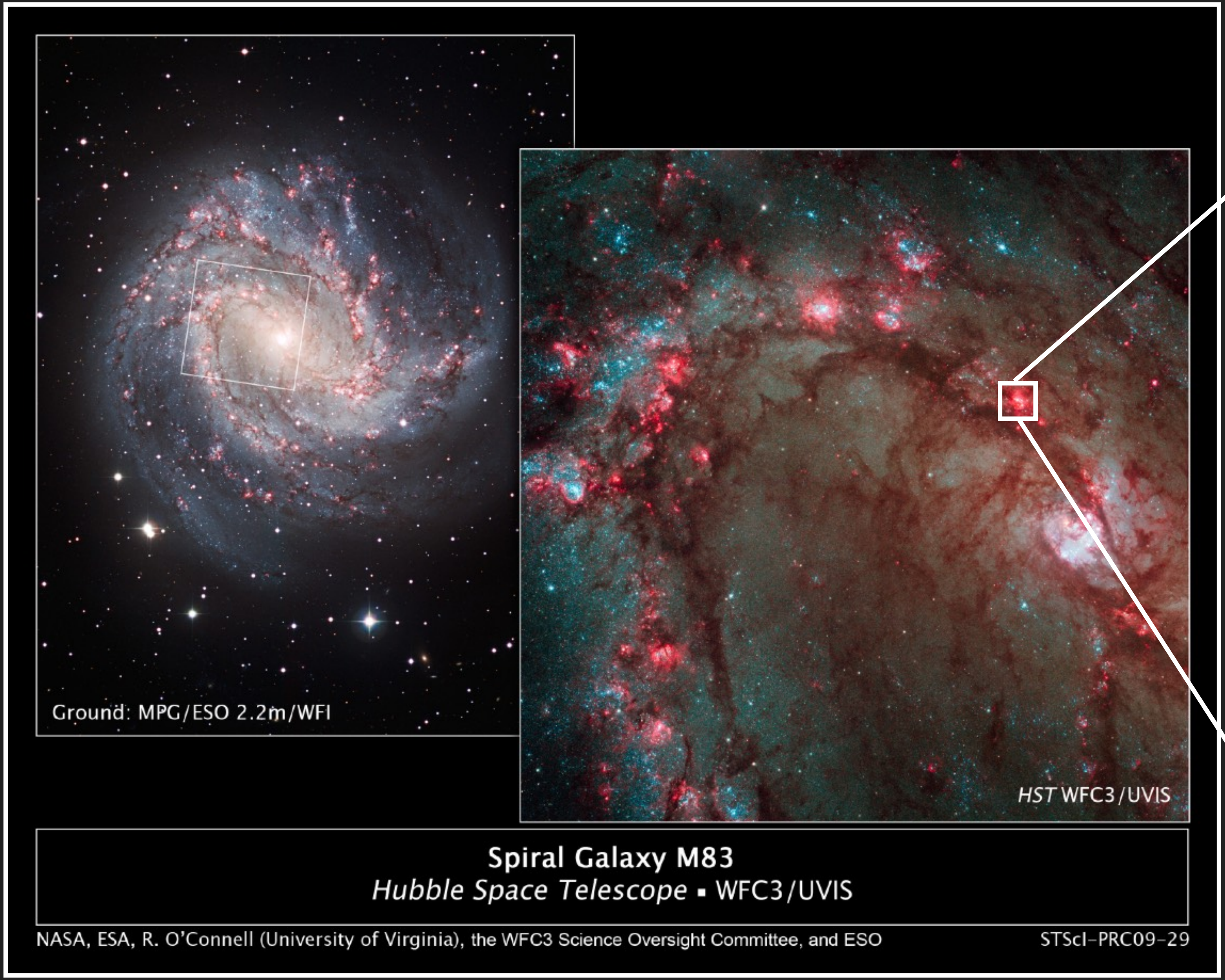
CRUSHER USER-EXPERIENCE TALKS

PREPARING CHOLLA FOR FRONTIER

December 9th 2022

SCIENTIFIC MOTIVATION:

