# SANDIA REPORT

SAND2014-0047
Unlimited Release
Printed January 2014

# A Pervasive Parallel Framework for Visualization: Final Report for FWP 10-014707

Kenneth Moreland

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

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

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.



Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2014-0047
Unlimited Release
Printed January 2014

# A Pervasive Parallel Framework for Visualization: Final Report for FWP 10-014707

Kenneth Moreland

**Abstract**

We are on the threshold of a transformative change in the basic architecture of high-performance computing. The use of accelerator processors, characterized by large core counts, shared but asymmetrical memory, and heavy thread loading, is quickly becoming the norm in high performance computing. These accelerators represent significant challenges in updating our existing base of software. An intrinsic problem with this transition is a fundamental programming shift from message passing processes to much more fine thread scheduling with memory sharing. Another problem is the lack of stability in accelerator implementation; processor and compiler technology is currently changing rapidly. This report documents the results of our three-year ASCR project to address these challenges. Our project includes the development of the Dax toolkit, which contains the beginnings of new algorithms for a new generation of computers and the underlying infrastructure to rapidly prototype and build further algorithms as necessary.

# Acknowledgement

Thanks to everyone who contributed to this project and helped make it a success. Thanks to Kwan-Liu Ma from the University of California at Davis for the encouragement to start the project and subsequent support. Thanks to Berk Geveci and Utkarsh Ayachit from Kitware, Inc. for managing so many of the management and technical details of the project. Thanks to Robert Maynard from Kitware, Inc. for implementing the larger moiety of the code, and thanks to Brad King for implementing the impossibly cool meta template programs. Thanks to Robert Miller from the University of California at Davis for providing the foundational research that make connectivity possible. Finally, thanks to everyone in the SciDAC Scientific Data Management, Analysis, and Visualization Institute for recognizing Dax as a DOE solution for scientific visualization on advanced architectures.

This work was supported in whole by the DOE Office of Science, Advanced Scientific Computing Research, under award number 10-014707, program manager Lucy Nowell.

Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration.

# Contents

# List of Figures

# List of Examples

# Executive Summary

The evolution of the computing world from teraflop to petaflop has been relatively effortless, with several of the existing programming models scaling effectively to the petascale. The migration to exascale, however, poses considerable challenges. All industry trends infer that the exascale machine will be built using processors containing hundreds to thousands of cores per chip. It can be inferred that efficient concurrency on exascale machines requires a massive amount of concurrent threads, each performing many operations on a localized piece of data.

Currently, visualization libraries and applications are based off what is known as the visualization pipeline. In the pipeline model, algorithms are encapsulated as *filters* with inputs and outputs. These filters are connected by setting the output of one component to the input of another. Parallelism in the visualization pipeline is achieved by replicating the pipeline for each processing thread. This works well for today's distributed memory parallel computers but cannot be sustained when operating on processors with thousands of cores.

Our project investigates a new visualization framework designed to exhibit the pervasive parallelism necessary for extreme scale machines. Our framework achieves this by defining algorithms in terms of *worklets*, which are localized stateless operations. Worklets are atomic operations that execute when invoked unlike filters, which execute when a pipeline request occurs. The worklet design allows execution on a massive amount of lightweight threads with minimal overhead. Only with such fine-grained parallelism can we hope to fill the billions of threads we expect will be necessary for efficient computation on an exascale machine.

## Progress and Accomplishments

Although the "Pervasive Parallel Processing Framework for Data Visualization and Analysis at Extreme Scale" project is a research project to make progress on designing and implementing massively threaded visualization algorithms, our project also aims to explore techniques that simplify the development of such algorithms and to provide useful software for this purpose. To that end we have developed the Dax toolkit as a deployment platform for our research. The Dax toolkit is a comprehensive C++ header library that embodies the techniques developed within the project. A summary of our major accomplishments is as follows.

**Development of Framework** Much thought has gone into the design of the core components of the toolkit API that users will use to define their analysis algorithms. The API

has been developed to be succinct, type safe, and efficient when executing on multiple architectures. We developed adapters to enable execution and testing of the framework on multiple architectures including GPUs and CPUs. We added support for advanced core data structures needed for data analysis such as data set types, cell types, and structures for sorting geometrical and topological information. The framework supports advanced analysis algorithms including those that change both geometry and topology, such as marching cubes and threshold.

