Portable and Performant GPU/Heterogeneous Asynchronous Many-Task Runtime System

Ph.D Research Proposal

Brad Peterson

Uintah Overview

- Asynchronous many-task (AMT) framework.
- Strong separation between runtime layer and task layer.
- Uintah provides data stores.
- Uintah prepares tasks then executes them when dependencies are met.
- Developing support for GPUs and Xeon Phis.

High-Level Goal

Demonstrate performant execution of production quality Uintah tasks on GPUs using tasks previously written with Kokkos constructs for CPUs and Xeon Phis.

Novelty of This Work

	AMT runtime	Developer involvement with runtime	Automatic internodal data movement	Automatic halo gathering	Runtime data store	Automatic data sharing among tasks	GPU support	Portable code for CPU and GPU tasks
Uintah	Yes	Light	Yes – local memory/MPI	Host mem. (This work for GPU)	Host mem. (This work for GPU)	This work	Yes	This work
Charm++	Yes	Medium	Invoked by user	No	No	No	With add-ons	Weak (add-ons required)
Legion	Yes	Heavy	Yes – global memory	Yes	No	Yes	Yes	No
HPX	Yes	Medium	Yes – global memory	No	No	No	No	No
OpenACC	No	N/A	No	No	No	No	Yes	Yes
OpenCL	No	N/A	No	No	No	No	Yes	Yes
CUDA Unified Memory	No	N/A	No	No	No	Yes	Yes	N/A
Kokkos	No	N/A	No	No	No	No	Yes	Yes

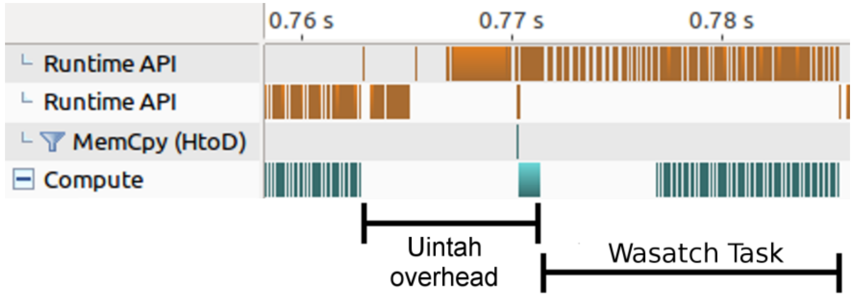
Defining Portable

Kokkos	Run Kokkos enabled tasks on GPUs, CPUs, and Xeon Phis with minimal architecture specific requirements in task code.
Task API	Preserve application layer API with minimal GPU runtime interaction.
Data Warehouse API	Keep data warehouse logic uniform for host and GPU memory.

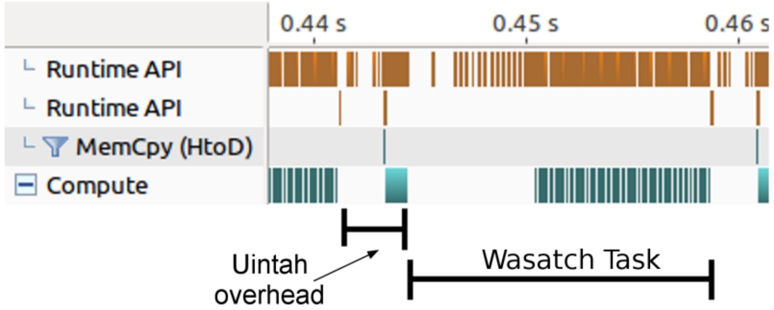
Defining Performant

- Low wall time overhead (in milliseconds) for task preparation.

Before:

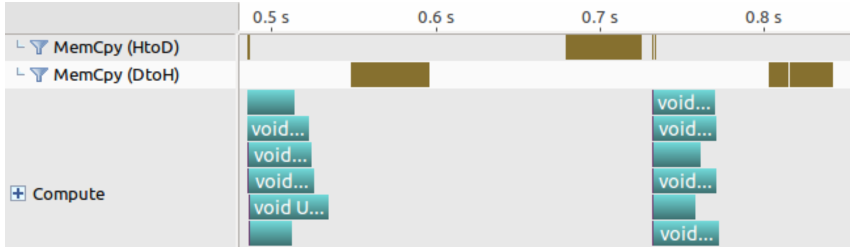


After:

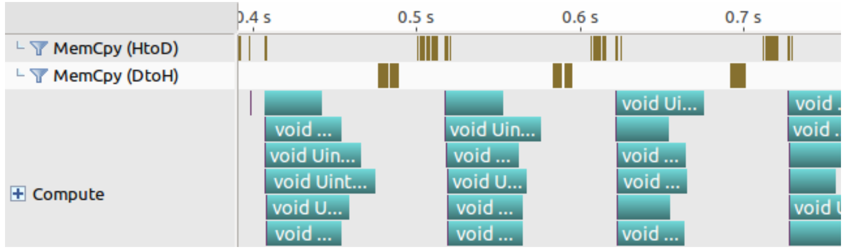


- Lower memory usage due to data sharing.

