Uintah Portability Through Template Metaprogramming and Task Tagging

Brad Peterson

July 6, 2019

1 Introduction

Before Uintah's current template metaprogramming model, supporting CPUs and GPU tasks in the same build utilized an awkward mix of preprocessor ifdef statements, logical if statement branching, and special compile time options. This process was not sustainable. One attempted solution tried macros and automatic function call generation, but that again was becoming unwieldy and unsustainable. A solution was found using template metaprogramming, and this document briefly describes its structure.

2 Portability Needs

The list below summarizes desired needs from the perspective of both Uintah's runtime and application developers:

- Ease of use for application developers
- Backwards compatibility with non-Kokkos tasks and loops
- Multiple backend modes
- Ease of adding additional future modes
- Command line backend specific arguments (such as threads per loop)
- Overriding these command line arguments within individual tasks
- Multiple modes in the same compiled build
  - Single build can run heterogeneous mixture of tasks: legacy, CPU only (e.g. solvers), GPU
  - Easier unit testing of modes (fewer builds needed)
  - Easier GPU debugging (Kokkos's nvcc_wrapper terribly obfuscates source code. So debug portable code for CPUs instead!)

- Support task portability, not just loop portability
  - Data warehouse retrieval should be portable
  - Need to write our own boundary conditions
  - Need our own portable variable initialization
- Future possibility: Run tasks on both CPUs and GPUs simultaneously (most tasks are memory/latency bounded, could benefit from more buses)

3 Core Pieces

Template metaprogramming in Uintah did not come easy. The significant challenge was that Uintah's runtime system which controlled task compilation was intertwined with C++ polymorphism. Templates requires compile-time knowledge (i.e. early binding), and polymorphism requires runtime knowledge (i.e. late binding). The cleanest solution was utilizing a parameter object to carry template parameters and template deduction. This reduced the template metaprogramming to three core pieces, 1) task tagging, 2) a Uintah templated *Execution Object*, 3) functors, and 4) task code. These four are discussed in the following sections.

3.1 Task Tagging

An application developer needs a mechanism to state which modes a task can support. This process occurs at the point of code where the task is declared (in Uintah's case, that is in the task declaration code section which details the task requirements and its entry function).

From the application developer's perspective, an example of task tagging is shown in Figure 1. The yellow region in that figure is the key. This mechanism marks the start of Uintah task template metaprogramming. The tagged task template parameters propagate through several layers of Uintah, with propagation ending at the task code and task loops.

The task tagging approach solves two core needs. First, it allows the application developer to specify all possible modes the task supports. Second, it helps ensure the compiler only compile the desired modes. The key mechanism is the preprocessor defines for `UINTAH_CPU_TAG`, `KOKKOS_OPENMP_TAG`, and `KOKKOS_CUDA_TAG`. These three tags are defined in the Uintah runtime code as shown below:

```
#define COMMA ,
#define UINTAH_CPU_TAG UintahSpaces::CPU COMMA UintahSpaces::
    HostSpace
#define KOKKOS_OPENMP_TAG Kokkos::OpenMP COMMA Kokkos::HostSpace
#define KOKKOS_CUDA_TAG Kokkos::Cuda COMMA Kokkos::CudaSpace
```

The C++ preprocessor expands the relevant portion of the Poisson code example into the following:

```
void Poisson1::scheduleTimeAdvance( const LevelP     & level
                                  ,       SchedulerP & sched
                                  )
{
  auto TaskDependencies = [&](Task* task) {
    task->requires(Task::OldDW, phi_label, Ghost::AroundNodes, 1);
    task->computesWithScratchGhost(phi_label, nullptr,
                                   Uintah::Task::NormalDomain,
                                   Ghost::AroundNodes, 1);
    task->computes(residual_label);
  };

  create_portable_tasks(TaskDependencies, this,
                        "Poisson1::timeAdvance",
                        &Poisson1::timeAdvance<UINTAH_CPU_TAG>,
                        &Poisson1::timeAdvance<KOKKOS_OPENMP_TAG>,
                        &Poisson1::timeAdvance<KOKKOS_CUDA_TAG>,
                        sched, level->eachPatch(),
                        m_sharedState->allMaterials(),
                        TASKGRAPH::DEFAULT);

}
```

Figure 1: Task declaration with Uintah's new portable mechanisms.

```
create_portable_tasks( . . . ,
&Poisson1::timeAdvance<UintahSpaces::CPU, UintahSpaces::HostSpace>,
&Poisson1::timeAdvance<Kokkos::OpenMP, Kokkos::HostSpace>,
&Poisson1::timeAdvance<Kokkos::Cuda, Kokkos::CudaSpace>,
. . .)
```

That code would then attempt to compile three different flavors of the templated function `Poisson1::timeAdvance<>()`. Note that the first template argument is the execution space, and the second template argument is the memory space.