We reevaluated the use of pipelines for analysis on massively parallel architectures. The idea being that the framework would control the flow of execution and perform a limited sort of kernel fusion to maximize the amount of computation per data load. We concluded that the complexity in API and the toolkit implementation due to scheduling and execution of pipelines could be greatly simplified by abandoning the connected pipeline paradigm. Instead, the users directly manage the data flow by dispatching calls to analysis worklets in order. We find that the framework structure lends to developing in such a way as to encourage performing as many operations per data load as possible without further adjustment by the framework.

**Cross-Platform Analysis** Our implementation now supports multiple target platforms including GPU (using CUDA [16] via Thrust [3]), multi-core CPUs (using either OpenMP [5] via Thrust or TBB [14]), and single core CPU. We achieve these cross-platform implementations through careful structure of the toolkit code. We have identified the basic features specific to each multi-threaded device and encapsulated them in a unit called a *device adapter*. A device adapter can be implemented by providing only a thread scheduling mechanism although more efficient custom algorithms can be provided as well. A device adapter can be changed with a single template parameter, thus enabling the porting of the majority of code with very little development.

**Development of Analysis Algorithms** We developed infrastructure to support analysis algorithms including those that change topology and geometry. These algorithms require multiple passes, particularly on architectures with memory restrictions, such as GPUs and potentially proposed exascale machines. With the framework API and infrastructure matured we developed key analysis algorithms that also exercise the framework. These include thresholding of cells using point fields, marching cubes for generating isosurfaces, and computing derived fields.

**Software Infrastructure** For software reliability and correctness, we set up a software process including a testing framework, dashboards for daily regression testing and verification, a developer wiki for design and implementation discussions, Doxygen for API documentation, and mailing list for developer communication.

# Presentations and Publications

During the course of our project, we presented our work to a broad audience to gain feedback and discuss the vision we have for parallelizing visualization algorithms. Some of the major locations where we presented the Dax toolkit are:

**Dax: Data Analysis at Extreme,** paper by Kenneth Moreland, Utkarsh Ayachit, Berk Geveci, and Kwan-Liu Ma. In Proceedings of SciDAC 2011, July 2011.

**Dax Toolkit: A Proposed Framework for Data Analysis and Visualization at Extreme Scale,** paper by Kenneth Moreland, Utkarsh Ayachit, Berk Geveci, and Kwan-Liu Ma. In IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV), October 2011.

**Next-Generation Capabilities for Large-Scale Scientific Visualization,** presentation by Kenneth Moreland. 15th SIAN Conference on Parallel Processing for Scientific Computing, February 2012.

**Next-Generation Codes/Portability: Dax Perspective,** presentation by Kenneth Moreland, DOECGF, April 2012.

**Oh, $#*@! Exascale! The Effect of Emerging Architectures on Scientific Discovery,** paper by Kenneth Moreland. In 2012 SC Companion (Proceedings of the Ultrascale Visualization Workshop), November 2012, pg. 224-231. DOI 10.1109/SC.Companion.2012.38.

**Dax for Multi- and Many-Core Architectures,** panel presentation by Kenneth Moreland. Supercomputing, November 2012.

**The SDAV Software Frameworks for Visualization and Analysis on Next-Generation Multi-Core and Many-Core Architectures,** paper by Christopher Sewell, Jeremy Meredith, Kenneth Moreland, Tom Peterka, Dave DeMarle, La-ta Lo, James Ahrens, Robert Maynard, and Berk Geveci. In 2012 SC Companion (Proceedings of the Ultrascale Visualization Workshop), November 2012, pg. 206-214. DOI 10.1109/SC.Companion.2012.36.

**Flexible Analysis Software for Emerging Architectures,** paper by Kenneth Moreland, Brad King, Robert Maynard, and Kwan-Liu Ma. In 2012 SC Companion (Proceedings of Petascale Data Analytics: Challenges and Opportunities), November 2012. DOI 10.1109/SC.Companion.2012.115.