Before:



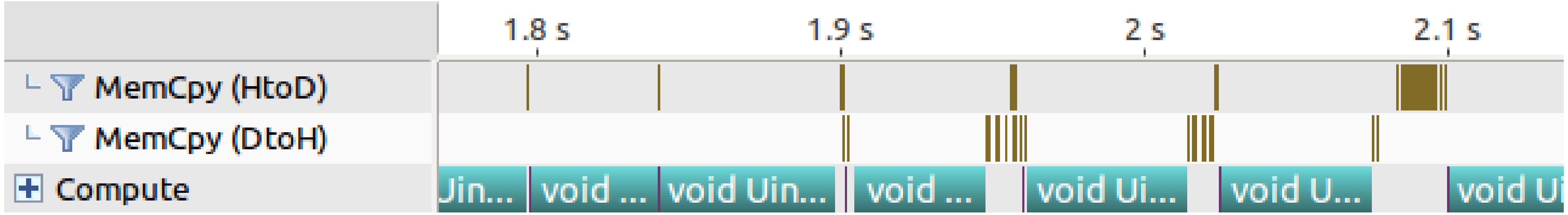
After:



- Similar Kokkos and non-Kokkos tasks execution wall times should be comparable.
  - Note: Kokkos can't force performant code, and users can still write slow code!

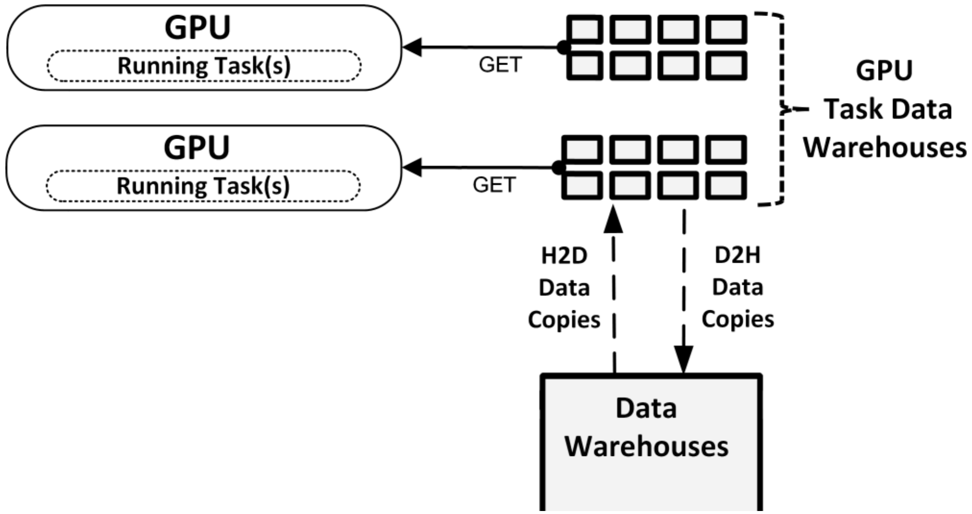
Prior Uintah GPU runtime

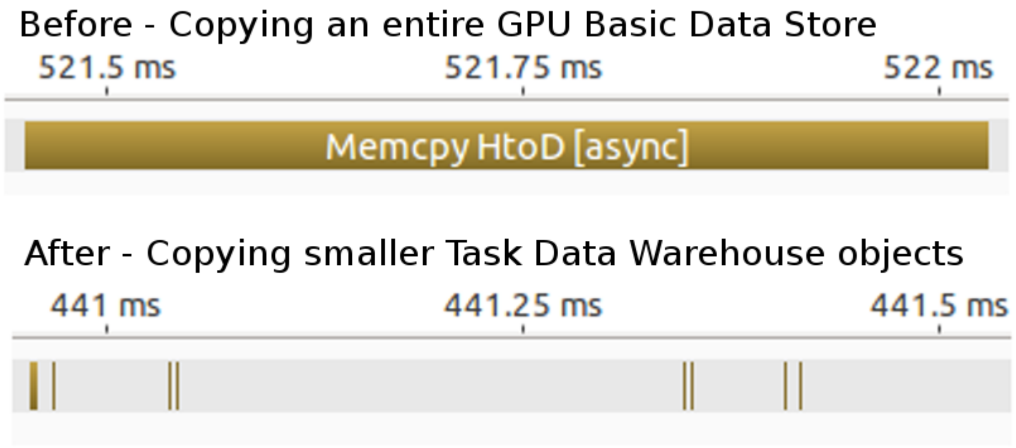
- Proof of concept
- Before GPU task execution
  - Halo gathers happened in host memory
  - Perform allocates and host-to-device copies
  - Copy monolithic GPU Data Store host-to-device
- After GPU task execution
  - Perform device-to-host copies
- Result: Serial execution of GPU tasks