GALAXY EVOLUTION AT PARSEC* SCALE


Ground: MPG/ESO 2.2m/WFI
HST WFC3/UVIS
Spiral Galaxy M83
Hubble Space Telescope ▪ WFC3/UVIS
NASA, ESA, R. O'Connell (University of Virginia), the WFC3 Science Oversight Committee, and ESO
STScI-PRC09-29

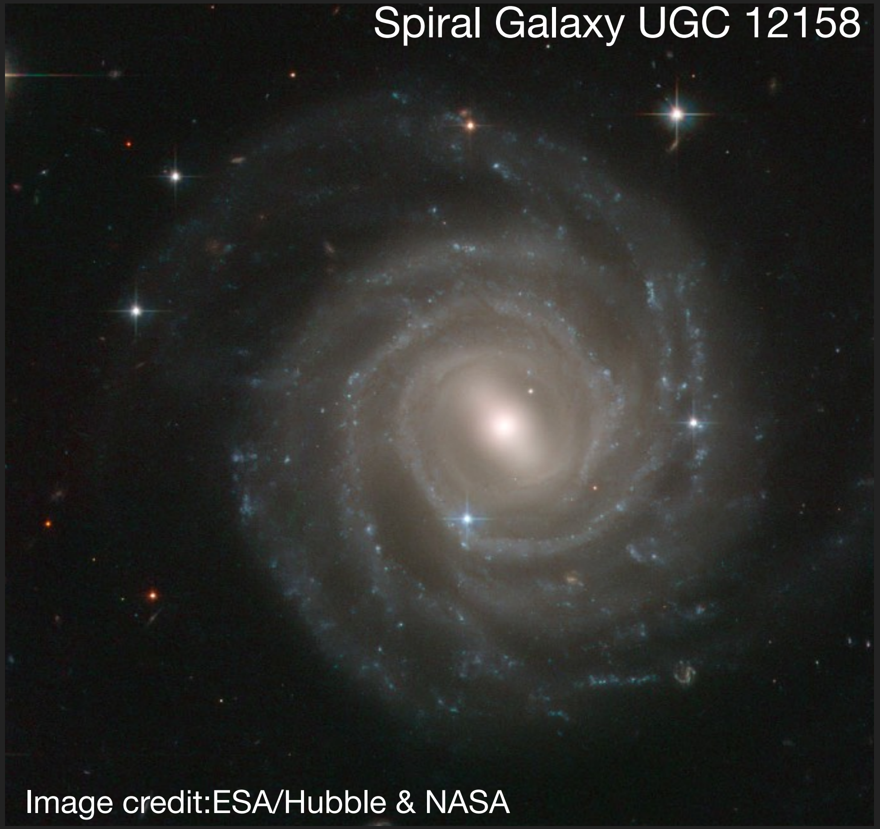
Star Cluster NGC 1929

*1 Parsec = a few light years

Image credit: X-ray: NASA/CXC/U.Mich./S.Oey, IR: NASA/JPL, Optical: ESO/WFI/2.2-m

SCIENTIFIC MOTIVATION:

- Goal is to simulate a Milky Way-like galaxy at a resolution that allows for self-consistent star formation and supernova explosions within a multiphase interstellar medium
- Milky Way diameter: ~30 kpc
- Resolution required to resolve star clusters: ~few pc
- Target resolution for "grand challenge" problem on Frontier ~10,000^3 cells

