UCD: A High-performance Datatype Engine for Noncontiguous Data

Pavan Balaji, Argonne National Laboratory

Akshay Venkatesh, NVIDIA

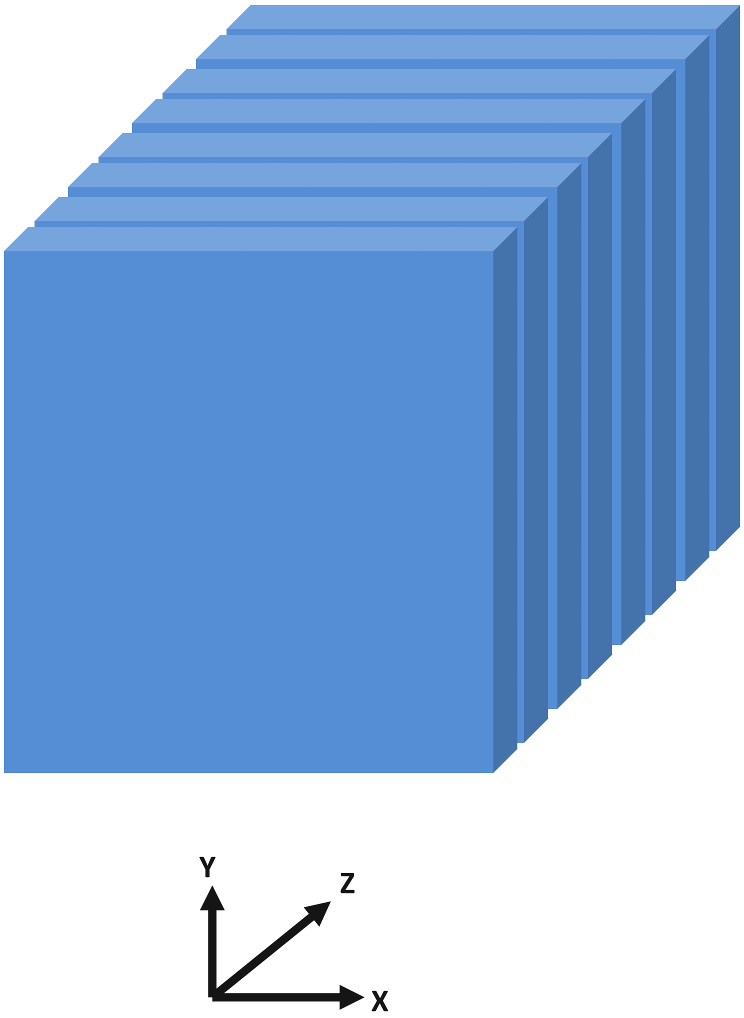
Artem Polyakov, NVIDIA

Jim Dinan, NVIDIA

Manjunath Gorentla Venkata, NVIDIA

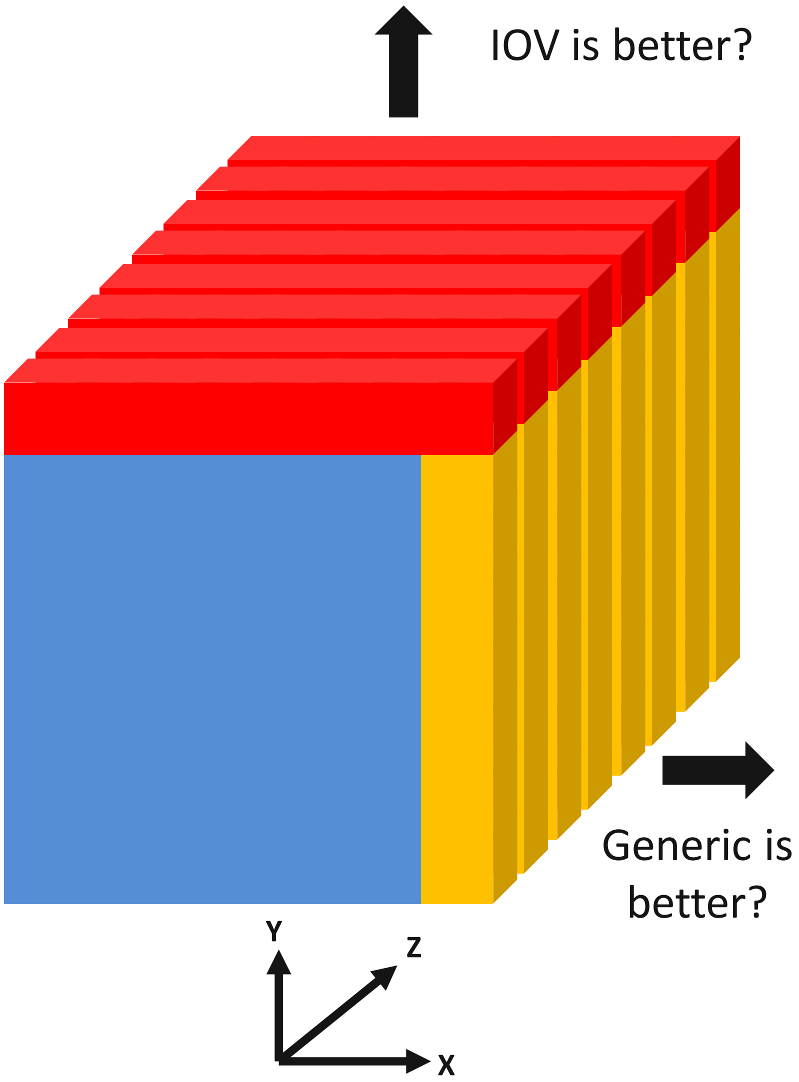
Noncontiguous Data Movement

- Important form of communication for scientific computing (MPI) and modern DL systems
  - Users can create static, but not contiguous, data layouts
  - Vector-of-struct-of-indexed-of-hvector-of-contig-of-doubles
- UCX provides three (or four) modes of communication today
  - Contig: for contiguous data
  - IOV: allows users to describe data as a series of contiguous chunks
  - Generic: no information passed to UCX about the data layout (user has to provide pack/unpack functionality)
  - Strided: not implemented



