13th ANNUAL WORKSHOP 2017

EXTENDED MEMORY WINDOWS

Tzahi Oved, Alex Margolin

Mellanox Technologies

[March, 2017]

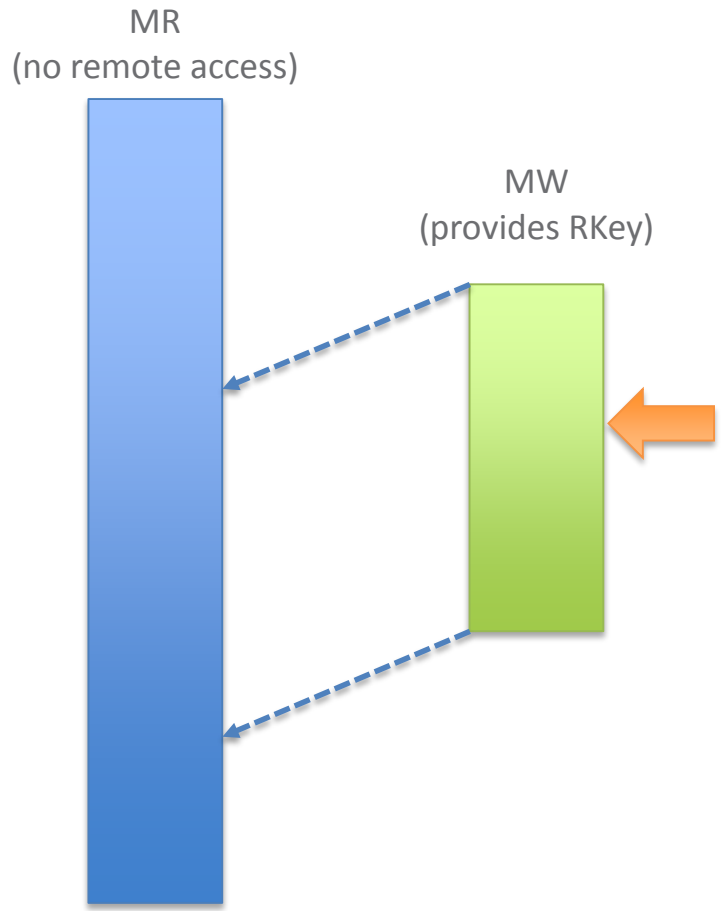
AGENDA

- **What is Memory Window**
- **Motivation for extending Memory Window**
- **Proposed solution and address patterns**
- **API Proposal**
- **Examples and use cases**
- **Summary**

WHAT IS MEMORY WINDOW?

- **Motivation: efficient, safe remote transactions**
  - Register once local memory buffer with no remote access
  - Provide peer proper access rights and aperture only throughout the duration of the expected transaction

- **Open a "window" into an existing MR with elevated remote rights**
  - Does not alter or consume new translation resources
  - Not a privileged operation
    - Accomplished through posting a WR to the Send Queue



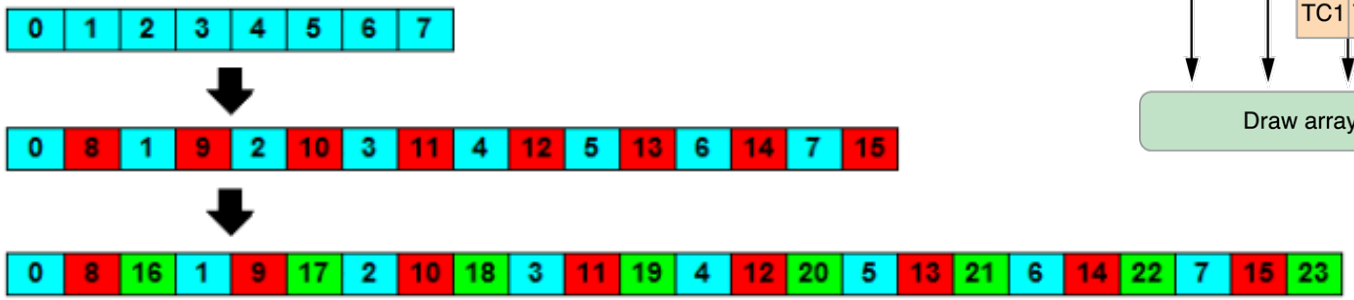
MEMORY WINDOWS FROM APP PERSPECTIVE

- **Application can have more flexible control over remote access to its memory**
  - Efficient grant and revoke remote access rights to a registered Region in a dynamic fashion
  - Grant different remote access rights to different remote agents and/or grant those rights over different ranges within a registered Region
- The operation of associating an MW with an MR is called Binding
- Different MWs can overlap the same MR (with different access permissions)

