# Kokkos CUDA Modifications

Brad Peterson

## 1 Kokkos's Implementation

The current Kokkos CUDA implementation invokes numerous blocking calls which prevents asynchrony. Blocking calls occur at 1) invoking the CUDA kernel, 2) memory copies to and from GPU memory, and sometimes 3) copying functor parameter data into GPU memory and 4) after a reduction. For asynchrony to work in Kokkos, all these blocking operations must be removed, and CUDA streams should additionally be supported.

## 2 High-Level Goal

The high-level goal of Kokkos CUDA asynchrony is utilizing `Kokkos::CUDA` instances to encapsulate streams. Encapsulating streams comes in two forms. The first is down below:

```
Kokkos::Cuda myInstance("Some instance description");
Kokkos::RangePolicy myRange(myInstance 0, 100);
Kokkos::parallel_for(myRange, DemonstrationFunctor());
```

In this example, a CUDA stream is created by Kokkos and exists inside the `myInstance` object. The above process is termed **managed streams**, as they are similar in nature to Kokkos managed views. Alternatively, an **unmanaged stream** option exists:

```
cudaStream_t myStream;
cudaStreamCreate(&myStream);
Kokkos::Cuda myInstance(myStream);
Kokkos::RangePolicy myRange(myInstance 0, 100);
Kokkos::parallel_for(myRange, DemonstrationFunctor());
```

For asynchrony to work best, a stream should persist beyond the local scope of code. In the prior examples, the application developer should retain the `Kokkos::Cuda` instances for managed streams, or the raw stream for the unmanaged approach. Otherwise, the user has no way to determine if a stream completed.

## 2.1 Managed Streamed Usage

I first modified `Kokkos::Cuda` instance objects to create a CUDA stream in its constructor. Further, the instance objects now have internal reference counting. Suppose a `Kokkos::Cuda` instance object is copied, then both objects share the same stream, and the stream will not be reclaimed until both run the object's destructor code. Overall, the application developer doesn't need to provide any explicit CUDA code, and thus retains Kokkos's portability theme.

The application developer still requires asynchronous interaction to know when a parallel loop has completed. An undesirable option is to employ `Kokkos::fence()`, which in turn signals `cudaDeviceSynchronize()` to create a synchronization point. Another undesirable (and non-portable) option is exposing the stream to the application developer, and require using CUDA API to check the stream.

The implemented asynchrony interaction is an instance object `getStatus()` method that returns a new Kokkos C++ enum. The `getStatus()` performs the CUDA stream queries, then returns whether items on the stream have completed, not completed, or errored. In this manner, the application developer can utilize full asynchrony without interacting with any CUDA API.

## 2.2 Unmanaged Streams

As shown in the High-Level Goal, the application developer can manually create a stream, then pass that stream into a `Kokkos::Cuda` instance object's constructor. The application developer manages the stream allocation and deallocation, and is responsible for ensuring the stream persists beyond the local block of code. The application developer can now use the CUDA stream directly to query the stream's status. Uintah itself has adopted this route to seamlessly fit into its GPU/heterogeneous task scheduler.

## 2.3 Kokkos Parallel Modifications

My work modified Kokkos's `parallel_for` and `parallel_reduce` to support these instances. Additionally, the `RangePolicy` and `TeamPolicy` execution policies also support these changes. All these have been utilized thoroughly in Uintah.

## 2.4 Kokkos Deep Copies

Kokkos employs a `deep_copy()` API to transfer data from one memory space into another, such as from host memory into GPU memory. The Kokkos team previously implemented an overloaded method of `deep_copy()` which accepts a `Kokkos::Cuda` instance object as an argument, and correctly invoked an CUDA asynchronous memory copy call. This work now supplies the encapsulated stream to the CUDA memory copy call. Now application developers can use Kokkos to asynchronously utilize `deep_copy()`. If no stream is supplied, the default blocking CUDA memory copy is used.

# 3 Functors via CUDA Constant Cache

All Nvidia GPUs have 64 KB of constant cache memory shared throughout the device. These act effectively as read only registers, where an instruction can retrieve data in the same clock cycle, provided all CUDA threads in the warp access the same data. Nvidia GPU uses this space for kernel parameter data. An executed CUDA kernel will 1) have its parameter data automatically copied here prior, and then 2) the kernel executes. Likewise, functors (and lambdas) are processed on the GPU using the same two-step process.

