General Structure

The LibWWM source is partitioned into three parts:

- **Host components:** These are source files that generally manage the top-level logic behind the wall model solver. MPI communications, memory management, and IO are all managed in these functions.
- **Kernel components:** These files contain the CUDA kernel invocations and the declaration for the GPU buffer decomposition parameters.
- **Hybrid components:** This is where most of the work in the library happens. Common components involve operations to be done using the CUDA runtime API and, more importantly, all of the wall model solving code, which is run on both the device and the host.

Procedure

I. The wall model module is initialized. The MPI communicator is duplicated to prevent interference, and is also split so that each node has its own node communicator.

II. The main code (and the user of) is responsible for passing the necessary pointers using the Fortran-C binding. This can be done using the `ProvidedVariableAssociations` class by calling the `AssociateVariable` function.

III. Domain allocation occurs. This is where each process on a single node works out how much of the workload to put on the GPU/CPU. Another MPI communicator is generated for all of the processes with > 0 GPU-allocated points. Buffer offsets are computed.

IV. Buffer groups and transfer protocols are initialized:

  i . Global input, output, and solution buffers are named and sizes determined based on model selection.

  ii . Allocation modes are determined. This can essentially be thought of as a smart-pointer stack system with some conditional logic.

  iii . If necessary, host-device channels are set up to facilitate data transfer. Global device handles are brodcast via MPI.

  iv . Host-heap and device-global allocation occurs. These are guaranteed to have matching free calls by specifying appropriate protocols.

  (*timestep loop*)

V. If specified, a separate thread is started on each process with GPU-allocation to handle input data transfer. As the main code proceeds, this new thread offloads input data, launches the GPU solution kernel, solves the CPU allocation, and notifies the main thread that the solution is ready. This usually happens before the main thread is requesting the wall model solution anyway.

  (*end timestep loop*)

VI. All resources are finalized.

Hybrid Compute Kernels

Buffer Management

In this context, a buffer can simply be thought of as a working variable: An input (e.g. u_F), an output (e.g. τ_{wall}), or a solution variable (e.g. u, v, T etc.). Presumably, different models will use different buffers, so each model should only manage the necessary ones. For example, no buffer for T should be managed for an isothermal case. For each model, a number of buffers are specified:

```
add_managed<double>("u", &(managed_buffer->in.u), 1, 1, manage_mode | GPU_ENDPOINT);
```

This is a declaration for an input buffer of dimension 1, with 1 entry per wall point. `GPU_ENDPOINT` specifies that this variable is copied onto the GPU as needed for the GPU allocated solve.

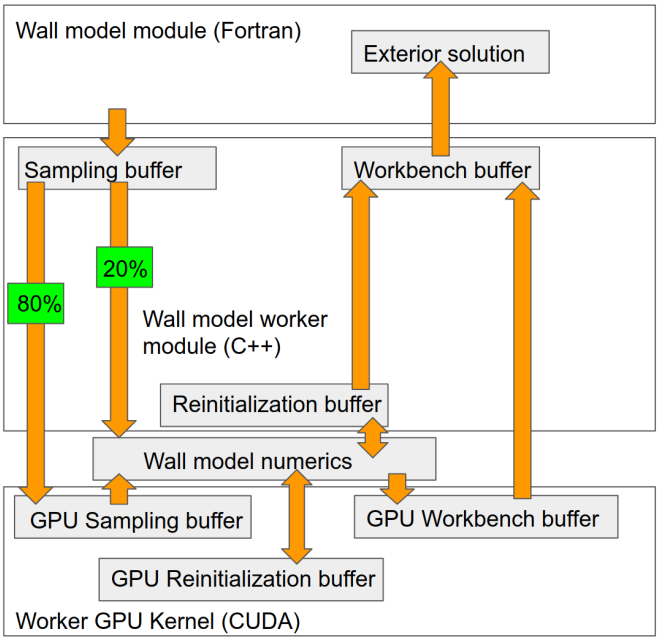

Figure 1: Overview of the data flow between the host and device.

```
add_managed<double>("stress_tensors", &(managed_buffer->out.stress_tensors), dim2, 1,
                                    manage_mode | CPU_ENDPOINT);
```

Here is an example output variable. The advantage of this system is that all buffers are defined in one place, and managed completely automatically.

Compiler Flags

Numerics

The ODE model can be reduced to a Newton root-find, specifically

$$\mathbf{X}_{k+1} = \mathbf{X}_k - \beta g\Big(J_c^{-1}\mathbf{f}(\mathbf{X}_k)\Big)ap$$

with

$$J_c = \begin{pmatrix} \frac{\mathrm{d}\mathbf{M_e}}{\mathrm{d}\mathbf{M_v}} & \alpha_1 \frac{\mathrm{d}\mathbf{M_e}}{\mathrm{d}\mathbf{T_v}} \\ \alpha_2 \frac{\mathrm{d}\mathbf{T_e}}{\mathrm{d}\mathbf{M_v}} & \frac{\mathrm{d}\mathbf{T_e}}{\mathrm{d}\mathbf{T_v}} \end{pmatrix}$$