Task GPU Data Warehouse

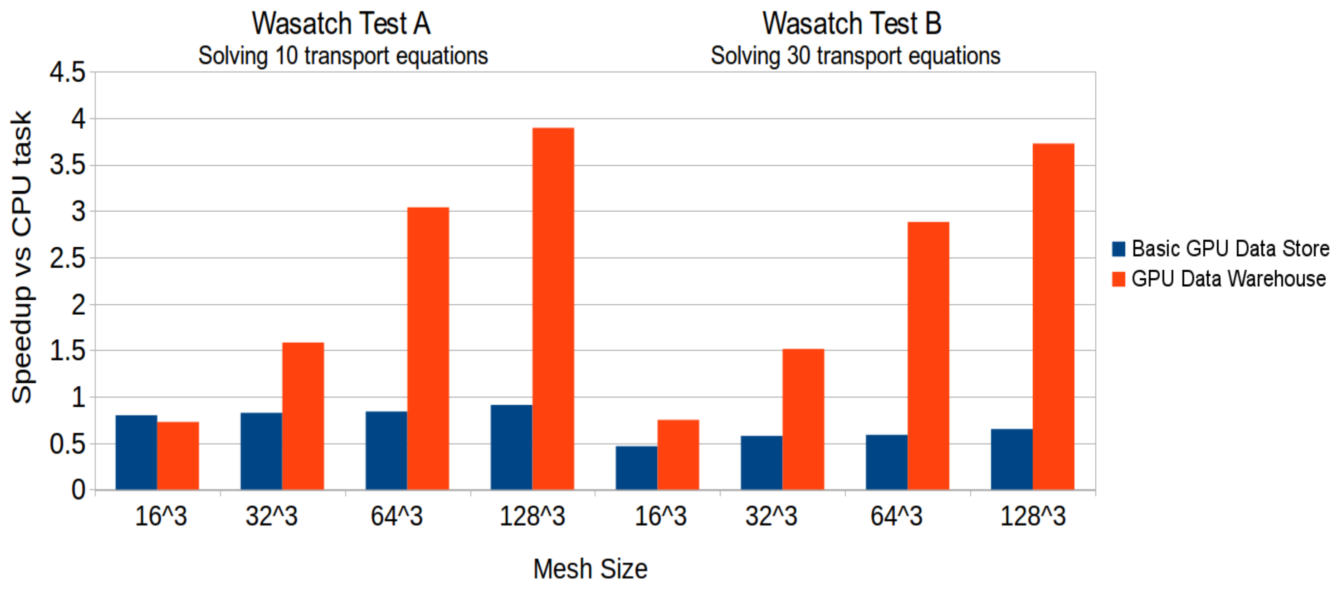
- Now each GPU task receives its own data warehouse.
- Difficult for asynchrony and concurrency to use one shared GPU data store.
- Tasks can *only* access simulation variables they indicated they would.





Tasks Dictate Data Persistence

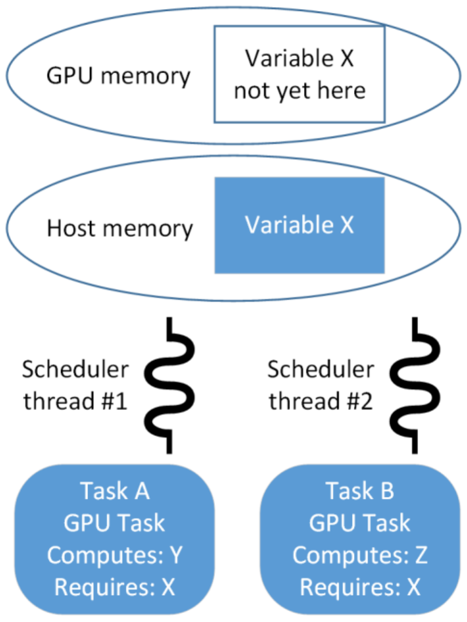
- Task scheduler now allocates and moves data into a memory space if it's not already there.
- Halo gathers stay within GPU memory if possible.
- This model works if Uintah ever switches to a dynamic task graph.



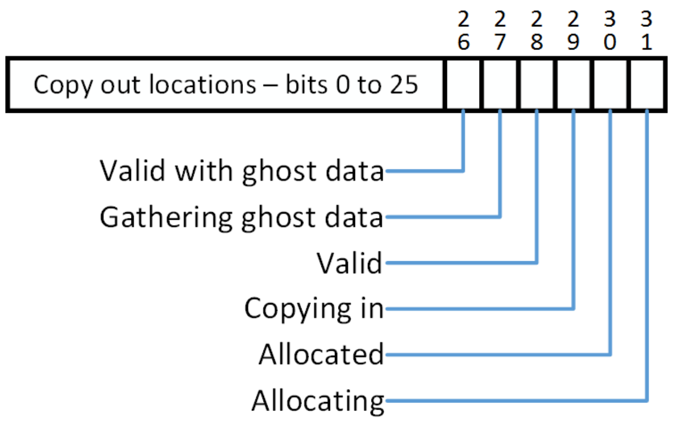
Wasatch tasks solving 10 and 30 transport PDEs respectively. Computations performed on a Nvidia 680 GPU and an Intel Xeon E5-2620.

