Basic Usage Guide

We can now walk through how to set up and run a basic program with cuDecomp. The code snippets in this section are taken from the basic usage example [link]. This example assumes we are using 4 GPUs.

Starting up cuDecomp

First, initialize MPI and assign CUDA devices. In this example, we assign CUDA devices based on the local rank.

C++ | Fortran

```
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, nranks, ierr)

call MPI_Comm_split_Type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, local_comm, ierr)
call MPI_Comm_rank(local_comm, local_rank, ierr)
ierr = cudaSetDevice(local_rank)
```

Next, we can initialize cuDecomp with cudecompInit using the MPI global communicator and obtain a handle.

C++ | Fortran

```
type(cudecompHandle) :: handle
...
istat = cudecompInit(handle, MPI_COMM_WORLD)
call CHECK_CUDECOMP_EXIT(istat)
```

Creating a grid descriptor

Next, we need to create a grid descriptor. To do this, we first need to create and populate a grid descriptor configuration structure, which provides basic information to the library required to set up the grid descriptor.

We create an uninitialized configuration struct and initialize it to defaults using cudecompGridDescConfigSetDefaults. Initializing to default values is required to ensure no entries are left uninitialized.

C++ | Fortran

```
type(cudecompGridDescConfig) :: config
...
istat = cudecompGridDescConfigSetDefaults(config)
call CHECK_CUDECOMP_EXIT(istat)
```

First, we can set the `pdims` (process grid) entries in the configuration struct. `pdims[0]` corresponds to P_{rows} and `pdims[1]` corresponds to P_{cols}. In this example, we use a 2×2 process grid.

C++ | Fortran

```
config%pdims = [2, 2] ! [P_rows, P_cols]
```

Next, we set the `gdims` (global grid) entries in the configuration struct. These values correspond to the X, Y, and Z dimensions of the global grid. In this example, we use a global grid with dimensions $64 \times 64 \times 64$.

C++ | Fortran

```
config%gdims = [64, 64, 64] ! [X, Y, Z]
```

For additional flexibility, the configuration structure contains an optional entry `gdims_dist` that indicates to the library that the global domain of dimension `gdims` should be distributed across processes with elements divided among processes as though the global domain was of dimension `gdims_dist`. This can be useful when dealing with padded domain dimensions. The entries in `gdims_dist` must be less than or equal to the entries in `gdims` and any extra elements are associated with the last rank in any row or column communicator.

Next, we set the desired communication backends for transpose (`transpose_comm_backend`) and/or halo communication (`halo_comm_backend`). See documentation of cudecompTranposeCommBackend_t and cudecompHaloCommBackend_t for the available communication backends options.

C++ | Fortran

```
config%transpose_comm_backend = CUDECOMP_TRANSPOSE_COMM_MPI_P2P
config%halo_comm_backend = CUDECOMP_HALO_COMM_MPI
```

We can next set the values of `transpose_axis_contiguous`, which are boolean flags indicating to the library the memory layout of the pencil buffers to use, by axis. For each axis, cuDecomp supports two possible memory layouts depending on the setting of these flags.

`transpose_axis_contiguous`	X-pencil	Y-pencil	Z-pencil
`true`	$[X, Y, Z]$	$[Y, Z, X]$	$[Z, X, Y]$
`false`	$[X, Y, Z]$	$[X, Y, Z]$	$[X, Y, Z]$

These memory layouts are listed in column-major order. When this flag is false for an axis, the memory layout of the pencil buffers remains in the original memory layout of the global grid, $[X, Y, Z]$. Alternatively, when this flag is true for an axis, the memory layout is permuted (cyclic permutation) so that the data is contiguous along the pencil axis (e.g., for the Z-pencil, the memory is ordered so that data along the Z axis is contiguous). This permuted memory layout can be desirable in situations where the computational performance of your code may improve with contiguous access of data along the pencil axis (e.g. to avoid strides between signal elements in an FFT). In this example, we set this flag to true for all directions.