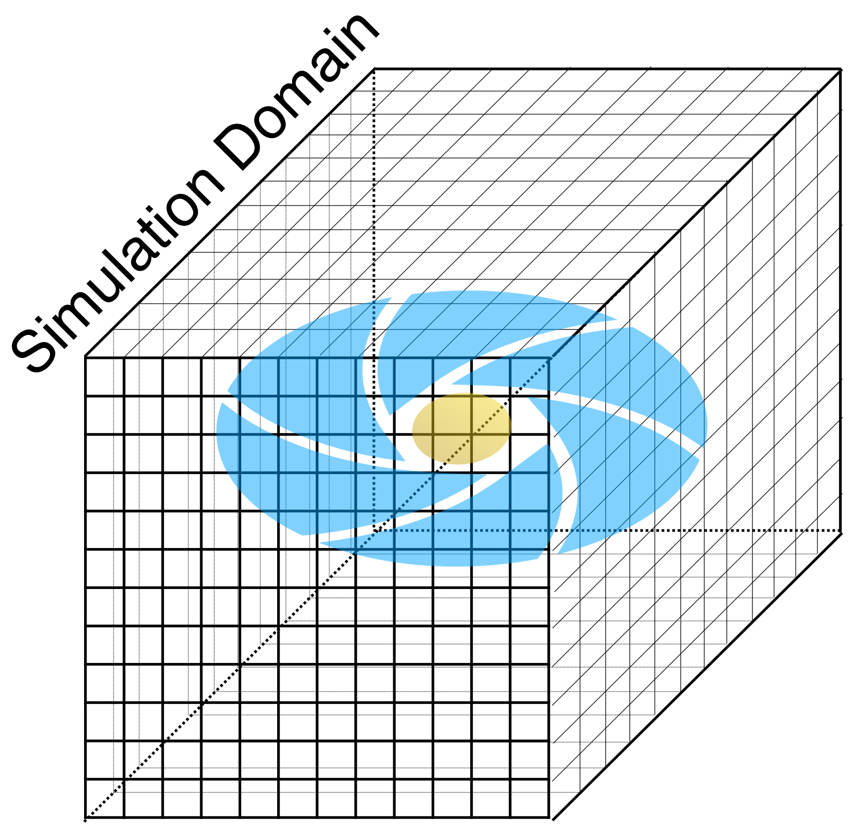


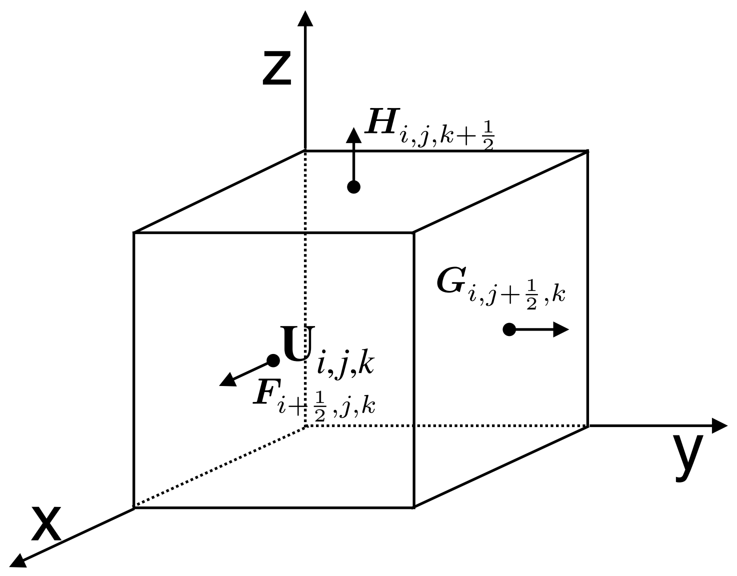
CHOLLA: COMPUTATIONAL HYDRODYNAMICS ON II ARCHITECTURES

- Cholla is a GPU-native, massively-parallel, finite-volume hydrodynamics code developed for astrophysics simulations
- Cholla is open source – code is publicly available at https://github.com/cholla-hydro/cholla
- All of the code development work I will discuss today is in the main branch of the public cholla repository

CHOLLA: COMPUTATIONAL HYDRODYNAMICS ON II ARCHITECTURES

- Cholla is a GPU-native, massively-parallel, **finite-volume** hydrodynamics code developed for astrophysics simulations




z
y
x
$\boldsymbol{H}_{i,j,k+\frac{1}{2}}$
$\boldsymbol{G}_{i,j+\frac{1}{2},k}$
$\mathbf{U}_{i,j,k}$
$\boldsymbol{F}_{i+\frac{1}{2},j,k}$