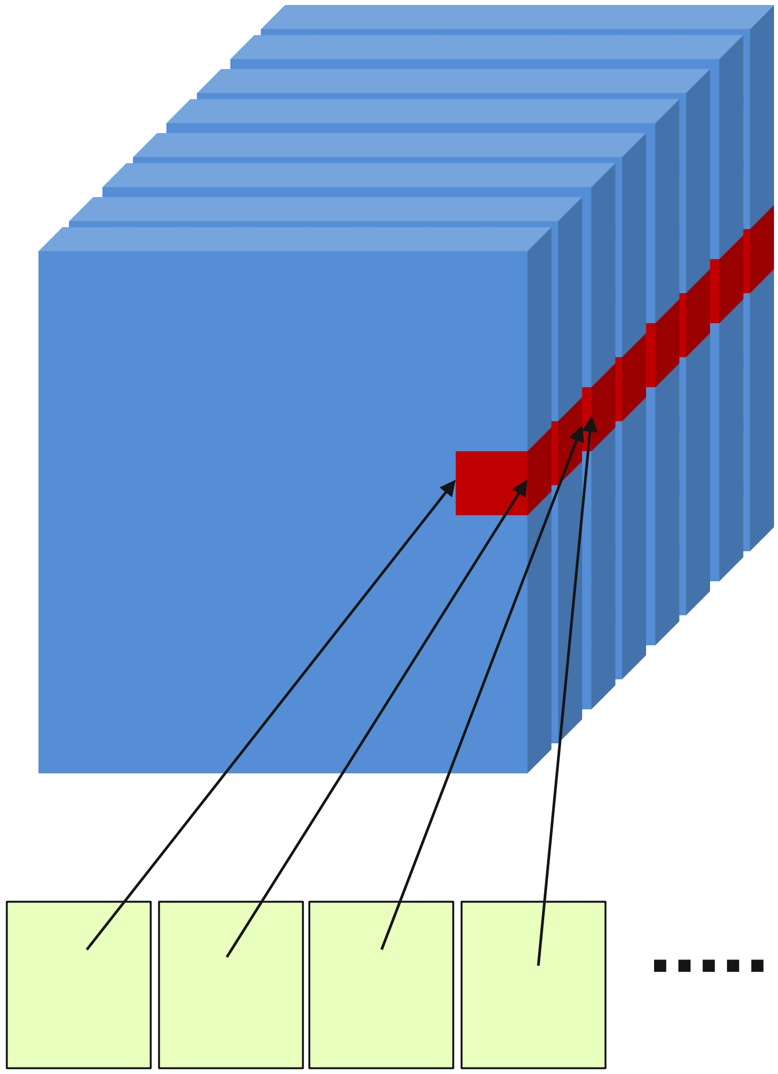
What are we missing?

- Impossible to provide some functionality such as reduction (need to know integers/floats, and not just bytes)
  - Important for collectives as well as for RMA accumulates
- Inefficient to utilize hardware features such as InfiniBand UMR or to decide between ”generic” (pack/unpack) vs. IOV
- Inconvenient to move noncontiguous data from non-CPU memory



Shortcomings of IOV-based datatype processing

- Each IOV element contains:
  - a pointer to the start of a contiguous segment
  - the length of the contiguous segment
- In common patterns, each contiguous segment is small (e.g., one double)
- IOV creation is typically more expensive than packing the data
  - Plus, the overhead of multiple small communication operations



UCD: Noncontiguous Datatype Engine

- UCD provides almost all of the MPI datatype functionality + additions needed for practical usage within other libraries
- Four sets of APIs
  - Predefined datatypes and datatype creation
    - All MPI basic datatypes (including pair types) are supported
    - All derived datatype creations (except darray) are supported
  - Pack/unpack/accumulate routines
    - With extensions, so one can perform partial packing (for pipelining)
  - IOV routines: convert derived datatypes to an IOV
  - Flatten/unflatten routines
    - Convert the derived datatype into a portable format
    - Can be portably sent to other processes (e.g., when RMA is implemented with active messages, or for shared memory)
- Internally utilizes “yaksa” to support both CPU and GPU memory
  - Working with NVIDIA (integrated), Intel (integrated) and AMD (in progress)

Datatype creation routines

```
UCD_TYPE__INT, UCD_TYPE__FLOAT, UCD_TYPE__DOUBLE, ..
```

```
int ucd_create_vector(int count, int blocklength, int stride,
                      ucd_type_t oldtype, ucd_info_t info, ucd_type_t *newtype);
```

- Very similar to MPI datatype creation routines
- Hierarchical construction, so data layouts can be arbitrarily complex
- Basically equivalent to pulling out the datatypes part of MPI outside the MPI standard, so it’s usable within other environments too

Pack/Unpack routines

```
int ucd_ipack(const void *inbuf,
    uintptr_t incount, ucd_type_t type, uintptr_t inoffset,
    void *outbuf, uintptr_t max_pack_bytes, uintptr_t *actual_pack_bytes,
    ucd_info_t info, ucd_op_t op, ucd_request_t *request);
```

