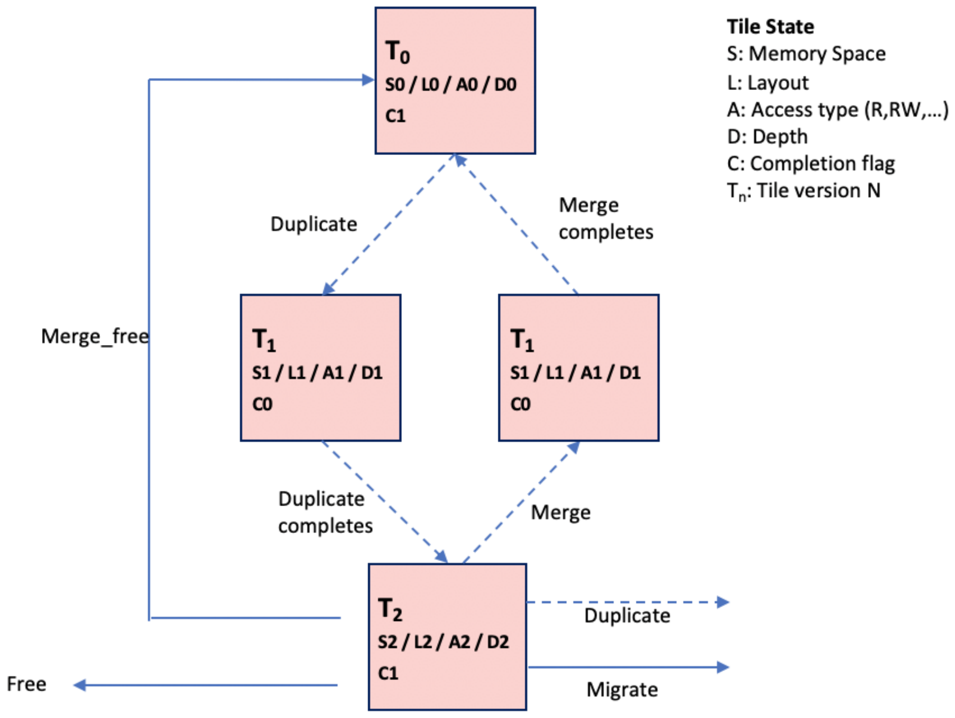

Figure 9: An asynchronous version of the state machine in Figure 8, where a completion flag is used to determine the state of array tiles during asynchronous movement.

Listing 11 demonstrates current Mamba API to perform this set of steps. In the prototype API, usage is more verbose that envisioned in the final implementation. Error handling is also omitted for brevity, and discussed further in Section 4.2.

