SAND2012-2138P

ArrayHandle Part Deux

From Daxtoolkit

The ArrayHandle object is an important class for managing data between the client (user) code and Dax as well as managing data between control (host) and execution (device) data spaces. However, the current design of ArrayHandle has multiple design flaws that are subverting in many ways including

- The ArrayHandle forces control arrays and execution arrays be allocated in separate memory spaces. For GPUs that is generally the case but for multicore CPUs this is folly.
- The ArrayHandle provides no mechanism for Dax to manage arrays in the control environment. All arrays must be allocated by the client code, which is really problematic when the correct array size is not known in advance.

In this document we first outline the requirements of ArrayHandle as seen today and then provide a new design for the class.

Contents

- 1 ArrayHandle Use Cases
  - 1.1 Input Data
  - 1.2 Intermediate Data
  - 1.3 Result Data
- 2 ArrayHandle Redesign
  - 2.1 ArrayHandle
  - 2.2 ArrayManagerControl
  - 2.3 Allocator
  - 2.4 ArrayManagerExecution
- 3 Use in the Control Environment
- 4 Acknowledgements

ArrayHandle Use Cases

Generally speaking, the ArrayHandle class is tasked with managing arrays of data that are used as inputs and outputs to worklets. Typically these are field data but might also be things like topology connections or indices. More specifically, the ArrayHandle falls under one of the following usages: input data from the client, intermediate data, and result data.

In all of these cases, the ArrayHandle class needs to work efficiently under two operating conditions. The first is the situation where the control and execution environments operate in two completely

independent memory spaces as is common in GPUs today. The second is the situation where the control and execution environments share the same memory space, as is the case for multicore CPUs. Although it is possible to treat the latter like the former (which is what we do today), this is not an efficient solution.

Input Data

For input data, the client code first provides the ArrayHandle with an array filled with data. Ideally, the client code can provide generic iterators to the beginning and end of its array (or array-like structure), as is the case now.

When the control/execution environments are in different memory spaces, the ArrayHandle must allocate and manage an array for the execution environment. Before the array is used in a worklet invocation, the data must be copied to this execution array. In the simple case of a straight copy, the ArrayHandle should have the option of keeping the execution array around in case it is used in a different invocation. (It might also be useful to allow the client to unallocated the array. It might also be useful to automatically unallocated the array if a different allocation fails.)

It might also be useful to stream data to the execution environment. That is, copy a partial array and execute only on that, then move to the next piece. A fancy version could even overlap the copying and execution by pipelining them. Streaming might require the data to be modified in some way. For example, point fields reorganized and duplicated by cells. When streaming, the execution array need not be saved. All that said, streaming is an advanced feature that need not be implemented right away.

When the control/execution environments share memory, the management of input data is significantly different but simpler. In this case, neither a separate allocation nor a memory copy needs to be done. However, there does need to be a special mechanism that allows execution environment objects to access these generic iterators. To make that work, we'll need to add DeviceAdapter template parameters to lots of execution environment objects like work and field so that they know how to access the memory. Even so, we will have to force the code to work through runtime polymorphism to differentiate between client supplied input data and Dax generated intermediate and result data, or we'll have to force both to be the same.

Intermediate Data

An ArrayHandle used to hold intermediate data must be allocated by Dax to store the results of some operation in the execution environment, but the data need never be returned to the user.

When the control/execution environments are in different memory spaces, the ArrayHandle must allocate and manage an array for the execution environment as was the case for input data. However, no control environment array is ever needed. Thus, a lot of the control logic goes away and no copies are needed. In some cases, it does not make sense to use an intermediate data ArrayHandle (as defined here) with streaming as there is nothing to store the full collection of intermediate values.

Conceptually, there is no change when the control/execution environments share memory. The same single array is created. However, the shared memory instance also shares everything with the result data

case, and as there is no other instance where the shared memory needs an execution-specific array, the implementation would probably be easier if it shared that implementation.

Result Data

An ArrayHandle managing result data, that is data created by worklet execution and eventually passed back to the client code, has much in common with input data, especially when the control/execution environments are in different memory spaces. For both input and result data there are arrays for control and execution environments and data must be copied between them. One main difference is that data is copied from execution to control instead of the other way around. After that, however, the result data array could be used as an input data array.

Another important feature of result data regardless of whether control/execution environments share memory is that Dax needs to allocate an array for the control environment that eventually gets passed back to the client code. This is important as there are instances when the proper size of the array will not be known until the middle of a worklet-scheduling function. Even when the proper size is known beforehand, allowing client code to allocate the array may be a mistake. Simply allocating an array with `std::vector` can destroy memory affinity on a multicore machine by initializing all the values from a single thread.