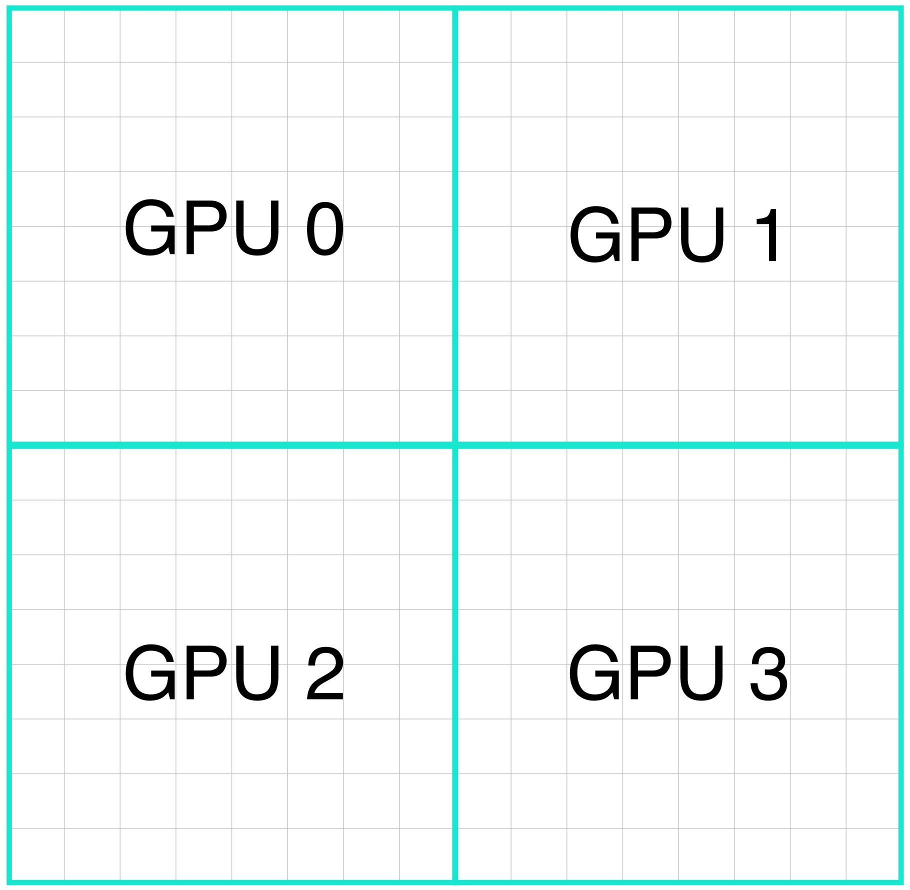
Finite Volume

$$U = [\rho, \rho u, \rho v, \rho w, E]^{\mathrm{T}}$$

$$\begin{aligned}\boldsymbol{U}^{n+1}_{i,j,k} = \boldsymbol{U}^{n}_{i,j,k} &- \frac{\delta t}{\delta x}\left(\boldsymbol{F}^{n+1/2}_{i+1/2,j,k} - \boldsymbol{F}^{n+1/2}_{i-1/2,j,k}\right)\\ &- \frac{\delta t}{\delta y}\left(\boldsymbol{G}^{n+1/2}_{i,j+1/2,k} - \boldsymbol{G}^{n+1/2}_{i,j-1/2,k}\right)\\ &- \frac{\delta t}{\delta z}\left(\boldsymbol{H}^{n+1/2}_{i,j,k+1/2} - \boldsymbol{H}^{n+1/2}_{i,j,k-1/2}\right)\end{aligned}$$