C++ | Fortran

```
config%transpose_axis_contiguous = [.true., .true., .true.]
```

With the grid descriptor configuration structure created and populated, we can now create the grid descriptor. The last argument in cudecompGridDescCreate is for an optional structure to set autotuning options. See Autotuning for a detailed overview of this feature. In this example, we will not autotune and pass a `nullptr` for this argument in C/C++, or equivalently, leave it unspecified in Fortran.

C++ | Fortran

```
type(cudecompGridDesc) :: grid_desc
...
istat = cudecompGridDescCreate(handle, grid_desc, config)
call CHECK_CUDECOMP_EXIT(istat)
```

Allocate pencil memory

Once the grid descriptor is created, we can now query information about the decomposition and allocate device memory to use for the pencil data.

First, we can query basic information (i.e. metadata) about the pencil configurations that the library assigned to this process using the cudecompGetPencilInfo function. This function returns a pencil info structure (cudecompPencilInfo_t) that contains the shape, global lower and upper index bounds (`lo` and `hi`), size of the pencil, and an `order` array to indicate the memory layout that will be used (to handle permuted, *axis-contiguous* layouts). Additionally, there is a `halo_extents` data member that indicates the depth of halos for the pencil, by axis, if the argument was provided to this function. This data member is a copy of the argument provided to the function and is stored for convenience.

It should be noted that these metadata structures are provided solely for users to interpret and access data from the data buffers used as input/output arguments to the different cuDecomp communication functions. Outside of autotuning, the library does not allocate memory for pencil buffers, nor uses these pencil information structures as input arguments.

In this example, we apply halo elements to the X-pencils only. For the other pencils, we instead pass a `nullptr` for the `halo_extents` argument, which is equivalent to setting `halo_extents = [0, 0, 0]` in C/C++. For Fortran, `halo_extents` is optional and defaults to no halo regions.

C++ | Fortran

```
type(cudecompPencilInfo) :: pinfo_x, pinfo_y, pinfo_z
...

! Get X-pencil information (with halo elements)
istat = cudecompGetPencilInfo(handle, grid_desc, pinfo_x, 1, [1, 1, 1])
call CHECK_CUDECOMP_EXIT(istat)

! Get Y-pencil information
istat = cudecompGetPencilInfo(handle, grid_desc, pinfo_y, 2)
call CHECK_CUDECOMP_EXIT(istat)

! Get Z-pencil information
istat = cudecompGetPencilInfo(handle, grid_desc, pinfo_z, 3)
call CHECK_CUDECOMP_EXIT(istat)
```

With the information from the pencil info structures, we can now allocate device memory to use with cuDecomp. In this example, we allocate a single device buffer `data_d` that is large enough to hold the largest pencil assigned to this process, across the three axes. We also allocate an equivalently sized buffer on the host, `data`, for convenience.

C++ | Fortran

```
real(real64), allocatable :: data(:)
real(real64), allocatable, device :: data_d(:)
integer(8) :: data_num_elements
...

data_num_elements = max(pinfo_x%size, pinfo_y%size, pinfo_z%size)

! Allocate device buffer
allocate(data_d(data_num_elements))

! Allocate host buffer
allocate(data(data_num_elements))
```

Working with pencil data

The pencil info structures are also used to access and manipulate data within the allocated pencil buffers. For illustrative purposes, we will use the X-pencil info structure here, but this will work for any of the axis pencils.

C/C++

First, here are examples of accessing/setting the pencil buffer data on the host in C/C++.