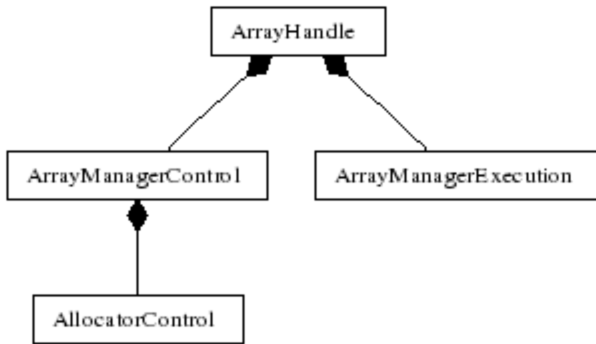
ArrayHandle Redesign

The following is a proposal for a redesign of ArrayHandle. As before, there should be a single class, which we still call ArrayHandle, responsible for managing arrays and their data. Also as before, ArrayHandle assumes that there are (or at least could be) separate arrays for the control and execution environments. These two separate arrays are managed by two delegate classes: ArrayManagerControl and ArrayManagerExecution.

The ArrayManagerControl class is a fairly straightforward class that can allocate and deallocate CPU arrays as well as return iterators to the arrays. The ArrayManagerControl is templated on a Dax version of an allocator. The allocator is similar to an STL allocator except that it also defines and returns an iterator instead of a pointer. This allows the allocator to encapsulate the memory layout like a container without having to actually contain anything.

The ArrayManagerExecution is defined by the DeviceAdapter. If the device has a separate memory space for the execution environment, then the ArrayManagerExecution is responsible for copying data back and forth. If the device shares memory, then it basically delegates all memory management to the ArrayExecutionControl.

Here is the basic relationship between these classes.



We now dive down into the details of how these classes can be structured.

ArrayHandle

The ArrayHandle class is the facade that organizes the access to all these design components.

```
template <typename ValueT,
          template <typename> class AllocatorT,
          class DeviceAdapter>
class ArrayHandle
{
private:
  typedef ArrayManagerControl<ValueType, AllocatorT> ArrayManagerControlType;
  typedef typename DeviceAdapter::template ArrayManagerExecution<ValueType, AllocatorT
      ArrayManagerExecutionType;
public:
  typedef ValueT ValueType;
  typedef AllocatorT<ValueType> AllocatorType;
  typedef typename AllocatorType::IteratorType IteratorControlType;
  typedef typename ArrayManagerExecutionType::IteratorType IteratorExecutionType;

  IteratorControlType GetIteratorBegin();
  IteratorControlType GetIteratorEnd();

  void ReleaseResourcesExecution();
  void ReleaseResourcesControl();

  IteratorExecutionType PrepareForInput();
  IteratorExecutionType PrepareForOutput(dax::Id numberOfValues);

  ArrayManagerControl<ValueType, AllocatorT> TakeArrayManagerControl();
private:
  ArrayManagerControlType ControlArray;
  ArrayManagerExecutionType ExecutionArray;
};
```

There's probably way more detail here than we need, but let's go over the features independently. One detail I've left out is how an ArrayHandle is constructed. There will be two ways to create an ArrayHandle. The first is to provide begin and end iterators that are the same as those defined by the allocator. The second is to create an empty ArrayHandle that will be used later to store the output of some operations.

GetIteratorBegin/End
: Provides an easy mechanism to access the data managed by this ArrayHandle (generally used in the control environment). If necessary, copies data from the execution to the control array, and then returns the iterator to the control array.

ReleaseResourcesExecution/Control
: Frees up any allocated memory in the respective environment.

PrepareForInput
: This method is called before the managed array is used in the execution environment as a source of data. If necessary, copies data from the control environment to the execution environment and marks the two arrays as valid. If neither array has valid data, this method may throw an exception. Returns an iterator that can be used in the execution environment.

PrepareForOutput
: This method is called before the managed array is used in the execution environment as a destination of data. If necessary, allocates memory for the execution array (which, if memory is shared, could also allocate for the control array). Marks the execution array as valid and the control array as invalid with the assumption that the calling scope will soon fill the execution array. Returns an iterator that can be used in the execution environment.

TakeArrayManagerControl
: This method provides a means for client code to assume management over the control environment array. It provides the class that allows the client code to embed the array in its own class that manages the array. (A vtkDataArray might be an example of one such class.) After this call the control array is reset to NULL so that the ArrayHandle will no longer modify or, more importantly, delete the array.

  We probably won't have a real use case for this method until we couple Dax with some other system like VTK. We might want to hold off on implementing this part. --Kenneth Moreland 21:29, 13 March 2012 (EDT)

> So will the new array do away with the need to call CompleteAsOutput after using an array? This was extremely annoying when using temporary or execution generated arrays as input for another device adapter call. Also I disliked the way that using arrays for output than input required copying the contents to the control environment when the user hasn't explicitly asked for that.
>
> Secondly can you show how to copy the data from an ArrayHandle that doesn't have control iterators to the control environment? --Robert Maynard 08:58, 14 March 2012 (EDT)

ArrayManagerControl

The ArrayManagerControl class assumes a type and allocator and manages the creation and deletion of an array. In sort, it is a very simple container. The allocator is discussed in the following section.

```
template <typename ValueT, template <typename> class AllocatorT>
class ArrayManagerControl
{
public:
  typedef ValueT ValueType;
  typedef AllocatorT<ValueType> AllocatorType;
  typedef typename AllocatorType::IteratorType IteratorType;

  IteratorType GetIteratorBegin() const;
  IteratorType GetIteratorEnd() const;
  dax::Id GetNumberOfValues() const;  // Or GetNumberOfItems?

  void Allocate(dax::Id numberOfValues);  // Can throw
  void ReleaseResources();

  IteratorType TakeReference();
};
```

Most of the typedefs and methods are straightforward, so I won't bother explaining them.