Listing 11: Prototype Mamba API for a typical user scenario, constructing a Mamba array, tiling and iterating over the tiles, passing each tile in turn to a computational kernel.

```
size_t array_size = 32;
size_t tile_size = 8;

// Initialise mamba and register some memory
stat = mmb_init(MMB_INIT_DEFAULT);
mmbMemSpace *dram_space;
stat = mmb_register_memory(MMB_DRAM, 8000, MMB_DEFAULT_EXECUTION_CONTEXT,
  &dram_space);

// Create mmb array
mmbArray *mba;
mmbLayout *layout;
size_t arrdims[1] = {array_size};
mmbDimensions dims = {1, arrdims};

mmb_layout_create_regular_1d(sizeof(float), MMB_PADDING_NONE, &layout);
mmb_array_create(&dims, layout, dram_space, MMB_READ_WRITE, &mba);
init_matrix_buffer(mba);

// Request tiling of all arrays with chunkx chunky
size_t chunkdims[1] = {tile_size};
mmbDimensions chunks = {1, chunkdims};
stat = mmb_array_tile(mba, &chunks);

// Loop over tiles using standard indexing
// Duplicate each tile, write to it, and merge back to original
mmbArrayTile *tile;
mmbIndex *idx;
mmbDimensions *tiling_dims;
mmb_index_create(1, &idx);
mmb_dimensions_create(0, &tiling_dims);
mmb_tiling_dimensions(mba, tiling_dims);

for (size_t ti = 0; ti < tiling_dims->d[0]; ++ti) {
    // Set tile indices for c array
    stat = mmb_index_set(idx,ti);
    stat = mmb_tile_at(mba, idx, &tile);
    // Duplicate tile in DRAM; write to and merge duplicated tile
    mmbArrayTile *duplicate_tile;
    mmb_tile_duplicate(tile, dram_space, MMB_READ_WRITE, layout,
  &duplicate_tile);
    write_to_tile(duplicate_tile);
    mmb_tile_merge(duplicate_tile, MMB_OVERWRITE);
  }

  // Cleanup intermediate objects, check result buffer, cleanup mamba
  mmb_dimensions_destroy(tiling_dims);
  mmb_index_destroy(idx);

  check_result(mba);
  mmb_array_destroy(mba);
  stat = mmb_finalize();

  return EXIT_SUCCESS;
}

mmbError write_to_tile(mmbArrayTile *tile)
{
  // Fill array tile with 1's
  for (size_t i = tile->lower[0]; i < tile->upper[0] ;++i) {
    MMB_IDX_1D(tile, i, float) = 1.f;
  }
  return MMB_OK;
}
```

3.4 Implicit Tiling

The second use case envisioned for the Mamba library is implicit usage. In this case, direct modification of user-code is minimal, for example use of simple directives. The Mamba library may extract information about data movement from these directives, and/or the code itself, and manage tile movement appropriately. Furthermore, through compiler extensions, the Mamba API may also be inserted automatically to take advantage of Mamba tiling and automatic data movement. Using this approach, the user may potentially:

- Automatically exploit the Mamba API via a minimal amount of code markup.
- Benefit from automatic data movement where appropriate.
- Benefit from loop kernel optimisation where appropriate.

This use case is to be researched during the implementation and integration stages of the project, and is not expected to be production ready by the end of the project.

This is due to both the complexity and immaturity of the approach, and the dependence on readiness of existing and upcoming compilers and tools, for example the new Fortran front end to the LLVM compiler. The current prototype of the library includes a loop description API, and a loop analysis module based on the Polyhedral Extraction Tool (PET) [1] and the ISL library [2]. Prototype tooling implementation has also been developed within the new Flang (formerly F18) front end for LLVM.
The loop description API allows a user to describe a computational loop in terms of data references and accesses to these data references. Internally, an incomplete abstract syntax tree (AST) is built describing the loop. The tree is incomplete because only ordered data accesses are considered, rather than describing the control flow in full, e.g. a read and write may be considered, but not the assignment operator that caused them. Listing 12 demonstrates description of a simple loop using this API. At the point `mmb_compute_loop` is called, the internal Mamba AST is fully constructed, and translated to a PET AST, which is then used to perform polyhedral loop analysis using the PET library. An extended example, including resultant output, is included in the Mamba library, the `loop_description` example detailed in Section 4.4.