**Optimizing Threshold for Extreme Scale Analysis,** poster by Robert Maynard, Kenneth Moreland, Utkarsh Ayachit, Berk Geveci, and Kwan-Liu Ma. In Proceedings of SPIE Visualization and Data Analysis, February 2013.

**A Survey of Visualization Pipelines,** paper by Kenneth Moreland. IEEE Transactions on Visualization and Computer Graphics, 19(3), March 2013. DOI 10.1109/TVCG.2012.133.

**Dax Toolkit: Efficient Visualization at Extreme Scale,** presentation by Robert Maynard, GPU Technology Conference, March 2013.

**The effect of emerging architectures on data analysis software,** panel presentation by Kenneth Moreland, SOS 17, March 2013.

**Dax,** presentation by Kenneth Moreland, DOECGF, April 2013.

**Research Challenges for Visualization Software,** paper by Hank Childs, Berk Geveci, Will Schroeder, Jeremy Meredith, Kenneth Moreland, Christopher Sewell, Torsten Kuhlen, and E. Wes Bethel. IEEE Computer, 46(5), May 2013, pg. 34-42. DOI 10.1109/MC.2013.179.

**Upcoming Challenges for Scientific Visualization Software: Programming Future Architectures,** panel presentation by Kenneth Moreland, IEEE Visualization, October 2013.

**A Classification of Scientific Visualization Algorithms for Massive Threading,** paper by Kenneth Moreland, Berk Geveci, Kwan-Liu Ma, and Robert Maynard. In Proceedings of the Ultrascale Visualization Workshop, November 2013. DOI 10.1145/2535571.2535591.

# Chapter 1

# Introduction

High-performance computing relies on ever finer threading. Recent advances in processor technology include greater numbers of cores, hyperthreading, and accelerators with integrated blocks of cores, all of which require more software parallelism to achieve peak performance. Current visualization solutions cannot support this extreme level of concurrency. Extreme scale systems require a new programming model and a fundamental change in how we design algorithms. This document is a report on the project titled "A Pervasive Parallel Processing Framework for Data Visualization and Analysis at Extreme Scale" funded by the ASCR Scientific Data Management and Analysis at Extreme Scale program. This project delivers its work with the creation of the Data Analysis at Extreme (Dax) toolkit.

The Dax toolkit supports a number of algorithms and the ability to design further algorithms through a top-down design with an emphasis on extreme parallelism. Dax also provides support for finding and building links across topologies, making it possible to perform operations that determine manifold surfaces, interpolate generated values, and find adjacencies. Although Dax provides a simplified high-level interface for programming, its template-based code removes the overhead of abstraction.

The Dax toolkit simplifies the development of parallel visualization algorithms. Consider the code samples in Figure 1.1 that come from the Visualization Toolkit (VTK) on the left and our Dax toolkit on the right. Both implementations perform the same operation; they estimate gradients using finite differences. Both toolkits provide similar classes and functions, and consequently the code looks remarkably similar.

However, because the Dax toolkit is structured such that it can schedule its execution on a GPU, we measure that it performs this operation over 100 times faster than the VTK code running on a single CPU. Furthermore, the Dax API can be switched to a different device by changing only a single line of code. Dax currently provides scheduling for CUDA (GPU), OpenMP (multi-core CPU), Intel Threading Building Blocks (multi-core CPU), and serial execution.

This report documents the design of the Dax toolkit. Chapter 2 provides an overview of the Dax toolkit and describes the higher level context. Chapter 3 describes the API of the Dax toolkit and provides the initial software documentation. Chapter 4 reports on further lessons and achievements attained during this project.

```
int vtkCellDerivatives::RequestData(...)
{
  ...[allocate output arrays]...
  ...[validate inputs]...



  for (cellId=0;
       cellId < numCells;
       cellId++)
    {
    ...[update progress]...
    input->GetCell(cellId, cell);
    inScalars->GetTuples(cell->PointIds,
                         cellScalars);
    scalars = cellScalars->GetPointer(0);

    subId =
      cell->GetParametricCenter(pcoords);


    cell->Derivatives(
      subId, pcoords, scalars, 1, derivs);

    outGradients->SetTuple(cellId,derivs);

    }
  ...[cleanup]...
}
```