- Extended versions of MPI_Pack/unpack routines
  - Allow for offsets and partial packing (allows one to pipeline packing into temporary buffers)
    - E.g., pack the first 64KB into a temporary buffer, send it, pack the next 64KB into a temporary buffer, …
  - Allow for nonblocking packing
    - Useful for GPU resident buffers, where a DMA request or a kernel launch might need to complete for the pack
  - Allow for predefined ops on the packed data (SUM, BOR, LOR, …)

IOV routines

```
int ucd_iov(const void *buf,
        uintptr_t count, ucd_type_t type, uintptr_t iov_offset,
        struct iovec *iov, size_t max_iov_len, uintptr_t *actual_iov_len);
```

- Similar to packing, allows for offsets and partial conversion to IOV segments: useful for pipelining

Intended Usage

```
ucd_iov_len(count, type, &iov_len);
ucd_get_size(type, &size);
if (count * size / iov_len > THRESHOLD) {
    ucd_iov(..., iov, ...);
    for (int i = 0; i < iov_len; i++) internal_isend(...);
} else {
    ucd_ipack(..., &outbuf, ...);
    ucd_wait(request);
    internal_isend(...);
}
```

Flatten/unflatten routines

```
int ucd_flatten(ucd_type_t type, void *flattened_type);
int ucd_unflatten(ucd_type_t type, const void *flattened_type);
```

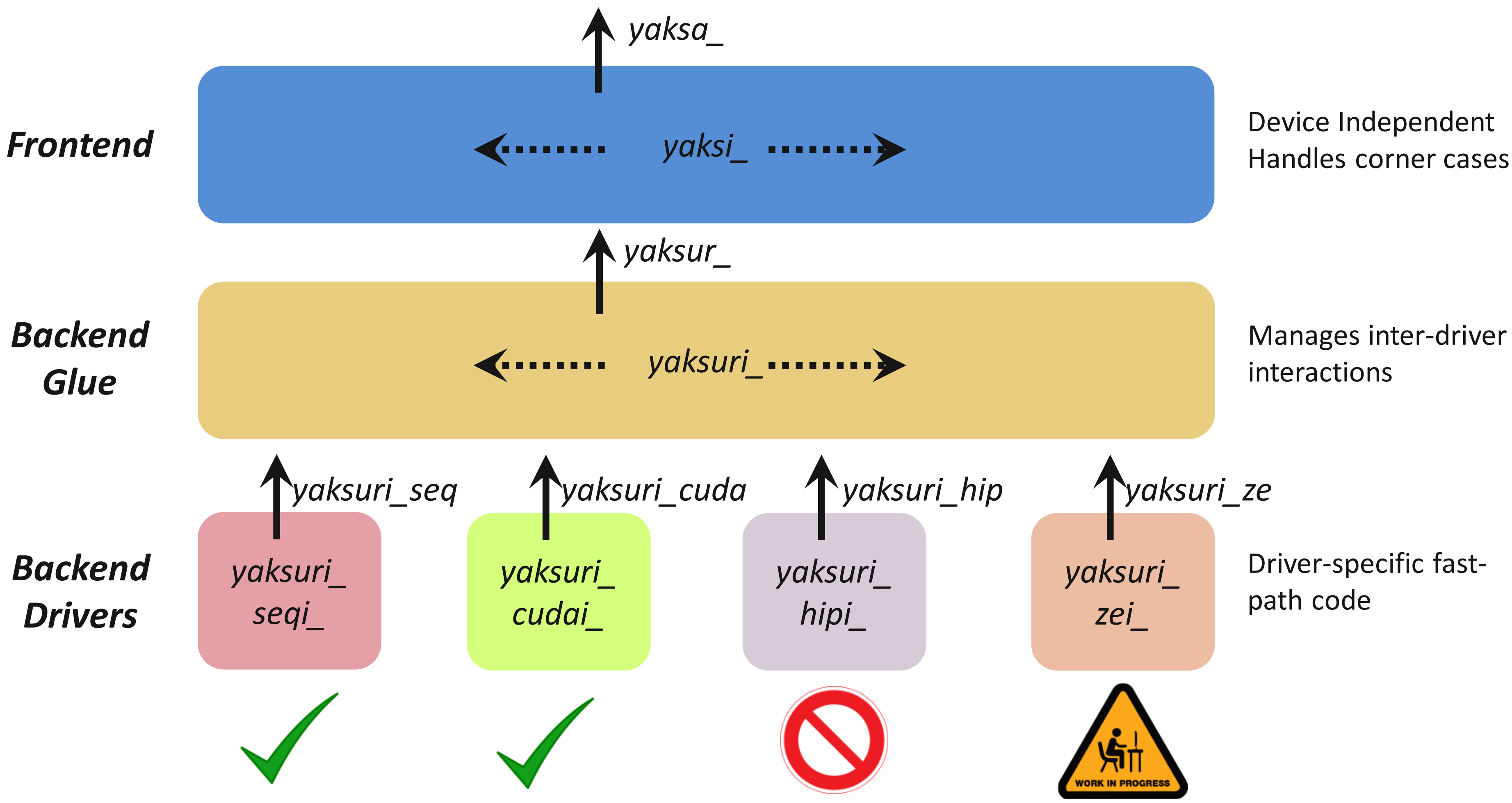
- Datatype flattening converts a UCD type into a portable format that can be transferred across virtual address space boundaries (e.g., between MPI processes)
- Particularly useful for one-sided communication
  - Origin process provides both origin and target datatype
  - If the communication library decides to use active messages to implement it, it would need to send the target datatype to the target process
- Can also be useful for some persistent collective operations