Here is an example of accessing the X-pencil buffer data on the host using a flattened loop:

```
for (int64_t l = 0; l < pinfo_x.size; ++l) {
  // Compute pencil-local coordinates, which are possibly in a permuted order.
  int i = l % pinfo_x.shape[0];
  int j = l / pinfo_x.shape[0] % pinfo_x.shape[1];
  int k = l / (pinfo_x.shape[0] * pinfo_x.shape[1]);

  // Compute global grid coordinates. To compute these, we offset the local coordinates
  // using the lower bound, lo, and use the order array to map the local coordinate order
  // to the global coordinate order.
  int gx[3];
  gx[pinfo_x.order[0]] = i + pinfo_x.lo[0];
  gx[pinfo_x.order[1]] = j + pinfo_x.lo[1];
  gx[pinfo_x.order[2]] = k + pinfo_x.lo[2];

  // Since the X-pencil also has halo elements, we apply an additional offset for the halo
  // elements in each direction, again using the order array to apply the extent to the
  // appropriate global coordinate.
  gx[pinfo_x.order[0]] -= pinfo_x.halo_extents[pinfo_x.order[0]];
  gx[pinfo_x.order[1]] -= pinfo_x.halo_extents[pinfo_x.order[1]];
  gx[pinfo_x.order[2]] -= pinfo_x.halo_extents[pinfo_x.order[2]];

  // Finally, we can set the buffer element, for example using a function based on the
  // global coordinates.
  data[l] = gx[0] + gx[1] + gx[2];
}
```

Alternatively, we can use a triple loop:

```
int64_t l = 0;
for (int k = pinfo_x.lo[2] - pinfo_x.halo_extents[pinfo_x.order[2]]; k < pinfo_x.hi[2] + pinfo_x.halo_extents[pinfo_x.order[2]]; ++k) {
  for (int j = pinfo_x.lo[1] - pinfo_x.halo_extents[pinfo_x.order[1]]; j < pinfo_x.hi[1] + pinfo_x.halo_extents[pinfo_x.order[1]]; ++j) {
    for (int i = pinfo_x.lo[0] - pinfo_x.halo_extents[pinfo_x.order[0]]; i < pinfo_x.hi[0] + pinfo_x.halo_extents[pinfo_x.order[0]]; ++i) {

      // i, j, k are global coordinate values. Use order array to map to global
      // coordinate order.
      int gx[3];
      gx[pinfo_x.order[0]] = i;
      gx[pinfo_x.order[1]] = j;
      gx[pinfo_x.order[2]] = k;

      // Set the buffer element.
      data[l] = gx[0] + gx[1] + gx[2];
      l++;
    }
  }
}
```

After assigning values on the host, we can copy the initialized host data to the GPU using `cudaMemcpy`:

```
CHECK_CUDA_EXIT(cudaMemcpy(data_d, data, pinfo_x.size * sizeof(*data), cudaMemcpyHostToDevice));
```

It is also possible to access/set the pencil data on the GPU directly within a CUDA kernel by passing in the pencil info structure to the kernel as an argument. For example, we can write a CUDA kernel to initialize the pencil buffer, using a similar access pattern as the flattened array example above:

```
__global__ void initialize_pencil(double* data, cudecompPencilInfo_t pinfo) {

  int64_t l = blockIdx.x * blockDim.x + threadIdx.x;

  if (l > pinfo.size) return;

  int i = l % pinfo.shape[0];
  int j = l / pinfo.shape[0] % pinfo.shape[1];
  int k = l / (pinfo.shape[0] * pinfo.shape[1]);

  int gx[3];
  gx[pinfo.order[0]] = i + pinfo.lo[0];
  gx[pinfo.order[1]] = j + pinfo.lo[1];
  gx[pinfo.order[2]] = k + pinfo.lo[2];

  gx[pinfo.order[0]] -= pinfo.halo_extents[pinfo.order[0]];
  gx[pinfo.order[1]] -= pinfo.halo_extents[pinfo.order[1]];
  gx[pinfo.order[2]] -= pinfo.halo_extents[pinfo.order[2]];

  data[i] = gx[0] + gx[1] + gx[2];

}
```

and launch the kernel, passing in `data_d` and `pinfo_x`:

```
int threads_per_block = 256;
int nblocks = (pinfo_x.size + threads_per_block - 1) / threads_per_block;
initialize_pencil<<<nblocks, threads_per_block>>>(data_d, pinfo_x);
```