Data Sharing Among Tasks

- Common use case: Two scheduler threads are assigned a different task to analyze. Each requires X in GPU memory, but it is not yet there.
- Task scheduler threads now coordinate with one another. (Before they were fully independent.)

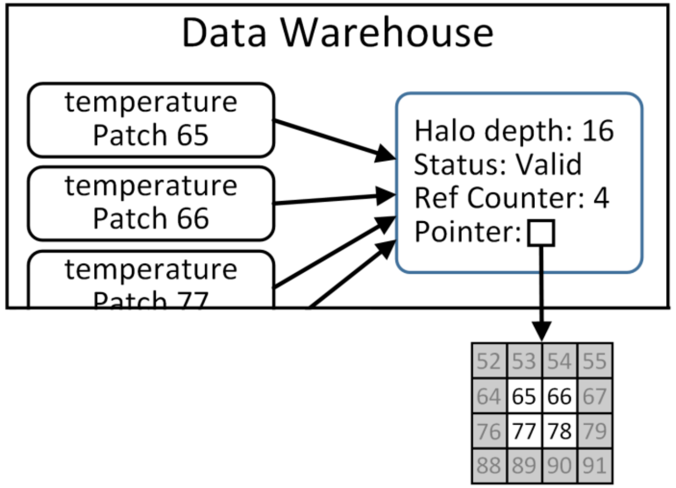


- Status bitset assigned for each simulation variable. Allows scheduler threads to coordinate.