General comments about the UCD API

- All routines are local: everything will complete “immediately” (i.e., in a finite amount of time)
- Routines can be separated into two classes:
  - Data touching: pack/unpack are the only two routines that touch the data and have nonblocking variants to allow for pipelining
    - It would be semantically correct if we waited for completion in the ipack/iunpack routines, but would hurt performance
  - Non-data-touching: everything else
    - No nonblocking variants for these routines

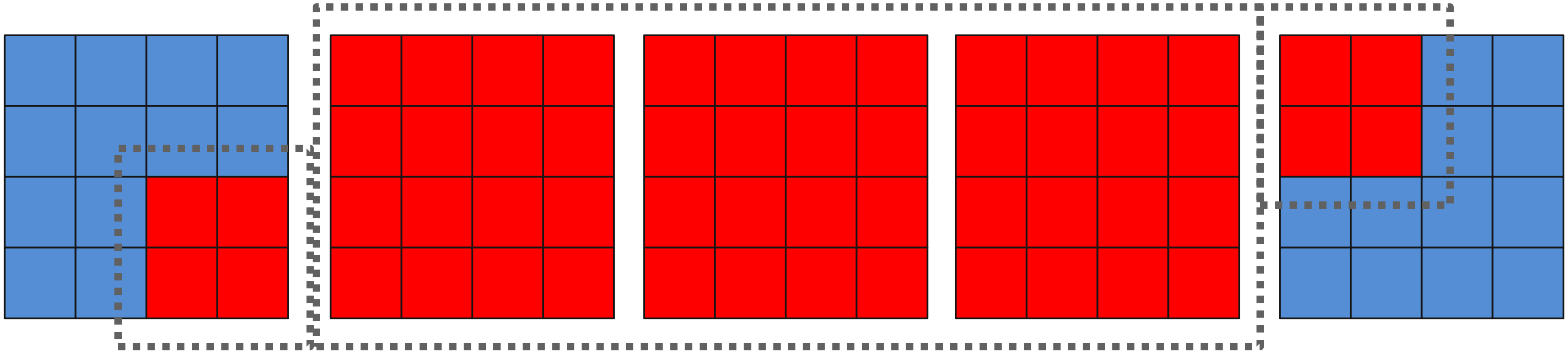
Yaksa:
UCD’s internal data management engine

Yaksa Software Architecture



No parallel backend for CPUs: easy to write a pthreads or OpenMP wrapper outside of Yaksa for parallel packing

Backend code generation (1/2)



- The frontend manages quirky inputs such as nonzero offsets or partial packing/unpacking
  - Converts into a series of smaller structured pack/unpack routines
  - Easier to generate code for structured pack/unpack blocks
- Functions generated for up to four levels nesting (three, if you don't include the basic datatype)
  - All derived datatype combinations, except struct
  - Each datatype has function pointers pointing to the specific pack/unpack functions that would work for that type

Backend code generation (2/2)

Up to 3-level datatypes (suitable for up to 4D data structures)

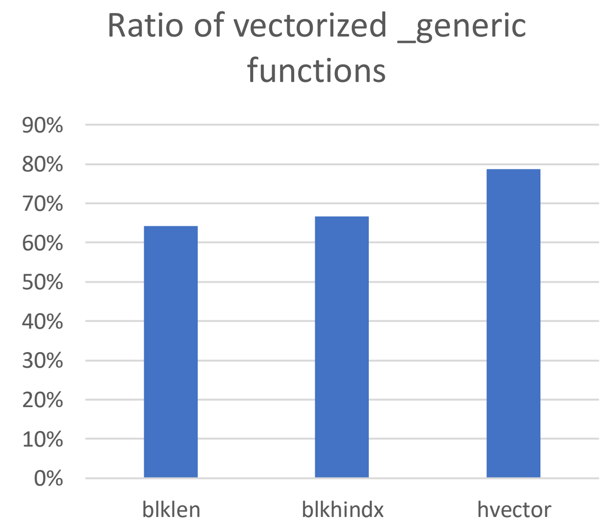
Can use any attribute available at type creation time (static block lengths, basic datatypes)

Structured access with restrict pointers makes it easier for compilers to vectorize and prefetch

```
int yaksuri_seqi_pack_hvector_hvector_blklen_1_double(const void *inbuf, void *outbuf, uintptr_t count, yaksi_type_s * type)
{
    int rc = YAKSA_SUCCESS;
    const char *restrict sbuf = (const char *) inbuf;
    char *restrict dbuf = (char *) outbuf;
    uintptr_t extent ATTRIBUTE((unused)) = type->extent;

    int count1 = type->u.hvector.count;
    int blocklength1 ATTRIBUTE((unused)) = type->u.hvector.blocklength;
    intptr_t stride1 = type->u.hvector.stride;
    uintptr_t extent1 ATTRIBUTE((unused)) = type->extent;

    int count2 = type->u.hvector.child->u.hvector.count;
    int blocklength2 ATTRIBUTE((unused)) = type->u.hvector.child->u.hvector.blocklength;
    intptr_t stride2 = type->u.hvector.child->u.hvector.stride;
    uintptr_t extent2 ATTRIBUTE((unused)) = type->u.hvector.child->extent;

    uintptr_t idx = 0;
    for (int i = 0; i < count; i++) {
        for (int j1 = 0; j1 < count1; j1++) {
            for (int k1 = 0; k1 < blocklength1; k1++) {
                for (int j2 = 0; j2 < count2; j2++) {
                    for (int k2 = 0; k2 < 1; k2++) {
                        *((double *) (void *) (dbuf + idx)) =
                            *((const double *) (const void *) (sbuf + i * extent + j1 * stride1 +
                                                               k1 * extent2 + j2 * stride2 + k2 * sizeof(double)));
                        idx += sizeof(double);
                    }
                }
            }
        }
    }

    return rc;
}
```