## 3.1 Manually Placed Functors

Kokkos has an interesting implementation that duplicates CUDA's existing process. Normally CUDA automatically copies parameters 4KB and smaller into constant cache, anything larger is rejected. In past conversation with Christian Trott, he indicated they used a 512 byte threshold. Anything smaller uses CUDA's automatic copy approach, otherwise, Kokkos first manually copies the functor parameter data, then manually executes the functor. Christian said in my meeting down at Sandia that extensive testing found this to be optimal.

### 3.1.1 My Kokkos Functor Call Modifications

I updated Kokkos to allow multiple CPU threads to asynchronously copy functor parameter data into constant cache memory for anything sized 512 bytes or more. This process now uses an atomic bitset in Kokkos, with each bit representing a chunk of that memory space. When Kokkos prepares to execute a parallel loop, Kokkos now first attempts to atomically enough bits representing a large enough region of constant cache space (starting with the first bit to avoid a Dining Philosophers lock). If successful, then Kokkos can perform the copy into that constant cache memory space using the CUDA stream. Immediately after, the functor is executed on the same stream, with the offset provided to the correct constant cache memory. Finally, a CUDA event is then set immediately after the CUDA copy call, and these events are stored globally within Kokkos. (I tested a few other approaches to detect functor completion, this was by far the most efficient.)

When no constant cache memory slots are available, Kokkos now checks those events. Any event reached means the functor is done, so those accompanying bits are unset. Now new functors can be placed into memory for more work.

### 3.1.2 Reflections

Overall, I think this process is somewhat over-engineered. Christian said they preferred a 512 byte threshold, but my testing didn't support that. CUDA has latency invoking a memory copy, and a second latency invoking a kernel. I found that CUDA's automatic copy process is simply better as its latency is smaller (though I only tested older GPUs, not the latest generation).

Kokkos's approach is optimal when functors have more than 4 KB of parameter data. Uintah's Char-Oxidation task is almost approaching that 4 KB barrier. Kokkos's manual copy approach does allow for larger functors beyond what CUDA would normally allow, and my work allows for several of these to execute simultaneously.

Additionally, I think Kokkos's code here is just old and needs a fresh look. The code itself frequently refers to "local" memory, but local memory is not being used. I suspect this Kokkos CUDA implementation was made back in the very early days of Kokkos.

# 4 Future Idea and Plans

## 4.1 Clean Up

Still needed are implementations for `parallel_scan` and for `MDRangePolicy`. This fix should be easy and would just be patterned on what's been done.

## 4.2 Non-portable Kokkos Execution Instances

Currently, an application developer must manually create a `Kokkos::Cuda` instance to utilize an encapsulated stream. ***This process is not portable***. In my opinion, streams should be automatically turned on and every CUDA call utilizes a pool of streams, and a compile time option available to disable it.

## 4.3 Pinned Memory Pools

As demonstrated in my thesis, data copies GPU-to-host *act as a synchronization barrier, unless pinned memory is used.* For `parallel_reduce`, this is especially a problem, as Kokkos copies the reduction result automatically GPU-to-host. In other words, *all the benefits of this work would normally disappear in reduction computations*, unless the application developer is aware of this problem and acts accordingly.

The application developer could supply his or her own CUDA pinned memory, but that is definitely a non-portable solution. Ultimately, I believe Kokkos should assist the application developer and provide a host pinned memory pool. Thus, a developer using `parallel_reduce` would obtain all the benefits of overlapping computation with communication on the GPU without blocking calls.

## 4.4 Multidemensional Team Policy

Kokkos has an `MDPolicy` to help multidimensional loops iterate in a data-aware manner for the given architecture (row major versus column major). Kokkos also has a `TeamPolicy` that exposes the number of CUDA blocks and threads per block, but it only does so for one dimensional loops.

For Uintah, we needed a multidimensional team policy. We needed control over CUDA blocks and threads per block, and we also needed 3D iterating

data-aware patterns. Because one didn't exist, we wrote our own wrapper for `parallel_for` and `parallel_reduce`. These performed our own data-aware 1D to 3D index mapping, which would in turn invoke a `TeamPolicy` for Kokkos's `parallel_for` and `parallel_reduce`.

A `MDTeamPolicy` implementation would be likely be a large undertaking.