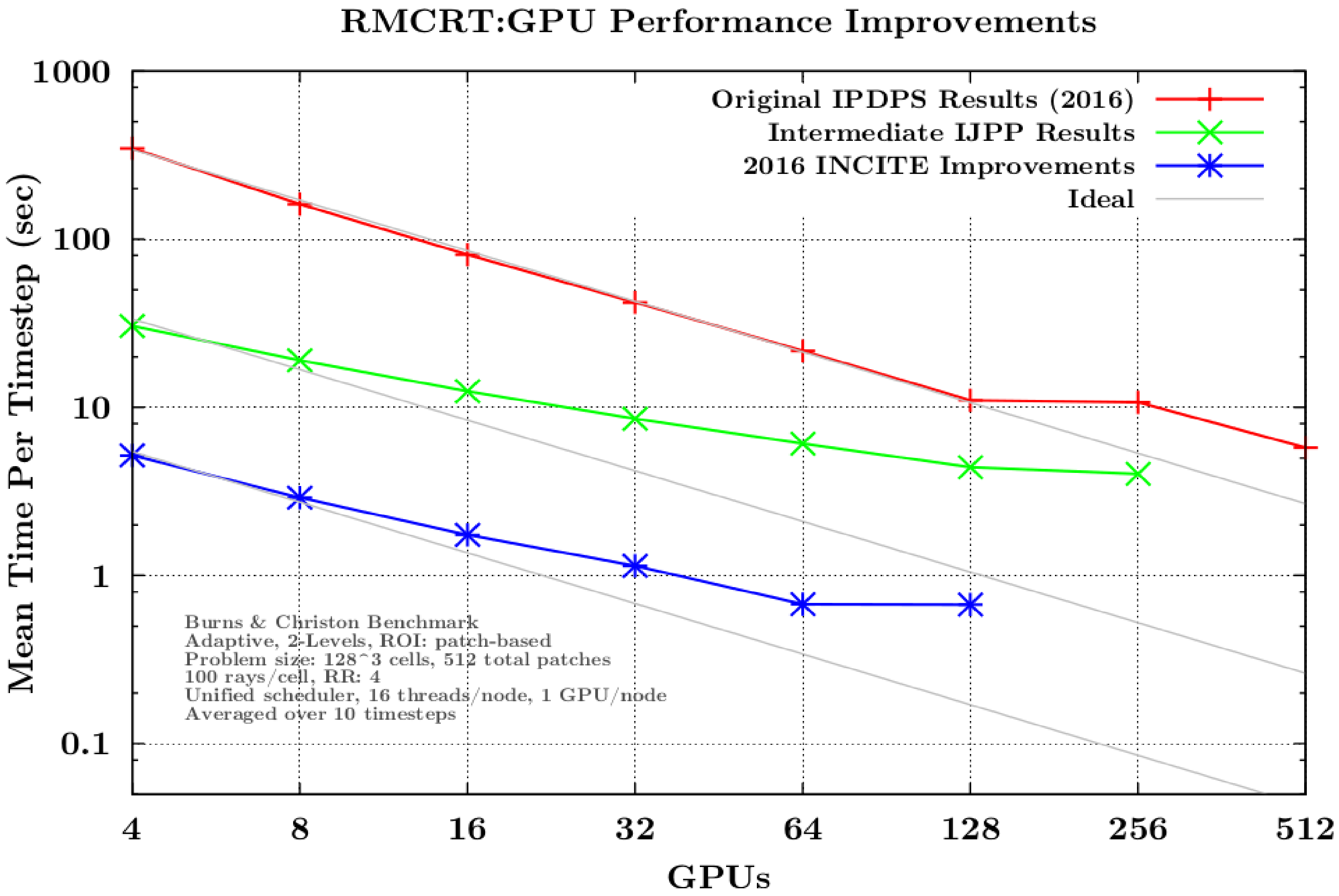
Data Sharing Among Data Dependencies

- Recent Titan production run of a propose high efficiency coal boiler had global data dependencies.
- Prior work could share a simulation variable among tasks, but not its halo data.
- Task scheduler and data warehouse changes facilitated a halo sharing mechanism.



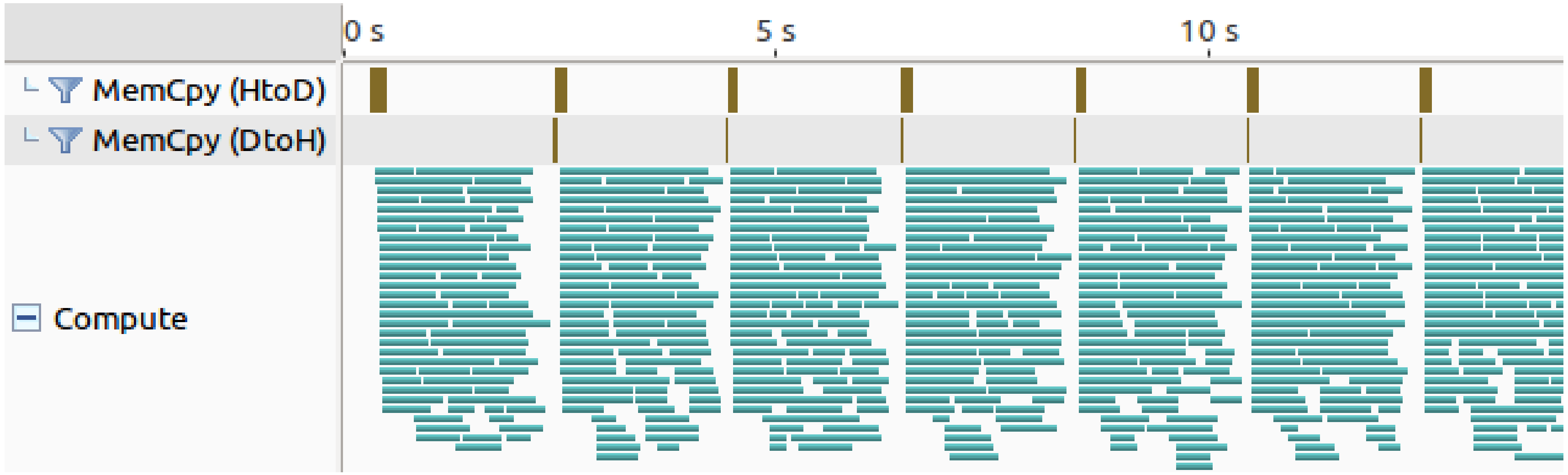
Memory Overhead Improvements		
Simulation Patch Layout	Host memory usage before (MB)	Host memory usage after (MB)
Coarse: 32^3 cells, 4^3 patches Fine: 64^3 cells, 4^3 patches	3073	65
Coarse: 32^3 cells, 4^3 patches Fine: 128^3 cells, 4^3 patches	23229	279
Coarse: 64^3 cells, 4^3 patches Fine: 128^3 cells, 4^3 patches	Exceeded memory	311