VTK Code

```
struct CellGradient
  : public dax::exec::WorkletMapCell
{
  typedef void ControlSignature(
      Topology, Field(Point),
      Field(Point), Field(Out));
  typedef _4 ExecutionSignature(_1,_2,_3);

  template<class CellTag>
  DAX_EXEC_EXPORT
  dax::Vector3 operator()(...)
  {



    dax::Vector3 parametricCellCenter =
        dax::exec::ParametricCoordinates<
          CellTag>::Center();

    return dax::exec::CellDerivative(
        parametricCellCenter,
        coords,
        pointField,
        cellTag);
  }

};
```

Dax Code

Figure 1.1: A comparison of code to compute a localized field derivative within VTK and within the Dax toolkit.

# Chapter 2

# Overview of Dax Toolkit

The Dax toolkit is built to develop a readiness for scientific data analysis and visualization at extreme scale. In particular, we address the challenges of emerging multi- and many-core architectures. To achieve this readiness, our project has three overarching goals.

- Create a toolkit that is well suited to the design of visualization operations with a great number of shared memory threads.
- Develop a framework that adapts to emerging processor and compiler technologies.
- Design multi-purpose algorithms that can be applied to a variety of visualization operations.

This chapter provides the broad design and high-level features of the Dax toolkit that makes these goals a reality.

## 2.1 General Approach

The Dax toolkit is designed to provide a *pervasive parallelism* throughout all its visualization algorithms, meaning that the algorithm is designed to operate with independent concurrency at the finest possible level throughout. The Dax toolkit provides this pervasive parallelism by providing a programming constructs called a *worklet*, which operates on a very fine granularity of data. The worklets are designed as serial components, and the Dax toolkit handles whatever layers of concurrency are necessary, thereby removing the onus from the visualization algorithm developer.

A worklet is essentially a small functor or kernel designed to operate on a small element of data. (The name "worklet" means a small amount of work. We mean small in this sense to be the amount of data, not necessarily the amount of instructions performed.) The worklet is constrained to contain a serial and stateless function. These constraints form three critical purposes. First, the constraints on the worklets allow the Dax toolkit to schedule worklet invocations on a great many independent concurrent threads and thereby making the algorithm pervasively parallel. Second, the constraints allow the Dax toolkit

to provide thread safety. By controlling the memory access the toolkit can insure that no worklet will have any memory collisions, false sharing, or other parallel programming pitfalls. Third, the constraints encourage good programming practices. The worklet model provides a natural approach to visualization algorithm design that also has good general performance characteristics.

This approach mirrors that of Baker et al. [2]. Both approaches use C++ templating to generically apply functors in parallel to vectors of data. Where the Dax toolkit significantly differs from that of Baker's is in that we are more focused on the computational geometry problems related to scientific visualization and data analysis. Where Baker provides a simple mapping mechanism onto a vector, our system is designed to provide a variety of parallel scheduling operations. These result in worklet types that get scheduled in different ways. Each worklet type has a different set of capabilities. The types of worklets and their functions is documented in Section 3.5.1. These, along with customized scheduling operations, provide reusable communicative operations that can be applied to many visualization algorithms.

Worklets also provide additional functionality beyond the typical functor by having flexibility in their call structure. Worklets are self describing in that they provide signatures specifying the type and meaning of input and output arguments. This functionality is described in Section 2.5.

## 2.2 Structure of Dax Framework

The Dax toolkit allows users to design algorithms that are run on massive amounts of threads. These algorithms are embedded in worklets and can be run on a number of devices. However, the Dax toolkit also allows users to interface to applications, define data, and invoke algorithms that they have written or are provided otherwise.

These two modes represent significantly different operations on the data. As explained in Section 2.1, the operating code in a worklet is constrained to access only a small portion of data that is provided by the framework. Conversely, code that is building the data structures needs to manage the data in its entirety, but has no reason to perform computations on any particular element.

Consequently, the Dax toolkit is divided into two *environments* that handle each of these use cases. Each environment has its own API, and direct interaction between the environments is disallowed. The environments are as follows.

**Execution Environment** This is the environment in which worklets are executed. The API for this environment provides work for one element with convenient access to information such as connectivity and neighborhood as needed by typical visualization algorithms. Code for the execution environment is designed to always execute on a very large number of threads.