Yaksa Vectorization

- Data copy in all the Yaksa kernels can be done in parallel
- Focusing on innermost loops of _generic functions:
  - Clang 10.0.0, GCC 9.2.0, GCC8.2.0, and GCC5.5.0 yield the same results (60 - 80%)
    - We believe all functions should be vectorized
- We are exploring the reason of failures and how to promote vectorization
  - Calculating induction variables outside the loop seems effective, but it needs more investigation



- Note:
  - Other innermost kernels (contig/hindexed/resized) are not vectorized.
  - Vectorization results of specialized innermost loops that have fixed loop ranges vary (because of a cost model and efficiency of SLP-vectorizer)

GPU backends (CUDA and ZE)

- Kernel-offload based packing
- Two sets of temporary buffers maintained on each device
  - One for staging data (in case the pack is between device <-> host)
  - One for staging datatype metadata
- Staging data: Simple pool of buffers for packing/unpacking
  - If the pool is empty, the operation is queued up in software (progress poke needed)
- Staging datatype metadata:
  - Managed memory, allowing for frequently used datatypes to be cached on the GPU
  - Allows the runtime to evict these buffers if the application needs it

GPU backend code generation: CUDA example

```
__global__ void yaksuri_cudai_kernel_pack_hvector_hvector_hvector_double(const double *__restrict__ sbuf, double *__r\
estrict__ dbuf, int count, const yaksuri_cudai_md_s *__restrict__ md)
{
    uintptr_t extent = md->extent / sizeof(double);
    uintptr_t idx = blockIdx.x * blockDim.x + threadIdx.x;
    uintptr_t res = idx;
    uintptr_t inner_elements = md->num_elements;

    if (idx >= (count * inner_elements))
        return;

    uintptr_t x0 = res / inner_elements;
    res %= inner_elements;
    inner_elements /= md->u.hvector.count;

    uintptr_t x1 = res / inner_elements;
    res %= inner_elements;
    inner_elements /= md->u.hvector.blocklength;
    uintptr_t x2 = res / inner_elements;
    res %= inner_elements;
    inner_elements /= md->u.hvector.child->u.hvector.count;

    uintptr_t x3 = res / inner_elements;
    res %= inner_elements;
    inner_elements /= md->u.hvector.child->u.hvector.blocklength;
    uintptr_t x4 = res / inner_elements;
    res %= inner_elements;
    inner_elements /= md->u.hvector.child->u.hvector.child->u.hvector.count;

    uintptr_t x5 = res / inner_elements;
    res %= inner_elements;
    inner_elements /= md->u.hvector.child->u.hvector.child->u.hvector.blocklength;
    uintptr_t x6 = res;

    intptr_t stride1 = md->u.hvector.stride / sizeof(double);
    intptr_t stride2 = md->u.hvector.child->u.hvector.stride / sizeof(double);
    uintptr_t extent2 = md->u.hvector.child->extent / sizeof(double);
    intptr_t stride3 = md->u.hvector.child->u.hvector.child->u.hvector.stride / sizeof(double);
    uintptr_t extent3 = md->u.hvector.child->u.hvector.child->extent / sizeof(double);
    dbuf[idx] = sbuf[x0 * extent + x1 * stride1 + x2 * extent2 + x3 * stride2 + x4 * extent3 + x5 * stride3 + x6];
}
```

```
int yaksuri_cudai_pack_hvector_hve
* type, yaksi_request_s **request)
{
    int rc = YAKSA_SUCCESS;
    cudaError_t cerr;

    rc = yaksuri_cudai_md_alloc(type);
    /* nvcc does not seem to like gotos */
    /* YAKSU_ERR_CHECK(rc, fn_fail); */

    yaksuri_type_s *backend = (yaksuri_typ
    yaksuri_cudai_type_s *cuda = (yaksuri_
    yaksuri_cudai_md_s *md = cuda->md;

    int n_threads = YAKSURI_CUDAI_THREAD_B
    int n_blocks = count * cuda->num_eleme
    n_blocks += !!(count * cuda->num_eleme

    void *args[4] = { &inbuf, &outbuf, &co

    cerr = cudaLaunchKernel((const void *)
ads, args, 0, yaksuri_cudai_global.stream)
    YAKSURI_CUDAI_CUDA_ERR_CHECK(cerr);

    cerr = cudaStreamSynchronize(yaksuri_cudai_global.stream);
    YAKSURI_CUDAI_CUDA_ERR_CHECK(cerr);

    return rc;
}
```