Listing 12: Example use of the loop description API to describe a simple loop.

```
const int M = 100;
float *array_A = malloc(sizeof(float) * M);
float alpha = 2.0;

// Omitted: array_A initialisation

// Create array and data reference for array and scalar
mmb_data_ref_create_array(array_A, 2, sizeof(float), "A", &ref_A);
mmb_data_ref_create_scalar(&alpha, sizeof(float), "alpha", &ref_alpha);

// Create loop nest object
mmbLoopNest *loop;
mmb_loop_nest_create(&loop);
// Add outer loop
mmbLoopId *outer;
mmb_loop_nest_add_loop(loops, &outer);

// Create some reuseable expressions
mmbExpr *zero, *i;
mmb_expr_create_int(0,&zero);
mmb_expr_create_iterator("i",&i);

// Set init, condition, and increment expressions for outer loop
mmbExpr *init;
```

```
mmb_expr_create_binary_decl_op(MMB_OP_ASSIGN, i, zero, &init);
mmb_loop_set_loop_init(outer, init);

mmbExpr *cond;
mmbExpr *expr_M;
mmb_expr_create_const_parameter("M", &expr_M);
mmb_expr_create_binary_op(MMB_OP_LT, i, expr_M, &cond);
mmb_loop_set_loop_cond(outer, cond);

mmbExpr *inc;
mmb_expr_create_unary_op(MMB_OP_PRE_INC, i, &inc);
mmb_loop_set_loop_inc(outer, inc);

// Create array index access expression
mmbExpr * i_access;
mmb_expr_create_access(i, &i_access);

// Add read access statement to array A
mmb_loop_add_access(inner, ref_A, i_access, MMB_EXPR_ACCESS_READ);
// Add read access statement to scalar alpha
mmb_loop_add_access(inner, ref_alpha, NULL, MMB_EXPR_ACCESS_READ);
// Add write access statement to array A
mmb_loop_add_access(inner, ref_A, i_access, MMB_EXPR_ACCESS_WRITE);

// Loop analysis, dependency computation, output etc, all happens here
mmb_loop_nest_compute(loop);

// Actual loop
for(int i = 0; i < M; ++i)
    A[i] *= alpha
```

The loop description API is clearly verbose, and it is not expected that a user will write this code themselves outside of an interim experimentation phase. However, this does provide a means of describing the accesses in a loop, which would be beneficial to the development of cost models for data access. It is envisioned that the main use case of this API will be exploitation in an extension to the currently under-development LLVM Fortran compiler front end, to allow Mamba-based polyhedral loop analysis on Fortran kernel loops. We are also exploring other mechanisms for capturing array accesses in loops that do not rely on compiler extensions such as this.

4 Library Implementation

In this section we briefly describe the structure and build process of the library prototype, the common utilities, along with existing and proposed language support.

4.1 Structure and Build Process

Figure 10 illustrates the directory structure of the Mamba library. At the top level, build files and a README file are provided, whilst most of the Mamba implementation is in the `common` subdirectory. Dependent libraries for loop analysis, stored in the loopanalyzer subdirectory, are imported via `git submodules`.

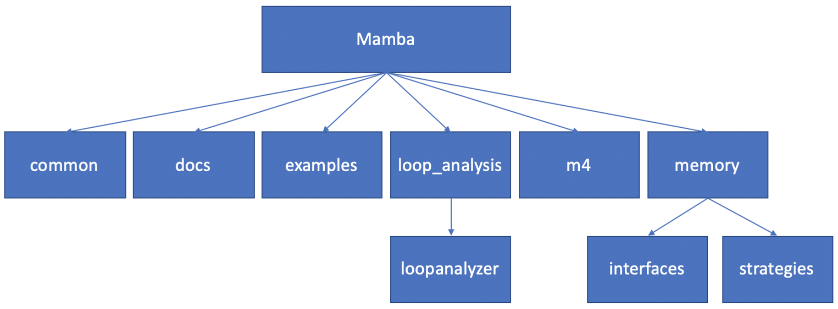

Figure 10: An illustration of the directory structure of the Mamba library.

The Mamba library exploits the `autotools` build system for compilation; each of the code subdirectories is built as a convenience library, and these are combined to form `libmamba`. Listing 13 shows the typical user build process.

Listing 13: The typical user build process for Mamba.

```
mkdir build;
cd build;
../autogen.sh;
../configure [--prefix=/p/a/t/h]        \
             [--with-sicm=/p/a/t/h]     \
             [--enable-cuda[=yes|no]]   \
             [--with-loop-analysis[=yes|no]];
make;
make check;
```

4.2 Common Utilities

Library Initialisation

The Mamba library must be initialised before first use, and de-initialised after final use. To this end, the library includes functions for initialisation, finalisation, and checking the state of the library, in a similar way to the MPI library. This API is shown in Listing 14.