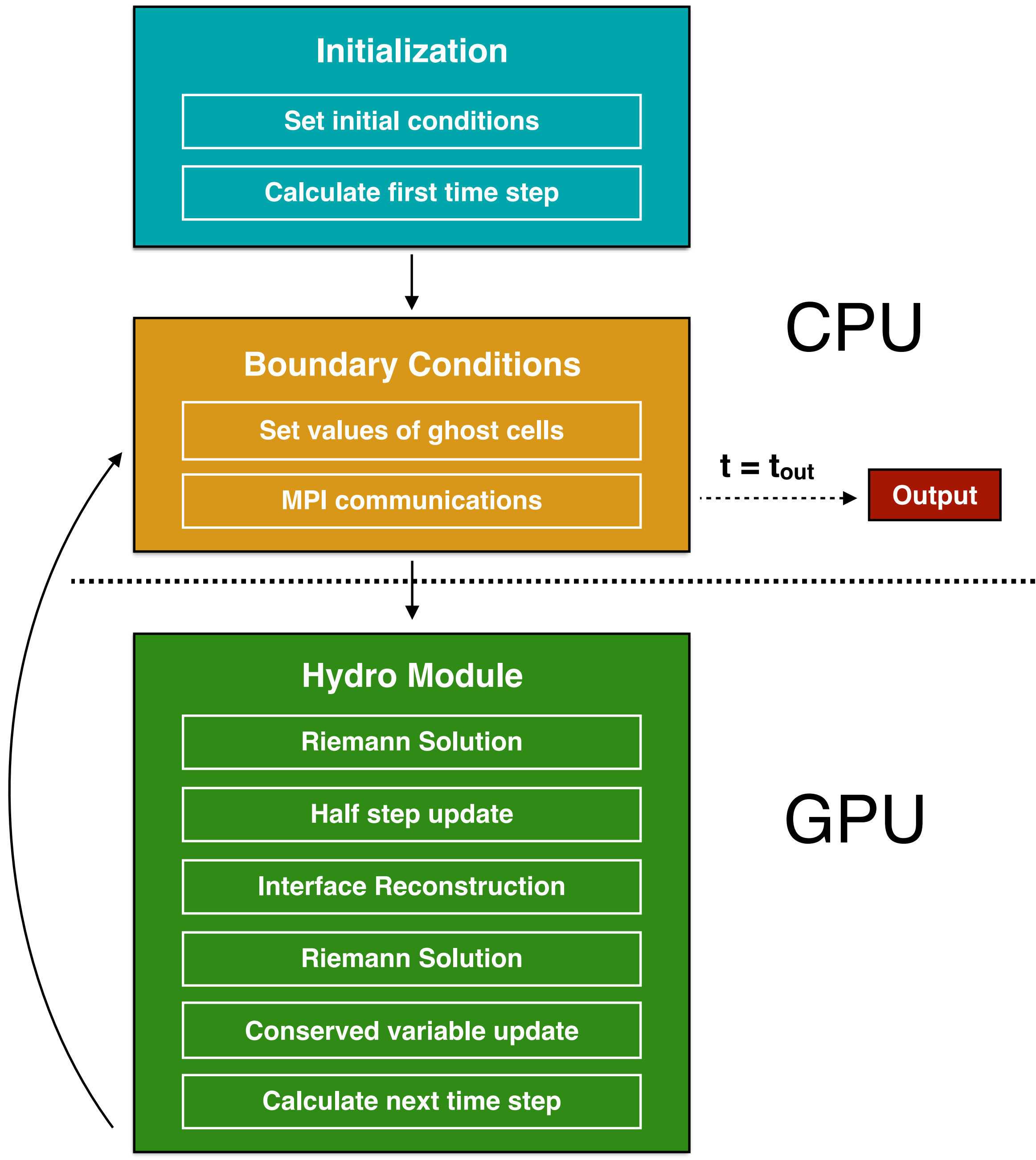
HOW DOES IT WORK? CHOLLA CIRCA 2019 (PRE-CAAR)

- Simulation domain is divided into sub volumes, each MPI rank is assigned a single sub-volume and a single GPU
- Typical sub-volume is 256^3 cells
- Each cell is mapped to a single thread on the GPU
- Subvolumes can be further divided if data size is too large to fit in memory on a single GPU