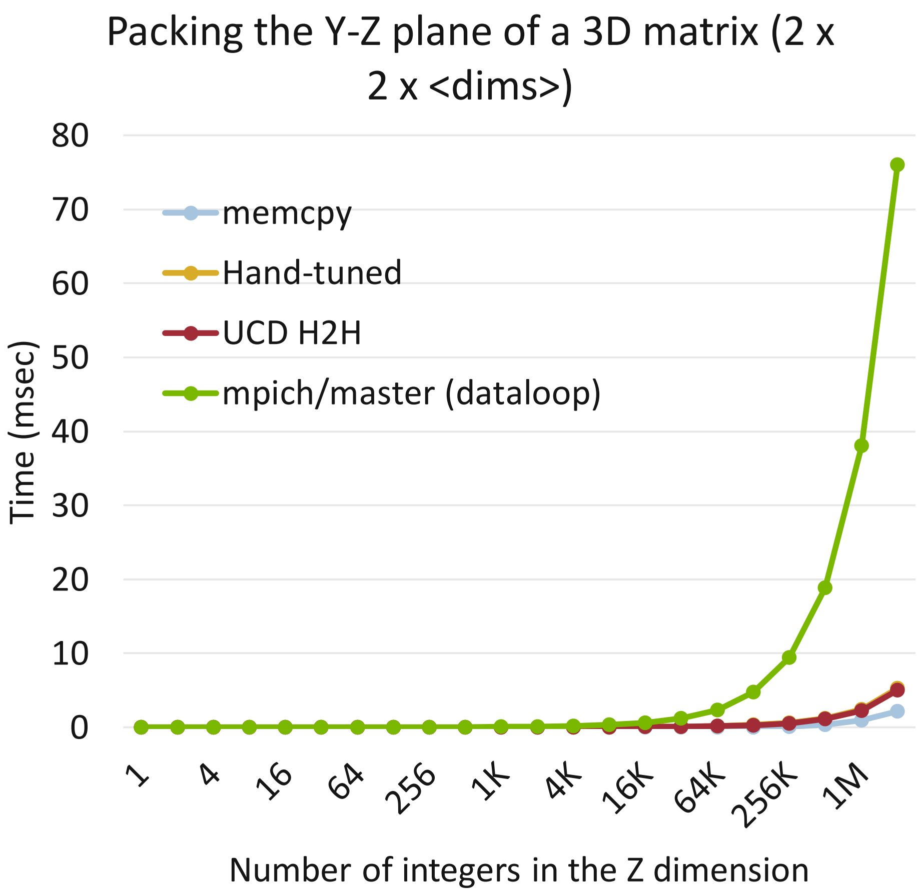
Separate host-side and device-side code generation, and explicit management of datatype metadata on the GPU is needed

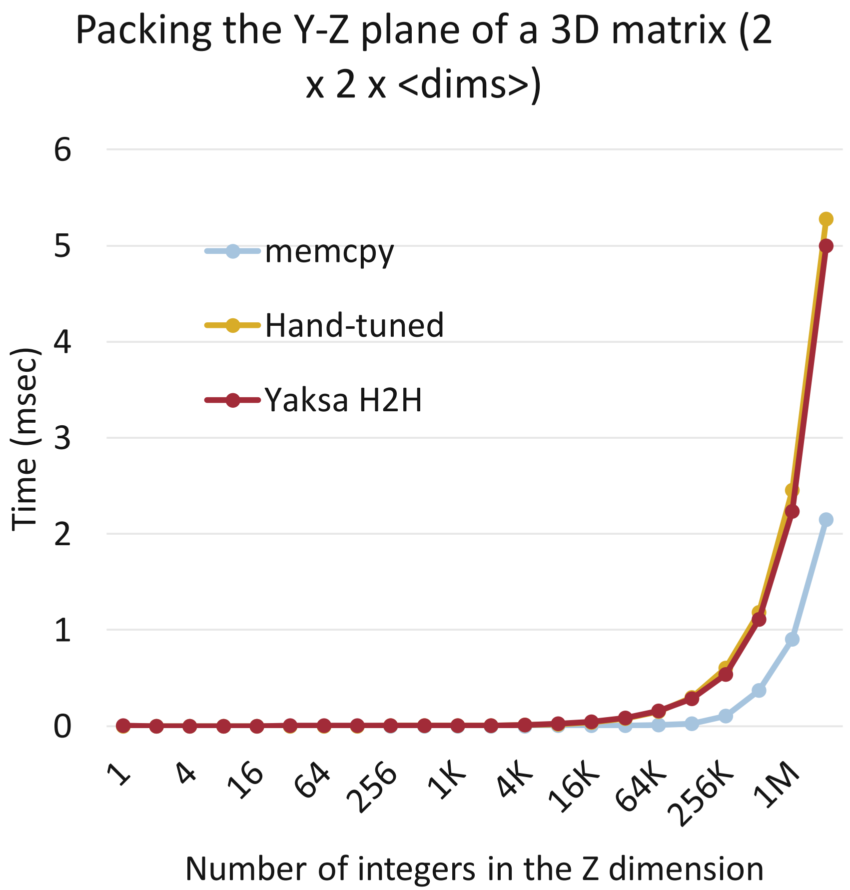
Backend Glue

- The backend glue layer handles multi-driver or device-to-device interactions for a single driver
  - Basically anything that uses temporary buffers
  - Uses a progress engine to keep the use of temporary buffers within a threshold
  - Converts all zero-copy calls to PUT-based, instead of GET-based
- Creates somewhat complex graph structures to help with temporary buffer management
  - E.g., unpack/compute from device 1 to device 2 when there is no D2D IPC available can have numerous steps
  - (1) Pack from device 1 (source buf) to device 1 (tmpbuf); (2) DMA from device 1 (tmpbuf) to host; (3) DMA from host to device 2 (tmpbuf); (4) accumulate from device 2 (tmpbuf) to device 2 (dest buf)