Uintah Improvements



Splitting Tasks into Multiple Streams and Kernels

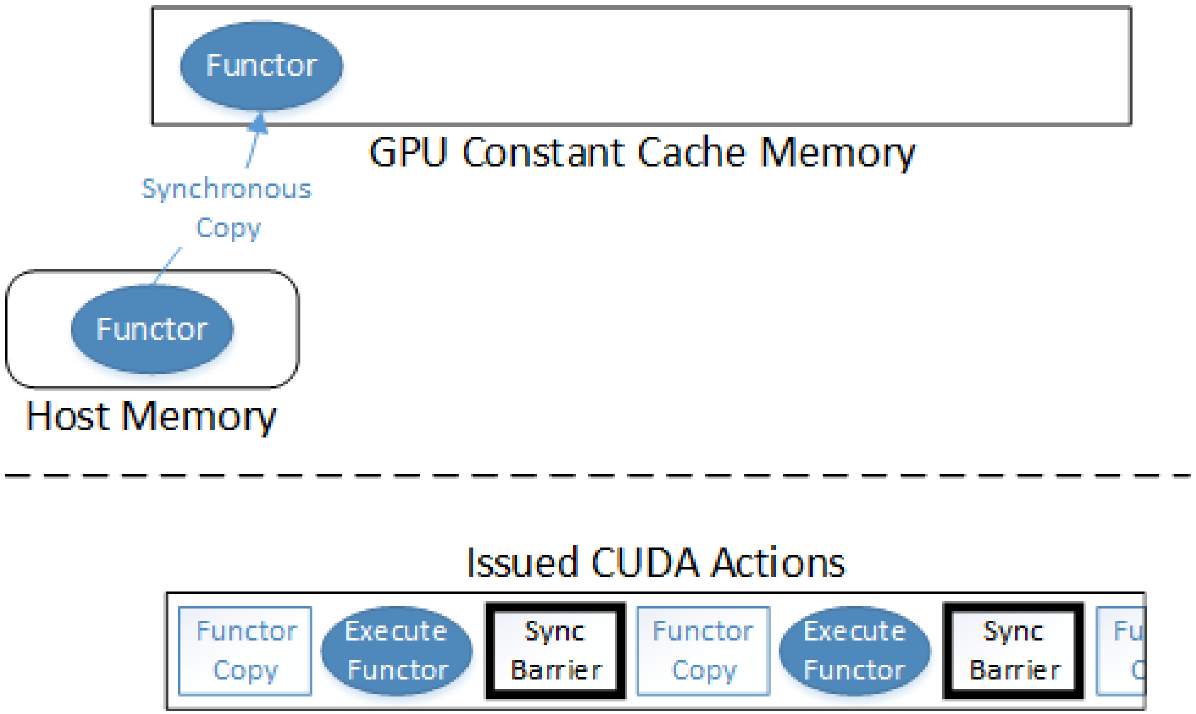
- A patch (a cuboid collection of cells) is the fundamental unit of Uintah decomposition and execution.
- Smaller patches mean more kernels but also more runtime overhead.
- A compromise is splitting tasks into multiple kernels, each launched on its own stream.
- Other approaches, such as multiple kernels on the same stream or one kernel in many blocks generated serialization.



RMCRT GPU tasks using 300 rays in a 28x28x56 cell domain in 16 patches - 8 CPU threads, 6 streams per kernel on a Nvidia K20c GPU