3.1.1 Conditional Mode Compilation

Suppose CUDA support was not desired. Here, Uintah has a configure time define which demotes the CUDA tag into the following (note the third line changing to OpenMP):

```
#define COMMA ,
#define UINTAH_CPU_TAG UintahSpaces::CPU COMMA UintahSpaces::
    HostSpace
#define KOKKOS_OPENMP_TAG Kokkos::OpenMP COMMA Kokkos::HostSpace
#define KOKKOS_CUDA_TAG Kokkos::OpenMP COMMA Kokkos::HostSpace
```

Now, the task declaration in Figure 1 will be seen by the compiler as listing `&Poisson1::timeAdvance<Kokkos::OpenMP, Kokkos::HostSpace>` twice, and so the compiler only compiles one of them. Suppose Uintah is not configured with OpenMP or CUDA, then all three tags turn into `&Poisson1::timeAdvance<UintahSpaces::CPU, UintahSpaces::HostSpace>`,

and so only 1 pthread mode is enabled. Note that at the time of this writing, Uintah will only compile at most two of these in a single build, as OpenMP and pthreads do not mix well together.

3.2 Execution Object

```
template <typename ExecSpace, typename MemSpace>
void Poisson1::timeAdvance( const PatchSubset* patches,
                            const MaterialSubset* matls,
                            OnDemandDataWarehouse* old_dw,
                            OnDemandDataWarehouse* new_dw,
                            UintahParams& uintahParams,
                            ExecObject<ExecSpace, MemSpace>& executionObject )
{
  int matl = 0;
  for (int p = 0; p < patches->size(); p++) {
    const Patch* patch = patches->get(p);

    auto phi = old_dw->getConstNCVariable<double, MemSpace> (phi_label, matl, patch,
                                                             Ghost::AroundNodes, 1);
    auto newphi = new_dw->getNCVariable<double, MemSpace> (phi_label, matl, patch);

    double residual = 0;
    IntVector l = patch->getNodeLowIndex();
    IntVector h = patch->getNodeHighIndex();

    Uintah::BlockRange range( l, h );

    Uintah::parallel_for(executionObject, range, KOKKOS_LAMBDA(int i, int j, int k){
        newphi(i, j, k) = phi(i,j,k);
    });

    Uintah::parallel_reduce_sum(executionObject, range,
                                KOKKOS_LAMBDA (int i, int j, int k, double& residual){
      newphi(i, j, k) = (1. / 6) * (
          phi(i + 1, j, k) + phi(i - 1, j, k) +
          phi(i, j + 1, k) + phi(i, j - 1, k) +
          phi(i, j, k + 1) + phi(i, j, k - 1));
      double diff = newphi(i, j, k) - phi(i, j, k);
      residual += diff * diff;
    }, residual);
  }
}
```

Figure 2: The portable Uintah Poisson task code capable of CPU and GPU compilation and execution.

The **Execution Object** can be seen as the last argument of `timeAdvance` in Figure 2. This object carries with it the execution space and the memory space information, which were defined previously in Section 3.1 in the task tagging phase.

A crucial piece is creating this Execution Object. The process is best understood in the full flow, with the Execution object creation occurring in step 5c:

1. The `create_portable_tasks` is called (see Figure 1). (Defined in

`Core/Parallel/TaskDeclaration.h`.)