Performance Results

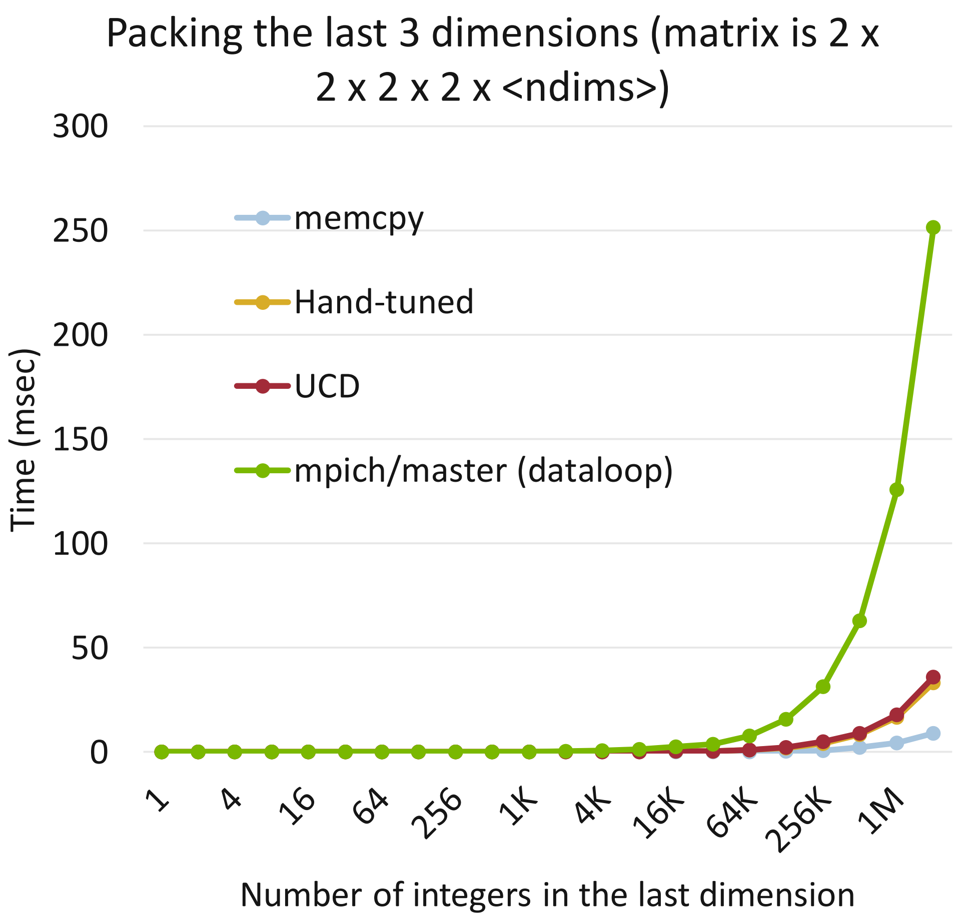
Data packing Y-Z plane of a 3D matrix

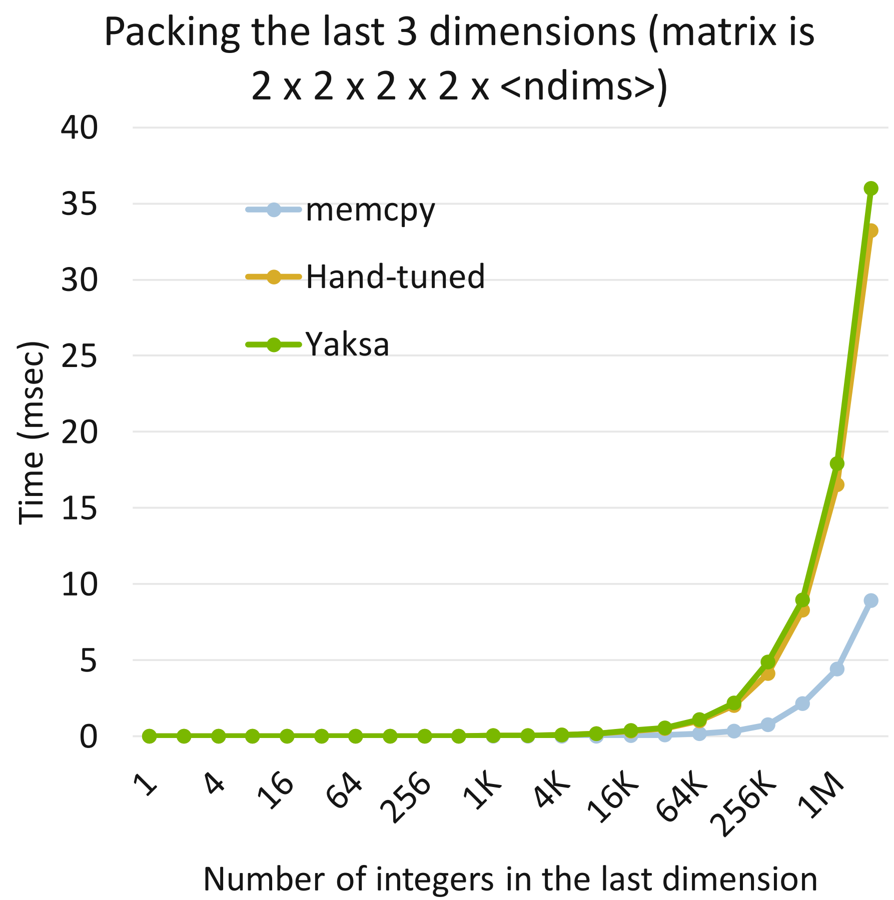




Extreme case: only the outermost dimension is large

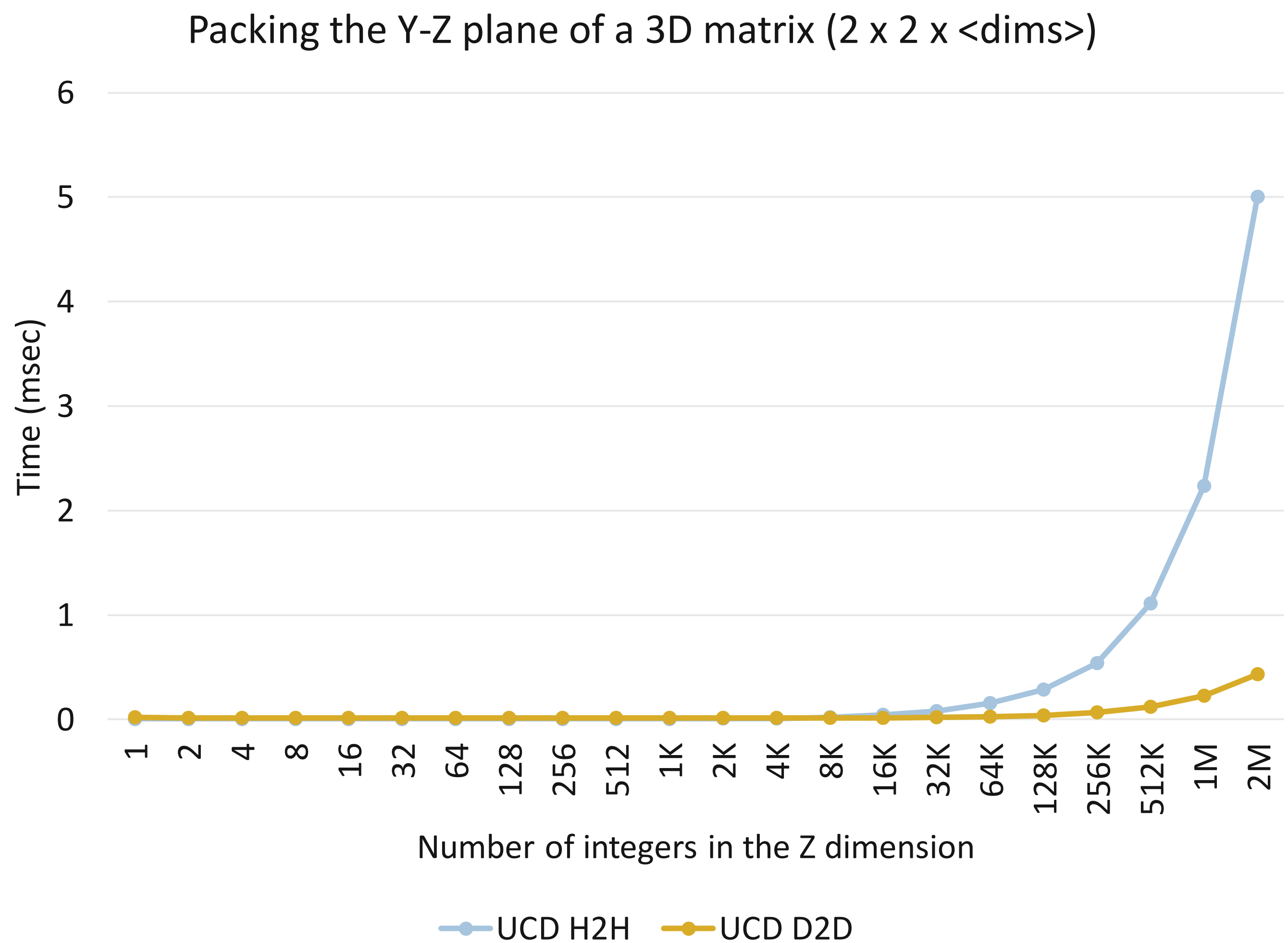
Data packing: Last 4 dimensions of a 5D matrix





Yaksa uses code generation for up to 4D matrices. Beyond that, at least one level has to be packed using multiple function calls. The above graphs show the worst-case scenario.

H2H vs. D2D (CUDA)



Extreme case: only the outermost dimension is large

Code Status

UCD code

- Available as a standalone library (not integrated into UCX)
  - https://github.com/pavanbalaji/ucd
  - (sorry, I just got our legal go ahead yesterday)

```
int ucd_ipack(const void *inbuf, uintptr_t incount, ucd_type_t type, uintptr_t inoffset,
              void *outbuf, uintptr_t max_pack_bytes, uintptr_t * actual_pack_bytes,
              ucd_info_t info, ucd_request_t * request)
{
    int rc = YAKSA_SUCCESS;

    rc = ucdi_init();
    UCDI_ERR_POP(rc, fn_fail);

    if (ucdi_yaksa_is_available) {
        rc = ucdi_yaksa_fns.ipack(inbuf, incount, ucdi_ucd_to_yaksa_type(type), inoffset, outbuf,
                                  max_pack_bytes, actual_pack_bytes, info, request);