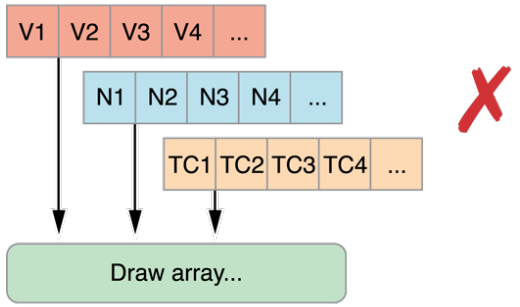
- **Supported actions:**
  - Allocate a Memory Window
  - Query a Memory Window
  - Bind a Memory Window
    - MW Type 1 – bound by calling a function, same scope as the memory region.
    - MW Type 2 – bound by post-sending a WR, bound to the QP it was sent on.
  - Deallocate Memory Window

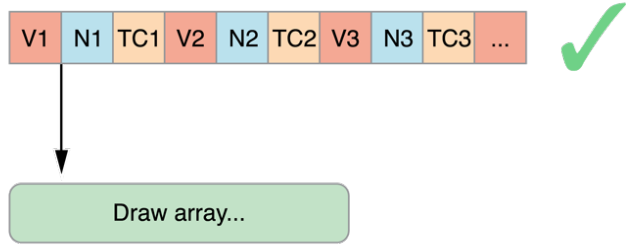
MOTIVATION FOR EXTENDING

- **Apps often reuse a non-contiguous memory layout, for example:**
  1. HPC – Boundary exchange, Collective operations (e.g. Bruck Alltoall algorithm)
  2. Vector Graphics – Interleaved Vertex Data
  3. Scientific – Matrix operations (see next slide)



Reference: Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems, Bruck et el.





Reference: OpenGL ES Programming Guide – "Best practices"

- **The motivation: a compact, reusable memory layout description**
  - Describe the memory layout once
  - Use like a stencil, applied at different base-pointers

MATRIX MULTIPLICATION EXAMPLE

- **The goal: share slices of a matrix for multiplication**
  - The problem: sending a column may not be efficient, because it's not contiguous in memory
    - In the next example, we need a row from A and a column from B to calculate C[0][0]:

A

[0,0]	[0,1]	[0,2]	[0,3]	[0,4]

X

B

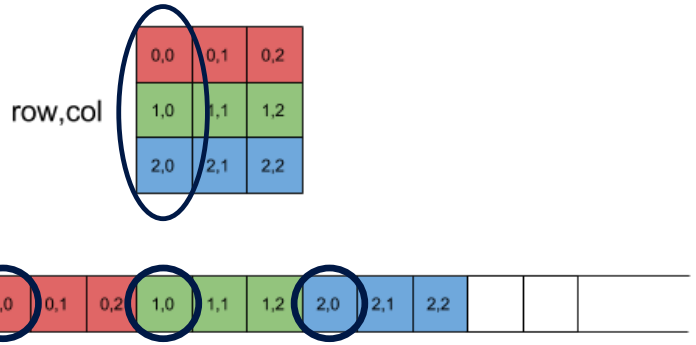
[0,0]				
[1,0]				
[2,0]				
[3,0]				
[4,0]				

=

C

[0,0]				

    - Notice that while the row from A is contiguous, the column from B is not:

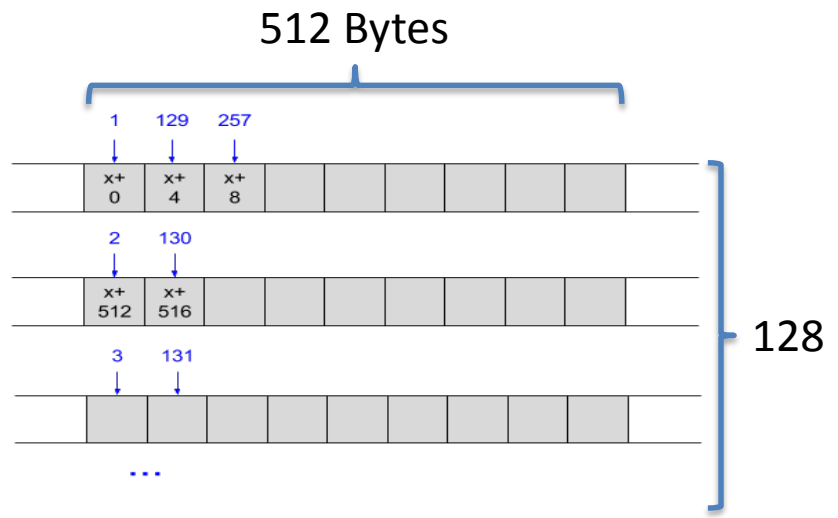


HOW DO WE SEND A TYPICAL PATTERN TODAY?