2. The function creates a `Task` object for each execution space and memory space. (Defined in `Core/Grid/Task.h`.)
   - The function also assigns a portable mode priority and launches the task in the appropriate backend
   - The function also ensures that user placed tags (Section 3.1) does not require a specific order
3. The `Task` object carries template information and creates an `ActionPortable` object
4. The `ActionPortable` stores the task function pointer address, preserving template parameters
5. During execution:
   (a) A `DetailedTask` object is created, which inherits the `Task` class.
   (b) Uintah invokes a `DetailedTask` object's `do_it()` method
   (c) The `do_it()` method instantiates an `ExecutionObject`. Template information is preserved as `do_it()` is defined inside of `class ActionPortable`
   (d) The `ExecutionObject` is loaded with additional backend specific parameters
   (e) The `do_it()` method calls `doit_impl()`
   (f) The `doit_impl()` method invokes the function pointer address
   (g) The application developer's task code executes with template information

For particular details, please refer to the specific code files themselves. The prior list may seem too long at first glance, and the temptation is to find a way to shorten this list by reducing steps. Unfortunately, the mixture of `DetailedTask`'s inheritance using late binding, with template metaprogramming's early binding, made the process difficult and necessitated each step. Polymorphism can't carry template information. Function pointers do not normally store template information either. The key insight is to carry template information via a templated object in a function pointer.

3.3 Functors

This section is somewhat obvious to experienced developers. I mentioned it for two additional insights. First, some forms of code portability are possible outside of functors, with OpenMP, OpenACC, OCCA, and Charm++ being prominent examples. However, in my opinion, functors give Uintah the greatest overall cost vs. benefit net gain. Second, functors allow Uintah to have portability outside of Kokkos, should such a need ever arise. We do not need to lock ourselves into Kokkos's model.

3.4 Task Code

The task code itself can now use the template parameters to aid in template metaprogramming. Refer back to Figure 2. The data warehouse calls can be effectively overloaded to return a different type of object depending on the memory space templated argument used. Note that the application developer uses the C++ `auto` keyword. Normally, C++ does not allow overloading a function call by return type, but the concept works with template metaprogramming and `auto`.

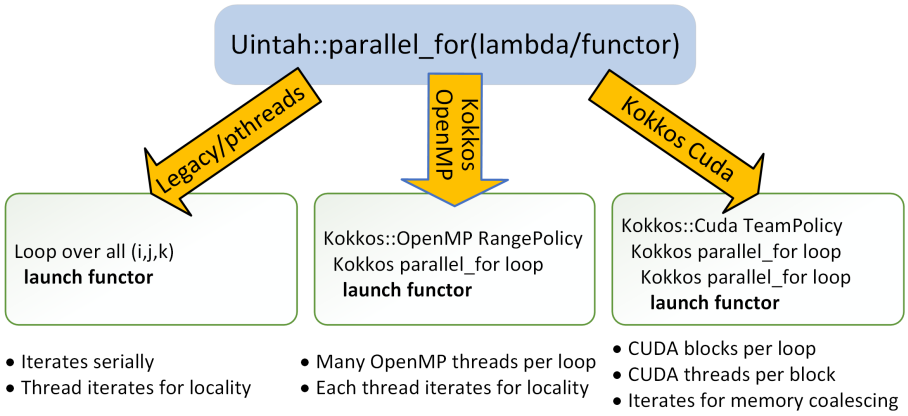

Figure 3: Uintah's `parallel_for` supports three portable execution modes.

Uintah's parallel loops avoid requiring template chevron syntax. The Execution Object again carries the templated information. Uintah relies on C++ template deduction to pass the template information. Figure 3 depicts how Uintah calls the correct parallel loop for the matching template execution space. From here, Uintah is able to supply its own architecture specific backend loop options. We personally found that Kokkos's portability options were insufficient and we needed more control. Further, we retain control to run our functors via pthreads in case Kokkos were not used at all.

3.5 Overview

The process described thus far is also summarized in Figure 4

4 Task Declaration, Configuration, Compilation, Runtime

Another helpful way to understand Uintah's template metaprogramming is to view the process in terms of building and using Uintah. The task defines all possible modes that a particular task supports. Uintah's configure script sets

appropriate defines to tell all possible modes that should be supported at compilation, and likewise sets the defines that guard the appropriate tags (Section 3.1). The compilation process then uses the templates ensuring all desires modes are compiled, and thus tasks can be compiled for more than one backend. The runtime process is controlled by Uintah command line arguments, and each task will only run on one backend. A heterogeneous mix of tasks is easily supported, but an individual task itself cannot run on both CPUs and GPUs, for example. The process described thus far is also summarized in Figure 5.

5 Conclusion

Uintah's template metaprogramming now supports the needs listed in Section 2. Portability is easily managed by application developers by simply declaring modes a task can support. Uintah supports heterogeneous tasks, and thus tasks can be tested and supported on new backends one-by-one. Additionally, future portability modes are easily supported with just a few minor modifications, so that Uintah need not be locked into Kokkos, CUDA, OpenMP, or any other parallel programming model.