HOW DOES IT WORK? CHOLLA CIRCA 2019 (PRE-CAAR)

- Serial portions of the code execute on the CPU
- Parallel portions execute on the GPU
- Some of the new physics modules executed partially on the GPU, some executed exclusively on the CPU
- Fundamentally, the grids "lived" on the CPU, and were transferred to the GPU with every time step



FIRST TASK: PORTABILITY

- Cholla was written in C++ / Cuda / MPI / OpenMP
- Crusher (and Frontier) have AMD GPUs, which use HIP
- Solution: use HIP
  - Option one: HIPify
    - Use AMD-provided perl script to modify all cuda source files, changing all cuda syntax to hip syntax
    - hipcc compiles resulting code for either AMD or NVIDIA hardware
  - Option two: HIPifly!

HIPIFLY: HIP ON THE FLY

Added a single header file, gpu.hpp

```
#ifdef O_HIP

#define cudaDeviceSynchronize hipDeviceSynchronize

#define cudaError hipError_t

#define cudaError_t hipError_t

#define cudaErrorInsufficientDriver hipErrorInsufficientDriver

#define cudaErrorNoDevice hipErrorNoDevice
```

etc.

This means there is a single CUDA code base for both NVIDIA and AMD GPUs.

HIPIFLY MAKEFILE

```
ifdef HIPCONFIG
  DFLAGS     += -DO_HIP
  CXXFLAGS   += $(HIPCONFIG)
  GPUCXX     ?= hipcc
  LD         := $(CXX)
  LDFLAGS    := $(CXXFLAGS) -L$(ROCM_PATH)/lib
  LIBS       += -lamdhip64
else
  CUDA_INC   ?= -I$(CUDA_ROOT)/include
  CUDA_LIB   ?= -L$(CUDA_ROOT)/lib64 -lcudart
  CXXFLAGS   += $(CUDA_INC)
  GPUCXX     ?= nvcc
  GPUFLAGS   += --expt-extended-lambda -arch $(CUDA_ARCH) -fmad=false
  GPUFLAGS   += $(CUDA_INC)
  LD         := $(CXX)
  LDFLAGS    += $(CXXFLAGS)
  LIBS       += $(CUDA_LIB)
endif
```

HIPIFLY BUILD SYSTEM

```
#-- make.host for Frontier at the OLCF with
#-- Compiler and flags for different build type
CC                = cc
CXX               = CC
GPUCXX           ?= hipcc
CFLAGS_DEBUG      = -g -O0
CFLAGS_OPTIMIZE   = -g -O2
CXXFLAGS_DEBUG    = -g -O0 -std=c++14
CXXFLAGS_OPTIMIZE = -g -Ofast -std=c++14 -Wno-unused-result

GPUFLAGS          = --offload-arch=gfx90a -Wno-unused-result
HIPCONFIG      = -I$(ROCM_PATH)/include

#-- Libraries
MPI_ROOT          = ${CRAY_MPICH_DIR}
FFTW_ROOT         = $(shell dirname $(FFTW_DIR))
GOOGLETEST_ROOT := $(if $(GOOGLETEST_ROOT),$(GOOGLETEST_ROOT),$
(OLCF_GOOGLETEST_ROOT))

#-- Use GPU-aware MPI
MPI_GPU           = -DMPI_GPU
```

See also: https://github.com/cholla-hydro/cholla/tree/main/builds

SECOND TASK: DATA MUST LIVE ON THE GPU

- Originally, Cholla was designed to offload hydro calculations to the GPU every time step (largely to allow bigger grids)
  - As GPUs speeds have increased, CPU-GPU communication speeds have stayed roughly the same

  –> Data transfer was taking up a larger portion of the time step than hydro calculation!
- Solution: keep the hydro grid on the GPU, transfer boundary cells using GPU-aware MPI, only transfer the grid back to the CPU for output
  - Results in a ~4x speedup for hydro on Summit hardware