1. List of Scatter-Gather entries

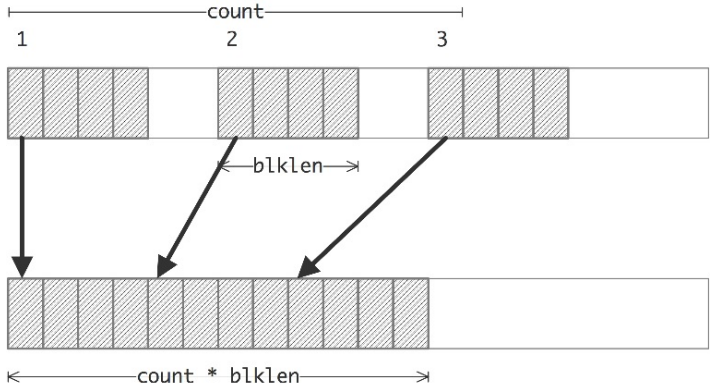
- For example, one column in a Matrix of size N x N requires N entries
- Adverse caching effect, since subsequent cells are 512 Bytes apart
- Long sg list usage

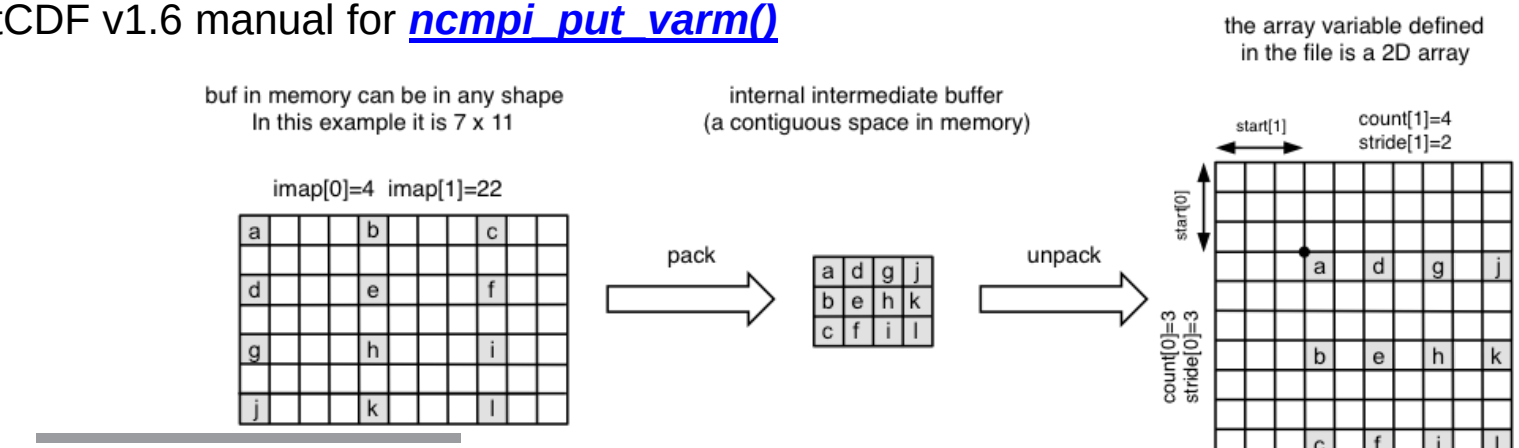
```
/* matrix[ROW#][COL#] */
struct ibv_sge first_column[N];
first_column[0].addr = &matrix[0][0];
first_column[0].length = CELL_SIZE;
first_column[1].addr = &matrix[1][0];
first_column[1].length = CELL_SIZE;
...
first_column[N-1].addr = &matrix[N-1][0];
first_column[N-1].length = CELL_SIZE;
```



2. "User-level packing"

- Data is copied to a contiguous intermediate buffer - No longer zero-copy
- The illustration on the right – from PnetCDF v1.6 manual for ***ncmpi_put_varm()***