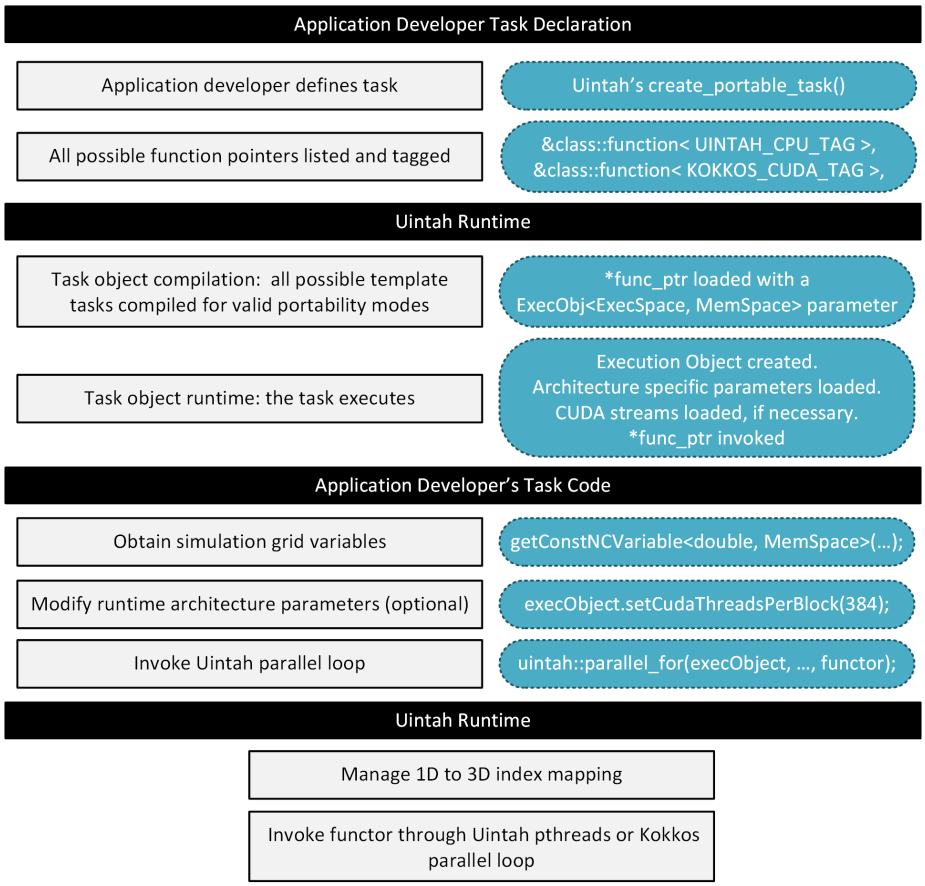

Figure 4: Template metaprogramming of Uintah portable code starts at the task declaration phase, propagates into the runtime, then back into task code, then into Uintah's parallel API, then into Kokkos's parallel API. The *Execution Object* is a central piece of this template metaprogramming.

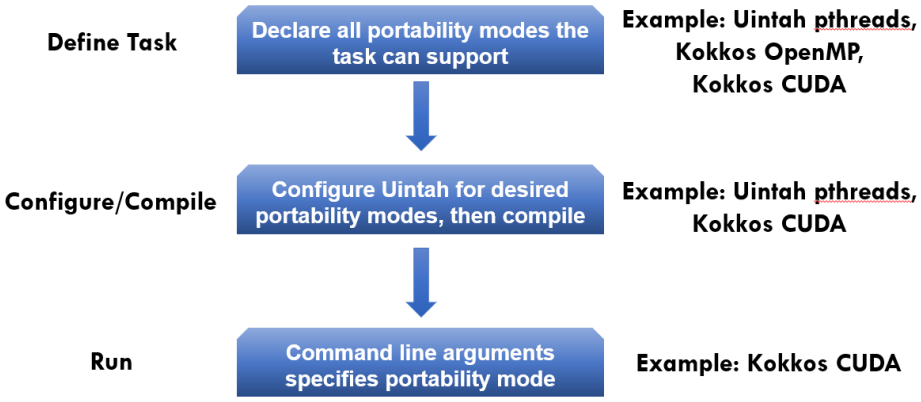

Figure 5: Tasks can be defined for many backends, compiled for a subset of these, and executed on only one of these.