THIRD TASK: PORT THE FFT SOLVER

- The primary gravity solver in Cholla is FFT-based; our domain decomposition is block based – need a block-based FFT library to do Poisson solve
  - Previously, did this with PFFT on the CPU (using FFTW)
  - There was no existing parallel block-based FFT library for GPUs
- Solution: Write one. Trey White (HPE) wrote a block-based Poisson solver, **Paris**, that uses either cuFFT (Nvidia GPUS) or rocFFT (AMD GPUs) to perform FFTs on the GPU, and GPU-direct MPI communication

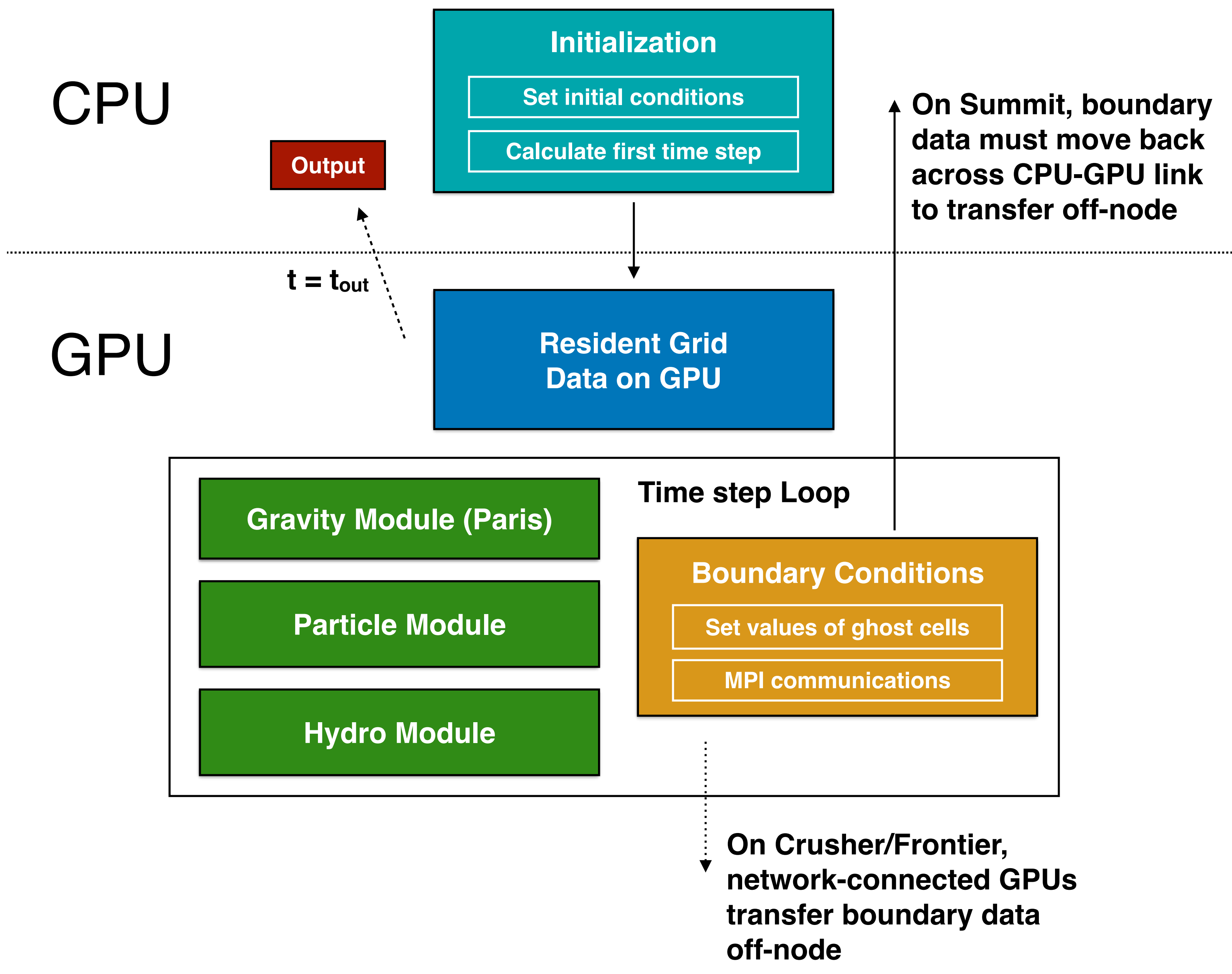
PARIS

- Paris moves all the following from the CPU to the GPU:
  - FFTs (now computed using cuFFT or rocFFT)
  - Poisson solve in frequency space
  - Buffers for MPI communication
  - Copies and transposes for changing dimensions in the 3D FFTs
  - Immediately saw at least a 3x speedup when using Paris vs PFFT

See also: https://github.com/cholla-hydro/cholla/tree/main/src/gravity/paris

HOW DOES IT WORK? CHOLLA CIRCA 2022

- Final step in the GPU-resident data transition:
  - Packing boundary buffers on the GPU (new Cuda kernels) and sending off-node using GPU-aware MPI


CPU
GPU
Initialization
Set initial conditions
Calculate first time step
Output
t = t_{out}
Resident Grid Data on GPU
Time step Loop
Gravity Module (Paris)
Particle Module
Hydro Module
Boundary Conditions
Set values of ghost cells
MPI communications
On Summit, boundary data must move back across CPU-GPU link to transfer off-node
On Crusher/Frontier, network-connected GPUs transfer boundary data off-node

See also: https://github.com/cholla-hydro/cholla/blob/main/src/grid/cuda_boundaries.cu

TRANSITION SLIDE

DOCUMENTATION

- Several old versions exist. Correct Docs at: https://docs.amd.com/
- ROCm is rapidly changing and so documentation is sometimes incomplete
- Documentation is improving