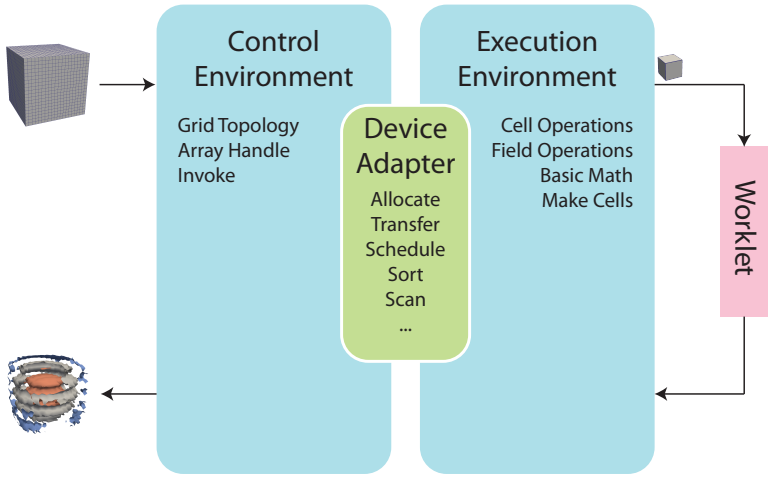



Figure 2.1: Diagram of the Dax framework.

**Control Environment** This is the environment that is used to interface with applications, interface with I/O devices, and schedule parallel execution of the worklets. The associated API is designed for users that want to use the Dax toolkit to analyze their data using provided or supplied worklets. Code for the control environment is designed to run on a single thread (or one single thread per process in an MPI job).

These dual programming environments are partially a convenience to isolate the application from the execution of the worklets and are partially a necessity to support GPU languages with host and device environments. The control and execution environments are logically equivalent to the host and device environments, respectively, in CUDA [16] and other associated GPU languages.

Figure 2.1 displays the relationship between the control and execution environment. They typical workflow when using the Dax toolkit is that first the control thread establishes a data set in the control environment and then invokes a parallel operation on the data using a worklet. From there the data is logically divided into its constituent elements, which are sent to independent invocations of the worklet. The worklet invocations, being independent, are run on as many concurrent threads as are supported by the device. On completion the results of the worklet invocations are collected to a single data structure and a handle is returned back to the control environment.

## 2.3 Device Independence

As multiple vendors vie to provide accelerator-type processors, a great variance in the computer architecture exists, and we expect to encounter further changes in the near future. Likewise, there exist multiple compiler environments and libraries for these devices. The most popular of these include OpenMP, CUDA, and OpenCL (although the latter does not

yet support C++ classes and templates). These compiler technologies also vary from system to system.

To make porting among these systems at all feasible, we require a base language support, and the language we use is C++. The majority of the code in Dax is constrained to the standard C++ language constructs to minimize the specialization from one system to the next.

Each device and device technology requires some level of code specialization, and that specialization is encapsulated in a unit called a *device adapter*. Thus, porting the Dax toolkit to a new architecture can be done by only adding a device adapter.

The device adapter is shown diagrammatically as the connection between the control and execution environments in Figure 2.1. The functionality of the device adapter comprises two main parts: a collection of parallel algorithms run in the execution environment and a module to transfer data between the control and execution environments.

Each device adapter is expected to implement a collection of algorithms containing operations like parallel for, scan, sort, parallel find, stream compact, and unique (remove duplicates). This list of operations is similar to those suggested by Blelloch [4] and Lo et al. [9]. It is also a subset of those operations provided by the Thrust library [3]. Thrust itself provides a convenient implementation for device adapters because it itself is portable among devices. However, the interface to the device adapter algorithms is independent of Thrust, and we have examples of device adapters that can be built without Thrust. In fact, the Dax toolkit contains generic implementations of every needed algorithm that minimally use only a provided parallel for operation. However, it is usually more efficient to provide specialized versions of at least sort and scan.