Fortran

When using Fortran, it is convenient to use pointers associated with the pencil data buffers to enable more straightforward access using 3D indexing. For example, we can create pointers for each of the three pencil configurations, associated with a common host or device data array:

```
real(real64), pointer, contiguous :: data_x(:,:,:), data_y(:,:,:), data_z(:,:,:)
real(real64), pointer, device, contiguous :: data_x_d(:,:,:), data_y_d(:,:,:), data_z_d(:,:,:)
...

! Host pointers
data_x(1:pinfo_x%shape(1), 1:pinfo_x%shape(2), 1:pinfo_x%shape(3)) => data(:)
data_y(1:pinfo_y%shape(1), 1:pinfo_y%shape(2), 1:pinfo_y%shape(3)) => data(:)
data_z(1:pinfo_z%shape(1), 1:pinfo_z%shape(2), 1:pinfo_z%shape(3)) => data(:)

! Device pointers
data_x_d(1:pinfo_x%shape(1), 1:pinfo_x%shape(2), 1:pinfo_x%shape(3)) => data_d(:)
data_y_d(1:pinfo_y%shape(1), 1:pinfo_y%shape(2), 1:pinfo_y%shape(3)) => data_d(:)
data_z_d(1:pinfo_z%shape(1), 1:pinfo_z%shape(2), 1:pinfo_z%shape(3)) => data_d(:)
```

Here is an example of accessing the X-pencil buffer data on the host using a triple loop with the `data_x` pointer:

```
integer :: gx(3)
...

do k = 1, pinfo_x%shape(3)
  do j = 1, pinfo_x%shape(2)
    do i = 1, pinfo_x%shape(1)
      ! Compute global grid coordinates. To compute these, we offset the local coordinates
      ! using the lower bound, lo, and use the order array to map the local coordinate order
      ! to the global coordinate order.
      gx(pinfo_x%order(1)) = i + pinfo_x%lo(1) - 1
      gx(pinfo_x%order(2)) = j + pinfo_x%lo(2) - 1
      gx(pinfo_x%order(3)) = k + pinfo_x%lo(3) - 1

      ! Since the X-pencil also has halo elements, we apply an additional offset for the halo
      ! elements in each direction, again using the order array to apply the extent to the
      ! appropriate global coordinate
      gx(pinfo_x%order(1)) =  gx(pinfo_x%order(1)) - pinfo_x%halo_extents(pinfo_x%order(1))
      gx(pinfo_x%order(2)) =  gx(pinfo_x%order(2)) - pinfo_x%halo_extents(pinfo_x%order(2))
      gx(pinfo_x%order(3)) =  gx(pinfo_x%order(3)) - pinfo_x%halo_extents(pinfo_x%order(3))

      ! Finally, we can set the buffer element, for example using a function based on the
      ! global coordinates.
      data_x(i,j,k) = gx(1) + gx(2) + gx(3)

    enddo
  enddo
enddo
```

We can then copy the initialized host data to the GPU, in this case using direct assignment from CUDA Fortran:

```
data_d = data
```