Listing 14: API for one-time initialisation, finalisation, and state checking in the Mamba library.

```
mmbError mmb_init(mmbOptions *in_opts);
mmbError mmb_is_initialized(int *initialized);
mmbError mmb_finalize();
```

The structure `mmbOptions` may consist of a series of flags for Mamba configuration, although in the prototype library there are not yet any configuration options made available to the user. A constructor for default configuration options exists, such that the user may provide `MMB_INIT_DEFAULT` as the argument to `mmb_init` for default initialisation.

Convenience Data Structures

Mamba includes various convenience data structures for interacting with the library. Commonly used in the public API are two wrapper structures for a simple resizeable array used to store either a list of dimensions, or a list of indices,

- `mmbDimensions`
- `mmbIndex`

Listing 15 highlights part of the API available for this type of data structure, this may be extended as necessary during the full library implementation.

Listing 15: A partial list of API available for convenience data structure mmbDimensions.

```
mmbError mmb_dimensions_create(const size_t ndim, mmbDimensions **out_dims);
mmbError mmb_dimensions_create_fill(const size_t ndim, const size_t *values,
       mmbDimensions **out_dims);
mmbError mmb_dimensions_copy(mmbDimensions *dst, const mmbDimensions *src);
mmbError mmb_dimensions_resize(mmbDimensions *in_dims, const size_t ndim);
mmbError mmb_dimensions_destroy(mmbDimensions *in_dims);
```

Logging

Mamba includes a logging infrastructure (`./common/mmb_logging.h/c`, used internally and available for users to exploit. Maximum logging verbosity is defined by default with a compile time switch, and reduced verbosity may be requested by setting the environment variable `MMB_LOG_LEVEL` to one of the values shown in Listing 16.

Listing 16: A list of available logging functions with example usage and output.

```
/** log level for errors */
#define MMB_LOG_ERR 0
/** log level for warnings */
#define MMB_LOG_WARN 1
/** log level for informational messages */
#define MMB_LOG_INFO 2
/** log level for debugging messages */
#define MMB_LOG_DEBUG 3
/** log level for really chatty logging */
#define MMB_LOG_NOISE 4

/** error messages */
ERR(format, ...)
/** warning messages */
WARN(format, ...)
/** informational messages */
INFO(format, ...)
/** debug messages */
DEBUG(format, ...)
/** chatty messages */
NOISE(format, ...)

/** example usage */
ERR("Error with integer value: %d\n", errno)
/** example output in format:
    [error type] (process id): function(filename:line) error message */
[E] (12083): main(example.c:25) Error with integer value: 1
```

Error Handling

Error handling is managed via datatype `mmbError`, used as the return type for almost all Mamba API functions, and demonstrated in Listing 17.

Listing 17: A typical example of error handling in the Mamba library.

```
mmbError mmb_example_function_wrapper(void) {

    mmbError err = mmb_example_function(arg1, arg2);
    if(err != MMB_OK) {
        ERR("Failed to run the example function\n");
        goto BAILOUT;
    }

    /** ... */

BAILOUT:
    return err;
}
```

4.3 Language Support

The prototype implementation of the library is written in C, based on the C99 language standard. As evidenced throughout the code listings in this document, we adhere to a programming style where:

- Functions are prefixed `mmb_`
- Spaces in function names are represented via underscore.
- Functions return an *mmbError* type for error handling.
- Typenames are camel case.
- Constructors are formatted `mmb_(object_name)_create`, with an optional further configuration postfix (e.g. `_2d`) for overloading constructors.
- Destructors are postfixed `mmb_(object_name)_destroy`
- Object memory allocation/free routines are similarly formatted `mmb_(object_name)_alloc`, `mmb_(object_name)_free`.
- Commenting in doxygen format is encouraged