- CUDA documentation is often the best resource for anything that isn't documented by AMD

POINTER ATTRIBUTES

- When null: cudaPointerAttributes.device = -2 but hipPointerAttribute_t.device = 0
- cudaPointerAttributes.type → hipPointerAttribute_t.memoryType

```
typedef enum hipMemoryType {
    hipMemoryTypeHost,     ///< Memory is physically located on host
    hipMemoryTypeDevice,  ///< Memory is physically located on device.
    hipMemoryTypeArray,  ///< Array memory, physically located on device.
    hipMemoryTypeUnified  ///< Not used currently
} hipMemoryType;
```

```
enum cudaMemoryType
{
  cudaMemoryTypeUnregistered = 0, // Unregistered memory.
  cudaMemoryTypeHost = 1, // Host memory.
  cudaMemoryTypeDevice = 2, // Device memory.
  cudaMemoryTypeManaged = 3, // Managed memory
}
```

SHUFFLE

- HIP’s __shfl_down uses the same syntax as CUDA’s deprecated __shfl_down
- __shfl_down_sync doesn’t appear in HIP documentation
- No *_sync shuffle operations

ATOMICS

- HIP supports floating point atomics!
- Hardware or software floating point atomics

INSTALLATION

- Installation methods have changed and aren't always well documented
- Additional libraries (rocRAND, rocFFT, etc) don't always install the same version as the system ROCm unless you're very careful with the repos
- Might need to trick ROCm into thinking there's a GPU when there isn't
  - `echo "gfx90a" | sudo tee --append $(hipconfig -R)/bin/target`
- Docker containers work great. https://hub.docker.com/u/rocm

CLANG-TIDY

- Uses CUDA instead of ROCm backend
- Runs into compilation errors
  - /cholla/cholla/src/particles/feedback_CIC_gpu.cu:382:32: error: no matching function for call to 'atomicMax' [clang-diagnostic-error]
  - /opt/rocm-5.2.3/include/hiprand/hiprand_kernel_nvcc.h:43:1: error: typedef redefinition with different types ('struct hiprandStateMRG32k3a' vs 'struct curandStateMRG32k3a') [clang-diagnostic-error]