We can also initialize the data directly on the device via a CUDA Fortran kernel, similar to the example shown in the C/C++ section above. For Fortran programs however, it is more common to use directive-based approaches like OpenACC or CUDA Fortran CUF kernel directives. For example, using an OpenACC directive (highlighted), we can directly use a triple loop like on the host to initialize the buffer on the device.

```
!$acc parallel loop collapse(3) private(gx)
do k = 1, pinfo_x%shape(3)
  do j = 1, pinfo_x%shape(2)
    do i = 1, pinfo_x%shape(1)
      ! Compute global grid coordinates. To compute these, we offset the local coordinates
      ! using the lower bound, lo, and use the order array to map the local coordinate order
      ! to the global coordinate order.
      gx(pinfo_x%order(1)) = i + pinfo_x%lo(1) - 1
      gx(pinfo_x%order(2)) = j + pinfo_x%lo(2) - 1
      gx(pinfo_x%order(3)) = k + pinfo_x%lo(3) - 1

      ! Since the X-pencil also has halo elements, we apply an additional offset for the halo
      ! elements in each direction, again using the order array to apply the extent to the
      ! appropriate global coordinate
      gx(pinfo_x%order(1)) =  gx(pinfo_x%order(1)) - pinfo_x%halo_extents(pinfo_x%order(1))
      gx(pinfo_x%order(2)) =  gx(pinfo_x%order(2)) - pinfo_x%halo_extents(pinfo_x%order(2))
      gx(pinfo_x%order(3)) =  gx(pinfo_x%order(3)) - pinfo_x%halo_extents(pinfo_x%order(3))

      ! Finally, we can set the buffer element, for example using a function based on the
      ! global coordinates.
      data_x_d(i,j,k) = gx(1) + gx(2) + gx(3)

    enddo
  enddo
enddo
```

Allocating workspace

Besides device memory to store pencil data, cuDecomp also requires workspace buffers on the device. For transposes, the workspace is used to facilitate local packing/unpacking and transposition operations (which are currently performed out-of-place). As a result, this workspace buffer will be approximately 2x the size of the largest pencil assigned to this process. For halo communication, the workspace is used to facilitate local packing of non-contiguous halo elements. We can query the required workspace sizes, in number of elements, using the cudecompGetTransposeWorkspaceSize and cudecompGetHaloWorkspaceSize functions.

C++ | Fortran

```
integer(8) :: transpose_work_num_elements, halo_work_num_elements

...

istat = cudecompGetTransposeWorkspaceSize(handle, grid_desc, transpose_work_num_elements)
call CHECK_CUDECOMP_EXIT(istat)

istat = cudecompGetHaloWorkspaceSize(handle, grid_desc, 1, [1,1,1], halo_work_num_elements)
call CHECK_CUDECOMP_EXIT(istat)
```

To allocate the workspaces, use the provided cudecompMalloc function. This allocation function will often use `cudaMalloc` to allocate the workspace buffer; however, if the grid descriptor passed in is using an NVSHMEM-enabled communication backend, it will use nvshmem_malloc to allocate memory on the symmetric heap, which is required for NVSHMEM operations (see NVSHMEM documentation for more details).

C++ | Fortran

```
real(real64), pointer, device, contiguous :: transpose_work_d(:), halo_work_d(:)
...

! Note: *_work_d arrays are of type consistent with cudecompDataType to be used (CUDECOMP_DOUBLE). Otherwise,
! must adjust workspace_num_elements to allocate enough workspace.
istat = cudecompMalloc(handle, grid_desc, transpose_work_d, transpose_work_num_elements)
call CHECK_CUDECOMP_EXIT(istat)

istat = cudecompMalloc(handle, grid_desc, halo_work_d, halo_work_num_elements)
call CHECK_CUDECOMP_EXIT(istat)
```

Transposing the data

Now, we can use cuDecomp's transposition routines to transpose our data. In these calls, we are using the `data_d` array as both input and output (in-place), but you can also use distinct input and output buffers for out-of-place operations. For the transposes between Y- and Z-pencils, we can pass null pointers to the halo extent arguments to the routines to ignore them in C/C++, or leave them unspecified in Fortran.