Fortran

The Fortran interface is designed to mirror that of the C API and in most cases uses identically-named types to the C API for declaration of opaque objects that are

passed to Mamba API procedures. The Fortran interface makes use of the latest Interoperability features defined in the Fortran standard (Fortran 2008). New derived types are provided for objects where direct access to the components is needed, this is true for example for the `mmb_Tile` type which provides corresponding elements to the `mmb_ArrayTile` structure in the C API.
The C API provides access to tile information directly and to tile array elements via indexing macros defined by the preprocessor. In the Fortran API we could provide similar macros but that would not be natural for Fortran. In Fortran, we will provide direct indexing of a pointer that points to the array elements. This pointer can also be multidimensional since Fortran supports multidimensional arrays.
The following example illustrates some features of the Fortran API...

Listing 18: Memory management API utilisation in Mamba Fortran Interface

```
program array_copy_wrapped_1d
  use mamba
  USE, INTRINSIC :: ISO_C_BINDING, only : C_loc
  implicit none
  integer, parameter :: M=128
  integer(mmbErrorKind) err
  integer(mmbSizeKind) ntiles
  type(mmbMemSpace) dram_space
  type(mmbArray) :: mba0
  type(mmbDimensions) dims
  type(mmbLayout) layout
  type(mmbTileIterator) it
  type(mmbTile) tile
  real, dimension(:), allocatable,target :: buffer0
  integer :: i,itile,chunksize

  print *,"Starting example 1d_array_copy (Fortran)"

  allocate( buffer0(m) )

  call mmb_init(err)
  call mmb_register_memory(MMB_DRAM, 8000_mmbSizeKind, &
                           MMB_DEFAULT_EXECUTION_CONTEXT,&
                           dram_space,err)
  call mmb_dimensions_create_fill(1_mmbIndexKind,[int(M,mmbIndexKind)],dims)
  call mmb_layout_create_regular_1d(int(storage_size(buffer0),mmbSizeKind),&
                     mmb_layout_padding_create_zero(), &
                     layout,err)
  call mmb_array_create_wrapped(c_loc(buffer0(1)),dims, layout, &
                               dram_space, MMB_READ_WRITE, mba0,err )
  call mmb_array_tile(mba0, dims, err)
  call mmb_tile_iterator_create(mba0, it, err)
  call mmb_tile_iterator_first_with_tile(it,tile,err)

  do i=tile%p_lower(1)+1,tile%p_upper(1)
    tile_%p(i)=i*0.001+3.0
  end do

  call mmb_finalize(err)

end program array_copy_wrapped_1d
```

The Fortran interface requires the use of the `mamba` module to provide the definitions of the API subprograms and support types. In addition to required types, kind values are defined for integer declarations: `mmbErrorKind` for the return error value, `mmbSizeKind` for memory counts and `mmbIndexKind` for quantities that relate to array indexes.
The Fortran interface uses subroutines instead of functions with the last argument used to return an error code. Returning the error code is optional. The example allocates a buffer, registers it with mamba and then creates a layout and mamba array.
A mamba iterator is used to get the first tile in the array, its location in the larger array can be determined from elements of the `mmbTile` type. This type also provides a pointer to the data (in this example we illustrate the initial prototype where this is the full array.
The interface will define pointers that cover the relevant array range to return only the tile. The prototype as yet does not support a large range of data types but this is work in progress.

C++ Interface

The initial C++ interface will be a C++ wrapper of the C interface, due to the similarity of the languages this will add a first phase of C++ support for little additional cost. Following this, more advanced features of C++ will be explored to provided a cleaner interface more suited to C++ programming. At the basic level, this will include using function overloading, templates and, to some extent, object orientated design to provide a lightweight and less verbose API than is possible in C. As implementation progresses, we will also explore the effect of overloaded array operators for tile indexing, while ensuring to maintain the possibility of compiler vectorization of the array accesses, and investigate the applicability of more modern C++ language features, for example storing loop kernels as lambdas for delayed execution in implicit tiling scenarios.