PROPOSED SOLUTION

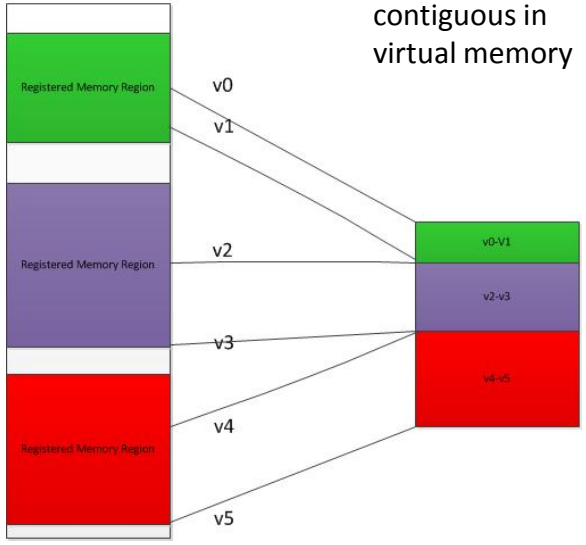
1. **Register the memory layout on the network device**
   - The representation of the layout is stored on the device
   - Either local or remote key is generated
2. **Each transaction applies that layout at a different base address (as a “stencil”)**
   - Pass only a pointer, length and a handle to the layout
3. **For send, the network device uses local DMA read to gather the data according to the layout**
4. **For receive, local DMA write will be used to scatter the data according to the layout**
5. **For RDMA, either Read or Write the data according to the layout directly from remote host memory**

TYPICAL ADDRESSED PATTERNS

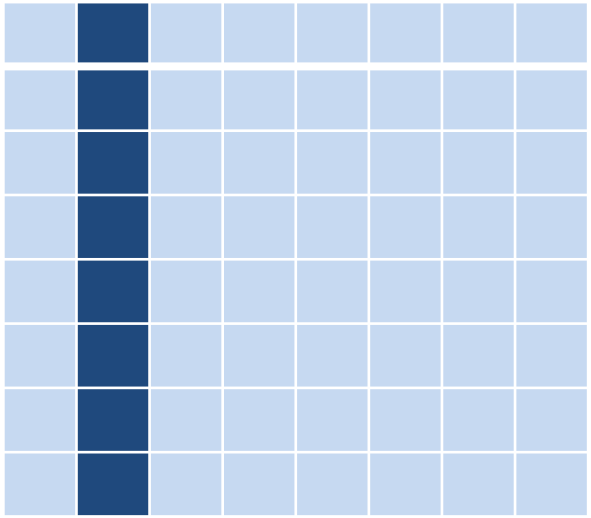
1. Composites

3 memory regions, each referenced by a different memory key

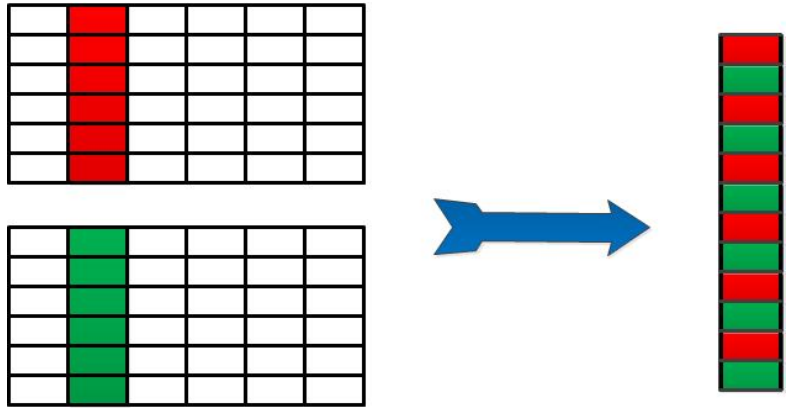
One memory region referenced by one memory key, non-contiguous in virtual memory



2. “Strided” data



3. Interleaved data



API PROPOSAL

EXTEND IBV_ALLOC_MW

include/infiniband/verbs.h:

```
verbs_context_mask {
+   VERBS_CONTEXT_ALLOC_MW  = 1 << 5,
}

struct ibv_alloc_mw_attr {
    unit32_t comp_mask;
    struct ibv_pd *pd;
    enum ibv_mw_type type;
    int max_descriptors;    // Hint to underlying device on ex_mw context size
};

struct verbs_context {
+   struct ib_mw * (*alloc_mw_ex)(struct ibv_alloc_mw_attr *mw_alloc_attr);
}
```