C++ | Fortran

```
! Transpose from X-pencils to Y-pencils.
istat = cudecompTransposeXToY(handle, grid_desc, data_d, data_d, transpose_work_d, CUDECOMP_DOUBLE, pinfo_x%halo_extents, [0,0,0])
call CHECK_CUDECOMP_EXIT(istat)

! Transpose from Y-pencils to Z-pencils.
istat = cudecompTransposeYToZ(handle, grid_desc, data_d, data_d, transpose_work_d, CUDECOMP_DOUBLE)
call CHECK_CUDECOMP_EXIT(istat)

! Transpose from Z-pencils to Y-pencils.
istat = cudecompTransposeZToY(handle, grid_desc, data_d, data_d, transpose_work_d, CUDECOMP_DOUBLE)
call CHECK_CUDECOMP_EXIT(istat)

! Transpose from Y-pencils to X-pencils.
istat = cudecompTransposeYToX(handle, grid_desc, data_d, data_d, transpose_work_d, CUDECOMP_DOUBLE, [0,0,0], pinfo_x%halo_extents)
call CHECK_CUDECOMP_EXIT(istat)
```

Updating halo regions

In this example, we have halos for the X-pencils only. We can use cuDecomp's halo update routines to update the halo regions of this pencil in the three domain directions. In this example, we set the `halo_periods` argument to enable periodic halos along all directions.

C++ | Fortran

```
! Setting for periodic halos in all directions
halo_periods = [.true., .true., .true.]

! Update X-pencil halos in X direction
istat = cudecompUpdateHalosX(handle, grid_desc, data_d, halo_work_d, CUDECOMP_DOUBLE, pinfo_x%halo_extents, halo_periods, 1)
call CHECK_CUDECOMP_EXIT(istat)

! Update X-pencil halos in Y direction
istat = cudecompUpdateHalosX(handle, grid_desc, data_d, halo_work_d, CUDECOMP_DOUBLE, pinfo_x%halo_extents, halo_periods, 2)
call CHECK_CUDECOMP_EXIT(istat)

! Update X-pencil halos in Z direction
istat = cudecompUpdateHalosX(handle, grid_desc, data_d, halo_work_d, CUDECOMP_DOUBLE, pinfo_x%halo_extents, halo_periods, 3)
call CHECK_CUDECOMP_EXIT(istat)
```

Cleaning up and finalizing the library

Finally, we can clean up resources. Note the usage of cudecompFree to deallocate the workspace arrays allocated with cudecompMalloc.

C++ | Fortran

```
deallocate(data)
deallocate(data_d)
istat = cudecompFree(handle, grid_desc, transpose_work_d)
call CHECK_CUDECOMP_EXIT(istat)
istat = cudecompFree(handle, grid_desc, halo_work_d)
call CHECK_CUDECOMP_EXIT(istat)
istat = cudecompGridDescDestroy(handle, grid_desc)
call CHECK_CUDECOMP_EXIT(istat)
istat = cudecompFinalize(handle)
call CHECK_CUDECOMP_EXIT(istat)
```

Building and running the example

Refer to the Makefiles in the basic usage example directories to see how to compile a program with the cuDecomp library.

Once compiled, the program can be executed using `mpirun` or equivalent parallel launcher.

We highly suggest making usage of the `bind.sh` shell script in the `utils` directory to assist in process/NUMA binding, to ensure processes are bound to node resources optimally (e.g. that processes are launched on CPU cores with close affinity to GPUs.) This is an example usage of the `bind.sh` script for Perlmutter system:

```
srun -N1 --tasks-per-node 4 --bind=none bind.sh --cpu=pm_map.sh --mem=pm_map.sh -- basic_usage
```

The `pm_map.sh` is a file (which can be found in the `utils` directory) containing the following:

```
bind_cpu_cores=([0]="48-63,112-127" [1]="32-47,96-111" [2]="16-31,80-95" [3]="0-15,64-79")

bind_mem=([0]="3" [1]="2" [2]="1" [3]="0")
```

These bash arrays list CPU core ranges (`bind_cpu_cores`) and NUMA domains (`bind_mem`) to pin each process to, by local rank. The `bind.sh` script will use these arrays to pin processes using `numactl`.