4.4 Examples

A series of examples are included in the prototype implementation, stored in the examples subfolder (`install_dir/examples/`), each existing example is briefly described here with instructions on use. As implementation develops, more examples will be implemented here.

1d_array_copy
This example demonstrates the construction, tiled initialisation, and copy of a 1d mamba array to another 1d mamba array with matching layout and size, with full error checking.

Source file: `examples/1d_array_copy.c`

Usage: `./1d_array_copy`

1d_array_copy_wrapped
This example demonstrates the construction of a 1d mamba array by wrapping an existing user pointer. Similarly to `1d_array_copy`, this is followed by tiled initialisation, and copy of the 1d mamba array to another mamba array with matching layout and size.

Source file: `examples/1d_array_copy_wrapped.c`

Usage: `./1d_array_copy_wrapped`

tile_duplicate
This shows construction of a 1d mamba array, demonstrates a tiled array loop where the tile is duplicated (in the same memory space), operated on by an artificial kernel, and merged back to the original tile.

Source file: `examples/tile_duplicate.c`

Usage: `./tile_duplicate`

matrix_multiply
This demonstrates a tiled matrix multiply using three mamba arrays constructed on top of pre-initialised matrix buffers with random values or, optionally, identity values. All arguments in the usage example are optional.

Source file: `examples/matrix_multiply.c`

Usage: `./matrix_multiply -v (for verbose mode) -t N (for tile size NxN) -m N (for matrix size NxN) -i (use identity for matrix B)`

loop_description
This example demonstrates the description of a loop using the loop description API described in Section 3.4, followed by polyhedral analysis of the loop with dependence computation. The loop description, auxiliary analysis information and calculated loop dependencies are output to the terminal.

Source file: `examples/loop_description.c`

Usage: `./loop_description`

report_mem_state
This example demonstrates dumping the memory system topology after automatic discovery during Mamba initialisation. When run, this example will list the available memory spaces and configuration options found during memory discovery.

Source file: `examples/report_mem_state.c`

Usage: `./report_mem_state`

5 Planned Features

Here we list a series of planned features for upcoming Mamba releases, sorted approximately by expected order of implementation. With each new release, this document will be updated to detail the new features and examples as necessary.

5.1 Short term

- **Full GPU tiling support**: Whilst copying data to and from the GPU is possible using either the tile duplication or manual memory management API, the tiling support is incomplete and requires tile metadata to be automatically transported with tile data, and made available in the appropriate GPU execution context.

- **Fortran interface**: A Fortran interface is under development, and expected to be included in one of the next releases.

- **Efficient non-contiguous tile management**: Extraction and movement of non-contiguous sections of tile data to be implemented in an efficient manner.

- **Non-volatile memory support**: Integration of the uMMapIO software as an external memory manager, to be completed in collaboration with KTH in the context of the EPiGRAM-HS project, will introduce support for non-volatile memory.

- **Asynchronous tile movement**: Implementation of async version of tile movement API.
- **Basic layout transformation**: We expect to implement API to support transformation of a subset of array layouts.

5.2 Medium Term

- **Advanced layout transformation**: We expect to implement API to support transformation of a wider range of layouts, including distributed arrays.
- **Inter-node transport**: In collaboration with WP2 of the EPiGRAM-HS project, we expect to add features for distributed arrays and transport between nodes.
- **Advanced tile scheduling**: Only a single tile schedule is currently supported, we plan to extend this to allow customisation tile scheduling.
- **C++ interface (native)**: Existing C++ support is a simple C API wrapper, we expect to extend this to utilise C++ features where possible to simplify the API usage in C++.

5.3 Long Term

- **Implicit tile management**: As detailed in Section 3.4, implicit tile management is expected in the longer term. This may include directive based tiling and automatic insertion of tile API.
- **Compiler-integrated loop analysis**: We are also exploring compiler integration for both C and Fortran to exploit advanced loop analysis to improve implicit tile management.

References

[1] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *In Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12*, 2012.

[2] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software – ICMS 2010*, pages 299–302, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.