EXTEND IBV_MW OBJECT

include/infiniband/driver.h:

```
enum verbs_mw_mask {
    VERBS_MW_LKEY    = 1 << 0,
    VERBS_MW_DESCRIPTOR_NUM = 1 << 1
};

struct verbs_mw {
    struct ibv_mw    mw;
    uint32_t         comp_mask;
    uint32_t         lkey;
    uint32_t         descriptor_num;
};
```

NEW CAPABILITIES – NON CONTIGUOUS MEMORY

```
struct ibv_mw_caps {
     uint64_t general_caps;
     uint32_t max_mkey_list_sz;
     uint32_t max_mkey_inline_list_sz;
     uint32_t max_mw_recursion_depth;
     uint32_t max_mw_stride_dimenson;
};

enum ibv_mw_general_caps {
     IBV_MW_SUPPORT_COMPOSITE = 1 << 0,
     IBV_MW_SUPPORT_INTERLEAVED = 1 << 1,
     IBV_MW_SUPPORT_INTERLEAVED_REPETITION = 1 << 2,
     IBV_MW_SUPPORT_INTERLEAVED_NONUNIFORM_REPETITION = 1 << 3,
     IBV_MW_SUPPORT_INTERLEAVED_NONUNIFORM_TOTAL_ITEMS = 1 << 4,
 };

struct ibv_device_attr_ex {
+    struct ibv_mw_caps     mw_caps;
};
```

CHANGES TO BIND

```
enum ibv_access_flags {
+    IBV_ACCESS_BIND_MW_NONCONTIG  = (1<<7),
};
```

Instead of:

```
struct ibv_mw_bind_info {
     struct ibv_mr    *mr;
     uint64_t  addr;
     uint64_t  length;
     int       mw_access_flags; /* use ibv_access_flags */
};
```

We propose (binary-compatible, see next slide for layout information):

```
struct ibv_mw_bind_info {
     union {
          struct {
               struct ibv_mr   *mr;
               uint64_t        addr;
               uint64_t        length;
          };
          struct ibv_mw_bind_layout *layout;
     };
     int       mw_access_flags; /* use ibv_access_flags */
};
```

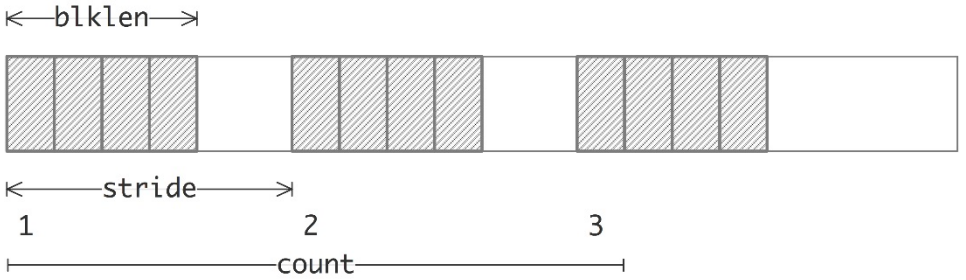
THE LAYOUT DESCRIPTION

```
enum ibv_mw_bind_layout_type {
    IBV_MW_BIND_LAYOUT_TYPE_COMPOSITE = 0,
    IBV_MW_BIND_LAYOUT_TYPE_INTERLEAVED
};

struct ibv_mw_bind_layout {
    uint32_t comp_mask;
    enum ibv_mw_bind_layout_type type;
    uint32_t entry_count;
    struct ibv_mw_bind_info_entry *entries;
};
```

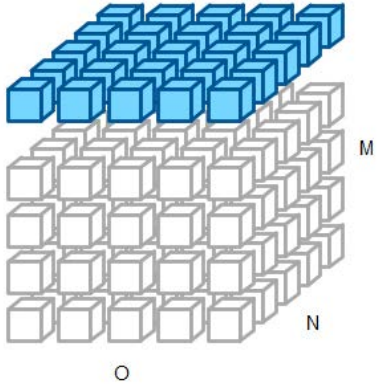
```
enum ibv_mw_entry_type {
    IBV_MW_ENTRY_USE_MR = 0,
    IBV_MW_ENTRY_USE_MW
};

struct ibv_mw_bind_layout_entry {
    uint32_t comp_mask;
    enum ibv_mw_entry_type em_obj_type;
    union mem_obj {
        struct ibv_mr *mr;
        struct ibv_mw *mw;
    };
    uint64_t addr;   // Start of entry Offset
    uint64_t length; // Entry length
    struct {
        uint64_t repeat_count;
        uint32_t dimension_count;
        struct {
            uint64_t stride;
            uint64_t count;
        } *dimension;
    } interleaved;
};
```