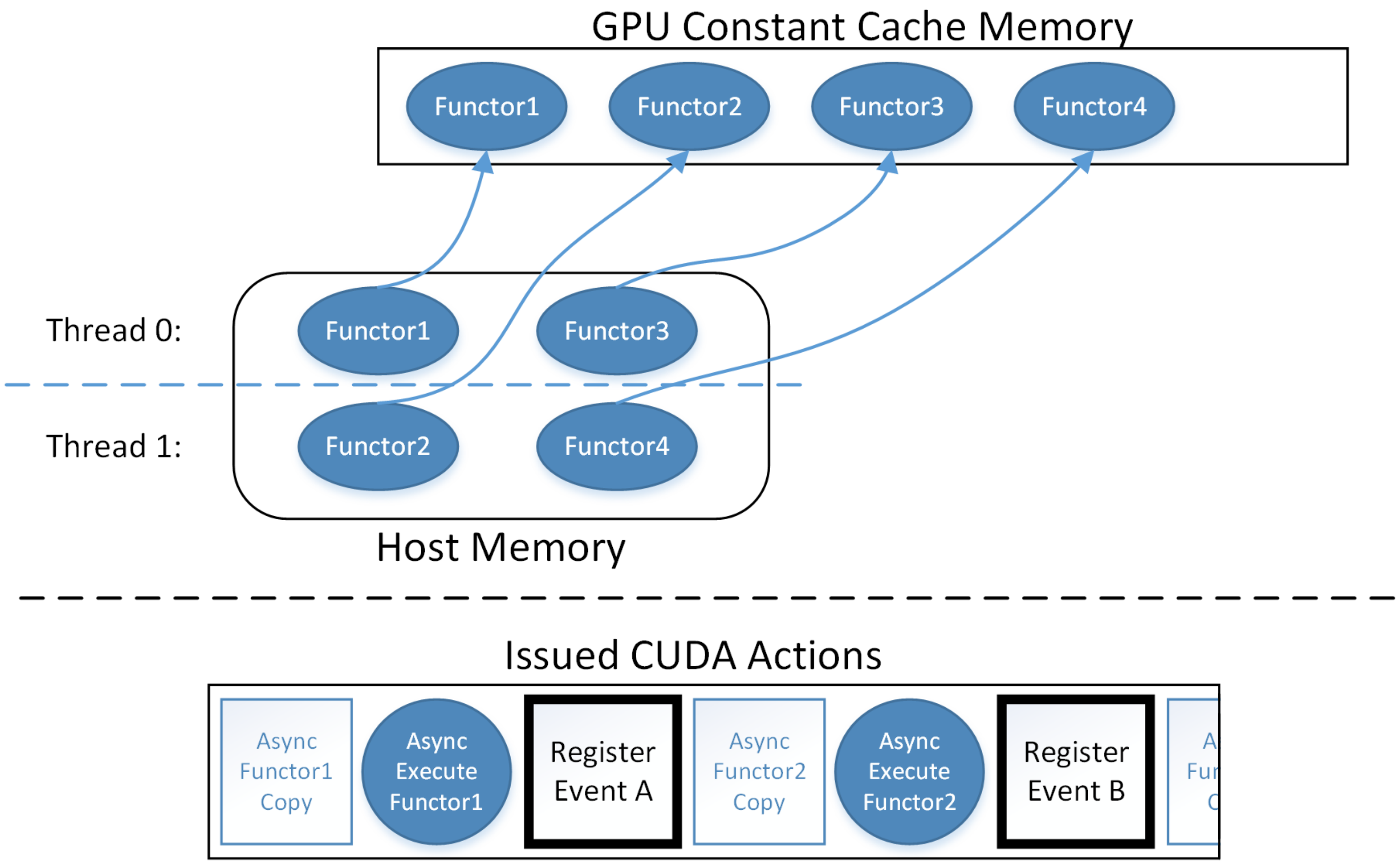
Current Status of Kokkos for GPUs

- GPU Constant Cache Memory acts as read-only registers.
- Kokkos::Parallel_for loops compiled as functors using gcc, nvcc, Intel's compiler, etc.
- Functor information loaded into GPU Constant Cache and functor executed.