        UCDI_ERR_POP(rc, fn_fail);
    } else {
        rc = UCD_ERR__NOT_SUPPORTED;
    }

  fn_exit:
    return rc;
  fn_fail:
    goto fn_exit;
}
```

Next steps

- UCD is currently standalone
  - Not very useful as a standalone library (very similar to yaksa)
  - Needs some interaction with UCX/UCC to be useful
- UCD needs to internally keep track of some datatype information:
  - Some “special datatypes” (e.g., vector-of-vector-of-double)
  - Base types (e.g., double, int)
- It needs to use some internal (not user visible) functionality to expose this to UCP/UCC
- UCP/UCC can use hardware features (such as UMR) to accelerate them

Other things the WG is thinking about

- UCD (actually yaksa) internally creates its own set of streams and temporary buffers for each GPU
  - Perhaps it is possible for UCP/UCT to pass the temporary buffers that it already has to UCD
- Runtime generation of pack/unpack kernels
  - Avoids static generation of common kernels
  - Somewhat easy to do with Intel GPUs, but might be harder for NVIDIA or AMD GPUs

UCD: A High-performance Datatype Engine for Noncontiguous Data

Pavan Balaji, Argonne National Laboratory

Akshay Venkatesh, NVIDIA

Artem Polyakov, NVIDIA

Jim Dinan, NVIDIA

Manjunath Gorentla Venkata, NVIDIA