LAYOUT DESCRIPTION ILLUSTRATION

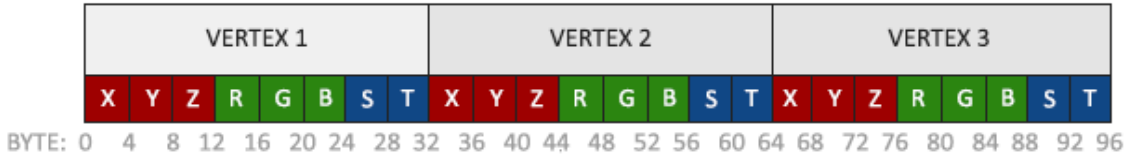
- **Every layout has one or more dimensions**
  - A Single dimension has “stride” and “count”:



  - Two or more dimensions can describe complex patterns:



  - The interleaving proportion is the repeat count of each dimension:




```
struct ibv_mw_bind_layout_entry {
     uint32_t comp_mask;
     enum ibv_mw_entry_type mem_obj_type;
     union mem_obj {
          struct ibv_mr *mr;
          struct ibv_mw *mw;
     };
     uint64_t addr;
     uint64_t length;    //blklen
     struct {
          uint64_t repeat_count;
          uint32_t dimension_count;
          struct {
               uint64_t stride;
               uint64_t count;
          } *dimension;
     } interleaved;
};

enum ibv_mw_bind_layout_type {
     IBV_MW_BIND_LAYOUT_TYPE_COMPOSITE = 0,
     IBV_MW_BIND_LAYOUT_TYPE_INTERLEAVED
};

struct ibv_mw_bind_layout {
     uint32_t comp_mask;
     enum ibv_mw_bind_layout_type type;
     uint32_t entry_count;
     struct ibv_mw_bind_info_entry *entries;
};
```

TYPICAL USAGE FLOW

1. Create a memory window

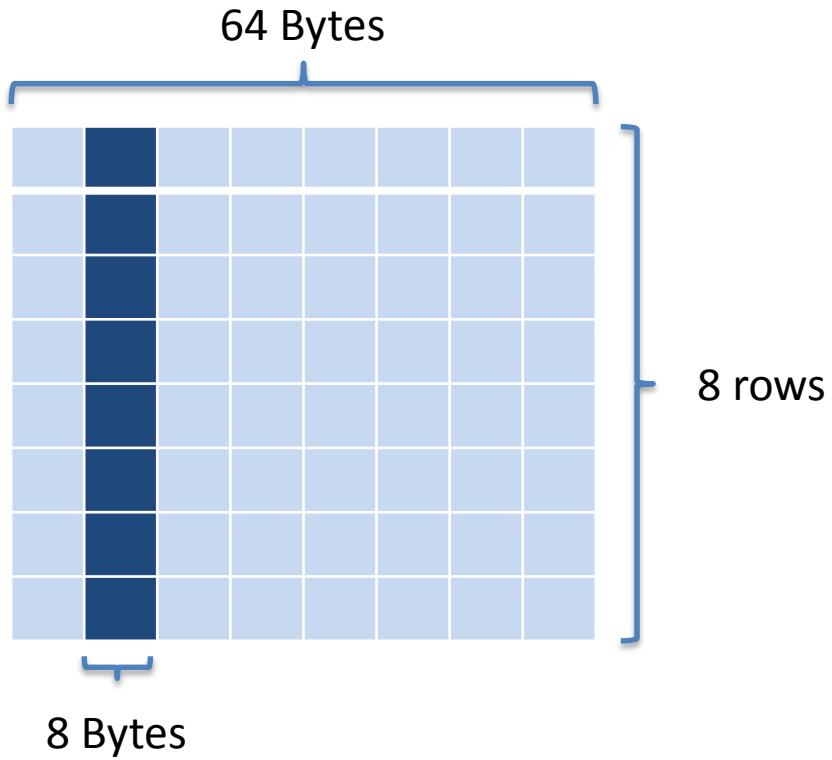
```
Struct ibv_mr *mr = ibv_reg_mr(pd, addr, length, access)
          OR
struct ib_mw *mw = ibv_alloc_mw(pd, type);
          OR
struct ib_mw *mw = ibv_alloc_mw_ex(mw_alloc_attr);
```