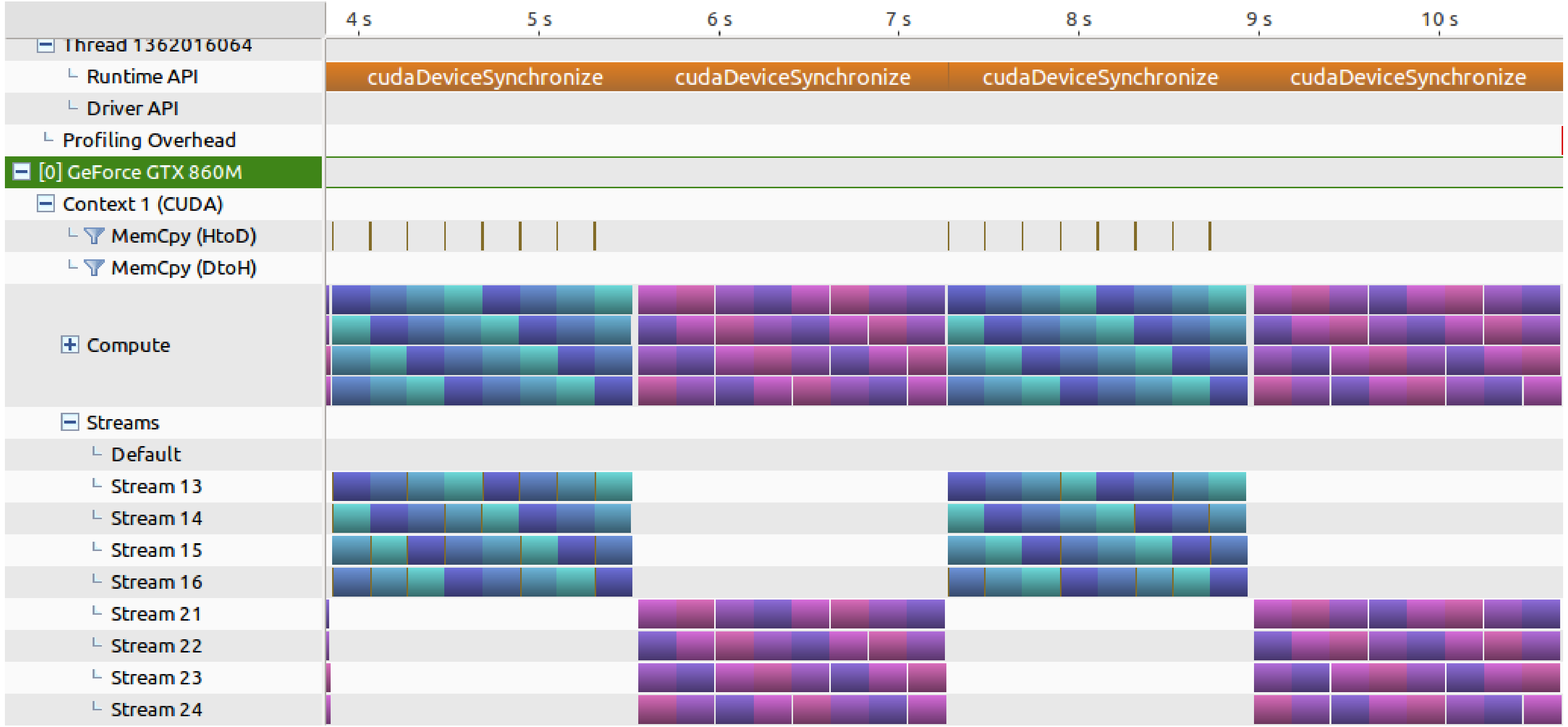
Modifying Kokkos for Multiple Functors

- I implemented a lock free allocation bitmap for functors.
- If the bitmap is full of functors, Kokkos checks all CUDA events to see if a prior functor completed, then can free that constant cache space.



Current Kokkos Results

- Blue is Kokkos parallel_for loops using Kokkos Cuda Instances.
- Purple is CUDA code using CUDA streams (both code sets use same logic).
- Both are executing on 4 streams and 4 different kernels/functors.



- Work is preliminary.

Future Work - Unified Data Warehouse Codebase

- Currently the OnDemand Data Warehouse (host memory) and GPU Data Warehouse take two different approaches for concurrency.
- Keep OnDemand Data Warehouse API and patch variable API.
- Place GPU Data Warehouse logic under the hood for data sharing and pre-allocation.

Future Work - Task Scheduler Modifications

- OnDemand Data Warehouse currently allocates variables on-the-fly during task execution.
- Need to adopt GPU Data Warehouse model where allocations occur prior.
- This will enable data sharing for the OnDemand Data Warehouse.
- Support modifies variables. (Effectively like a compute).

Future Work - Improved Halo Gathering

- OnDemand Data Warehouse’s halo gathering makes too many duplicates of variables.
  - Derek Harris is currently having big problems with this.
- GPU Data Warehouse gathers in halo cells through GPU kernels.
  - Both James Sutherland and I have noticed this is not as fast as we want.
- New model should:
  - Initiate all halo gathers initiated from CPU code,
  - Perform halo gathers only once per time step per simulation variable.
  - Intelligently pre-size computed variables in preparation for ghost cells.

Future Work - Continued Kokkos Modifications

- Investigate modifying Kokkos::parallel_reduce for asynchrony.
- Investigate multiple streams for multiple kernels within a parallel_for.
  - Past work has shown this is much better than single stream for multiple kernel blocks.
- Investigate register usage restriction. Vital for Titan production run speedups, very unclear how to generalize this into Kokkos API.

Future work - Uintah Integration with Kokkos

- The capstone of my work.
- Focus on tasks written with Kokkos code.
  - RMCRT and Arches tasks are the target problems.
  - Will not rewrite existing CPU tasks into Kokkos.
- Some tasks very likely can’t port if they depart from Uintah conventions.

Future work - Example of Kokkos GPU Integration

Application developer would add the object in red (supplied by the runtime).

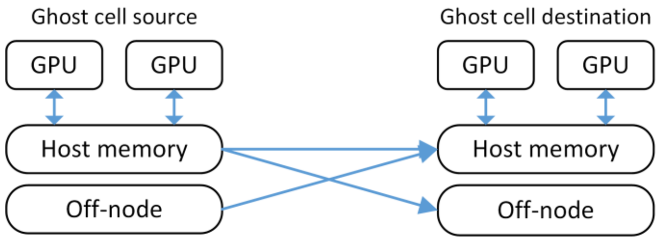
The runtime can ensure GPU, CPU, or Xeon Phi for the rest.

```
CCVariable<double> oxiSrc;
new_dw->allocateAndPut( oxiSrc, _src_label, matlIndex, patch );
constCCVariable<double> qn_gas_oxi;
new_dw->get( qn_gas_oxi, gasModelLabel, matlIndex, patch, gn, 0 );
Uintah::BlockRange range(patch->getCellLowIndex(),patch->getCellHighIndex());
sumCharOxyGasSource doSumOxyGasFunctor(qn_gas_oxi, oxiSrc);
Uintah::parallel_for(ExecutionInstanceObject, range, doSumOxyGasFunctor)
```

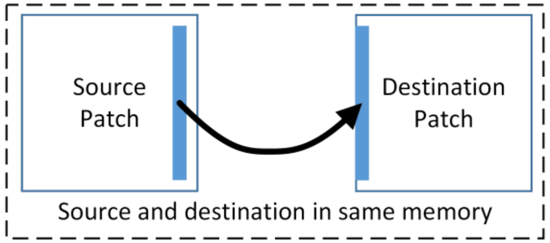
Questions?

Halo Gathers in GPU memory

- Before, all halo gathers happened in host memory



- Copying changed so halos can copy within the same memory region



- Halo buffers also sent into GPU memory if simulation variable resides there.