The TakeReference method, like ArrayHandle::TakeArrayManagerControl, is provided to allow client code to assume management of the array.

> Once again, we might want to hold off on implementing this feature. --Kenneth Moreland 21:29, 13 March 2012 (EDT)

Allocator

The allocator is similar to an STL allocator except that it also defines and returns an iterator instead of a pointer. This allows the allocator to encapsulate the memory layout like a container without having to actually contain anything.

The allocator is a template parameter through the rest of the classes principally so that client code can adapt the allocation and data layout to its own internal structures. To make things easier, we should use the same method employed in DeviceAdapter to select a default allocate to use in all instances.

Here is what a basic array allocator would look like.

```
template <typename T>
class AllocatorControlBasic
{
public:
  typedef T ValueType;
  typedef ValueType *IteratorType;
  IteratorType Allocate(dax::Id numberOfValues); // No constructors called!
  void Deallocate(IteratorType pointer); // No destructors called!
};
```

Once again, these members are straightforward. One important note is that the allocator never constructs nor destructs the classes it allocates. This is intentional to prevent problems with memory affinity if the constructor or destructor touches the actual memory. This lack of constructor/destructor should be fine

so long as we are only allocating basic types (e.g. float, int) or basic structures with no resource management (e.g. Vector3). We may want to add some traits/template magic to ensure that the allocator is only used with these simple types.

ArrayManagerExecution

The ArrayManagerExecution, which is really defined as a subclass of the DeviceAdapter, is a strange beast because sometimes there exists a separate execution array, and sometimes the control array is reused for execution. I expect there to be two different implementations of ArrayManagerExecution in the dax::cont::internal namespace, one for each case, and that most DeviceAdapters will just use one internally.

At any rate, the two types have relatively similar interfaces that should look something like this.

```
struct DeviceAdapter
{
  …
  template <typename T, template <typename> class AllocatorT>
  class ArrayManagerExecution
  {
  public:
    typedef T ValueType;
    typedef AllocatorT<T> AllocatorType;
    typedef ValueType *IteratorType;  // AllocatorType::IteratorType also common.

    void LoadDataForInput(
        ArrayManagerControl<ValueType,AllocatorType> controlArray);

    void AllocateArrayForOutput(
        ArrayManagerControl<ValueType,AllocatorType> controlArray,
        dax::Id numberOfValues);

    void RetrieveOutputData(
        ArrayManagerControl<ValueType,AllocatorType> controlArray);

    IteratorType GetIteratorBegin(
        ArrayManagerControl<ValueType,AllocatorType> controlArray) const;

    void ReleaseResources();
  };
```

Now, the behavior of these methods change significantly depending on whether the DeviceAdapter has separate or shared memory spaces. The following table defines what each method does in each of these conditions.

Method	Separate Memory Spaces	Shared Memory Spaces
LoadDataForInput	Allocates an array on the device and copies the given controlArray into it.	No operation.
AllocateArrayForOutput	Allocates an array on the device with the given number of values. controlArray is ignored.	Allocates an array in controlArray.

RetrieveOutputData	Allocates controlArray and copies data from the device to it.	No operation.
GetIteratorBegin	Returns the begin iterator for the device array. controlArray is ignored.	Returns the iterator for controlArray.
ReleaseResources	Frees the device array.	No operation.

Use in the Control Environment

One major change this makes for the Dax system is that the execution environment must now deal with more than one type array in which field values might be stored. The field might be either a fixed execution array type or whatever the client specified allocator provides.

To deal with this, we must introduce another template parameter to worklets and most classes that the worklets use. We will call this template parameter ExecutionAdapter (alternate names are welcome). The ExecutionAdapter is specified by the DeviceAdapter and initially would look something like this.

```
struct DeviceAdapter
{
  ...
  template<template <typename> class AllocatorT>
  struct ExecutionAdapter
  {
    template <typename T> class FieldIteratorType;
  };
```

This class will implicitly be set in a template parameter of a worklet, which is defined something like this.

```
template<class CellType, class FieldType, class ExecutionAdapter>
DAX_WORKLET void Square(
    DAX_IN dax::exec::WorkMapField<CellType, ExecutionAdapter> &work,
    DAX_IN const dax::exec::Field<FieldType, ExecutionAdapter> &inField,
    DAX_OUT dax::exec::Field<FieldType, ExecutionAdapter> &outField)
{
```

The class in dax::exec can use the ExecutionAdapter class to determine how to store and access arrays.

> Shouldn't we just have to make new types of DataArrays and make sure they have the same API and than template all execution operations on the type of DataArray it is using. That way the user code isn't complicated more? --Robert Maynard 09:01, 14 March 2012 (EDT)

Acknowledgements

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia
National
Laboratories