2. Create a memory layout

```
bind_layout_entry[0].flags = IBV_MW_ENTRY_USE_MR;
bind_layout_entry[0].mr = mr;
bind_layout_entry[0].addr = addr;
bind_layout_entry[0].length = 8;
bind_layout_entry[0].repeat_count = 1;
bind_layout_entry[0].dimension_count = 1;
bind_layout_entry[0].dimension[0].stride = 64;
bind_layout_entry[0].dimension[0].count = 8;

bind_layout.type = IBV_MW_BIND_LAYOUT_TYPE_INTERLEAVED;
bind_layout.entries = bind_layout_entry;
bind_layout.entry_cnt = 1;
ibv_bind_mw(qp, mw_parent, ibv_mw_bind);
```

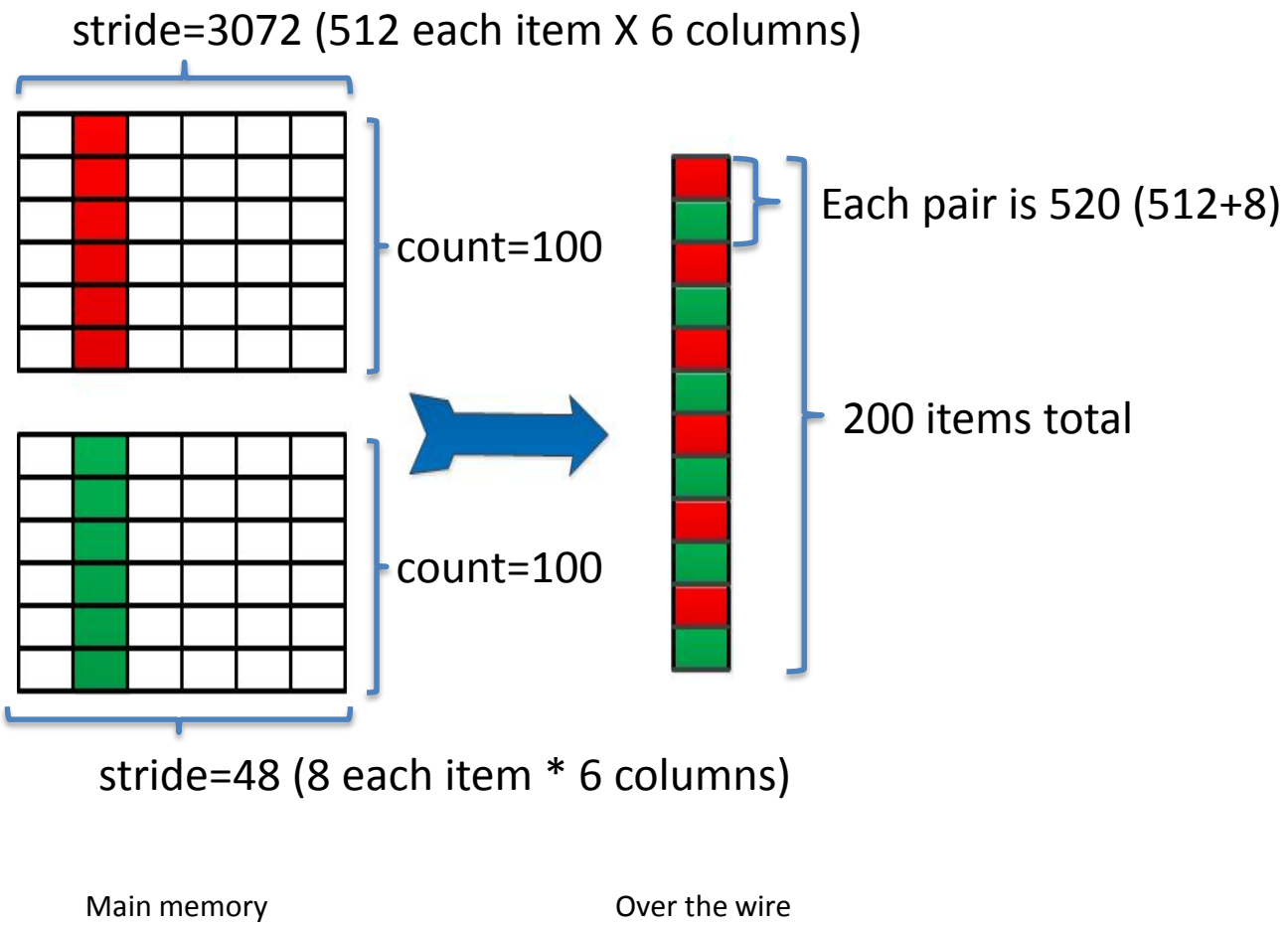
3. Send a packet

```
wr.sg_list[0].addr = ptr;
wr.sg_list[0].length = total_length;
wr.sg_list[0].lkey = mw_parent->lkey;
ibv_post_send(qp, wr, &bad_wr);
```



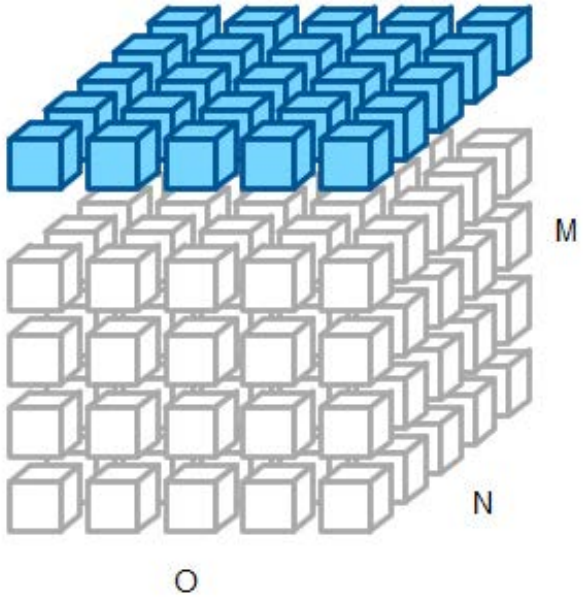
INTERLEAVED PATTERN EXAMPLE

```
bind_layout_entry[0].mw_entry_flags = IBV_MW_ENTRY_USE_MR;
bind_layout_entry[0].mr = mr1;
bind_layout_entry[0].addr = addr1;
bind_layout_entry[0].length = 512;
bind_layout_entry[0].repeat_count = 1;
bind_layout_entry[0].dimension_count = 1;
bind_layout_entry[0].dimension[0].stride = 3072;
bind_layout_entry[0].dimension[0].count = 100;

bind_layout_entry[1].mw_entry_flags = IBV_MW_ENTRY_USE_MR;
bind_layout_entry[1].mr = mr2;
bind_layout_entry[1].addr = addr2;
bind_layout_entry[1].length = 8;
bind_layout_entry[1].repeat_count = 1;
bind_layout_entry[1].dimension_count = 1;
bind_layout_entry[1].dimension[0].stride = 48;
bind_layout_entry[1].dimension[0].count = 100;

bind_layout.type = IBV_MW_BIND_LAYOUT_TYPE_INTERLEAVED;
bind_layout.entries = bind_layout_entry;
bind_layout.entry_count = 2;
```



LAYOUT DESCRIPTION – MULTI DIMENTION

```
bind_layout_entry[0].mw_entry_flags = IBV_MW_ENTRY_USE_MR;
bind_layout_entry[0].mr = mr1;
bind_layout_entry[0].addr = addr1;
bind_layout_entry[0].length = 3;
bind_layout_entry[0].repeat_count = 1;
bind_layout_entry[0].dimension_count = 2;
bind_layout_entry[0].dimension[0].stride = 3 * N;
bind_layout_entry[0].dimension[0].count = O;
bind_layout_entry[0].dimension[1].stride = 3 * M * O;
bind_layout_entry[0].dimension[1].count = N;
```



NON-UNIFORM REPETITIONS EXAMPLE

X	
Y	
Z	
X	
Y	
Z	

- Length = 1
- repeat_count = 3
- count = 18 (6 cycles X 3 items)
- stride = 2

R			
G			
B			
R			
G			
B			

- Length = 1
- repeat_count = 3
- count = 18 (6 cycles X 3 items)
- stride = 4

S				
T				
S				
T				
S				
T				

- Length = 1
- repeat_count = 2
- count = 12
  (6 cycles X 2 items)
- stride = 5

There are 6 cycles in this example

Each cycle takes 3:3:2, according to repeat_counts of each element (since all lengths are 1, cycle is 8 bytes)

SUMMARY

- **Devices to become aware of application objects memory layout**
- **Such scheme allows the layout re-use as a stencil for all objects with same type**
- **Non contiguous virtual address space can become contiguous for the device**
- **Main benefits**
  - Zero copy – no packing/unpacking
  - Avoid messing with long scatter-gather lists
- **API wise, good fit as an extension of the Verbs Memory Window object**

OPENFABRICS ALLIANCE

13th ANNUAL WORKSHOP 2017

THANK YOU

Tzahi Oved, Alex Margolin

Mellanox Technologies