A device adapter also provides a module to handle the transfer of data between the control and execution environments. Unlike other systems such as CUDA and Thrust, which explicitly define separate arrays and copy between them, the Dax device adapter allocates and copies data in one monolithic operation. The advantage of this approach is that a device adapter for a system that shares memory between the two environments (such as with OpenMP) can perform shallow copies to share the data.

The implementation of device adapters is described in more detail in Sections 3.4.1 and 3.4.7. The use of device adapters can be found throughout Chapter 3.

## 2.4 Generic Memory Structures

The basic data container in the Dax toolkit is an *array handle*. The array handle acts like a smart pointer to the data to manage its resource usage. Array handle objects maintain a reference count of how many instances point to the same array, which allows the array handle to release resources automatically once all references leave scope.

Array handle objects can also allocate and de-allocate data as necessary. For example, when an array handle is used to store the output of an algorithm, Dax will automatically allocate data in the array to store the appropriate amount of data.

Although the array handle interface is available exclusively in the control environment, an array handle object manages data in both the control and execution environment. This is done using the data transfer module of the device adapter. As described in Section 2.3, if the control and execution environments can share memory, then this data is not physically copied but rather shared. The array handle also maintains where data resides to avoid unnecessary copies. That is, if data is needed in the execution environment and is already available in the execution environment, no copy will be made. To help applications manage limited memory, the array handle allows applications to free memory either in the execution environment or both environments.

In addition to adapting to various device memory spaces with the device adapter, the array handle can also adapt to memory layout in the control application. This is an important technique when applying Dax algorithms to data defined in other library spaces. For example, some systems may define an array of coordinates as a single array with each entry containing 3 coordinates (an array of structures) whereas another might define the same data with three arrays, each containing a single coordinate (a structure of arrays). Rather than copy this data tot some canonical structure, the array handle uses generic access to adapt to any layout.

This generic access is achieved through a *container* object. The container provides an encapsulated interface around the data so that any necessary strides or offsets may be handled internally. The relationship between array handles and containers is shown in Figure 2.2.

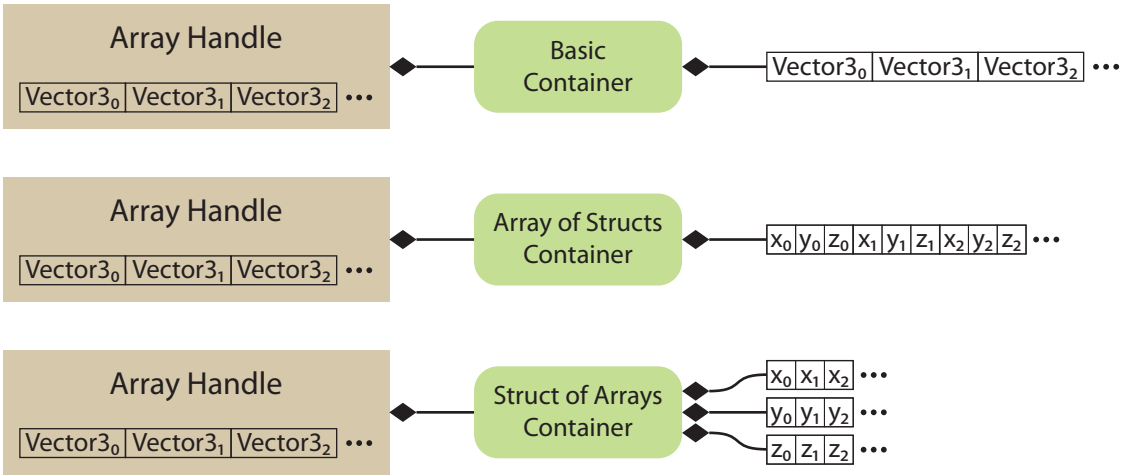



Figure 2.2: Array handles, containers, and the underlying data storage.

One interesting consequence of using a generic container object to manage data within an array handle is that the container can be defined functionally rather than point to data stored in physical memory. Thus, implicit array handles are easily created by adapting to functional containers. For example, the point coordinates of a uniform rectilinear grid are

implicit based on the topological position of the point. Thus, the point coordinates for uniform rectilinear grids can be implemented as an implicit array with the same interface as explicit arrays (where unstructured grid points would be stored).

The implementation and use of array handles and array containers are discussed in Section 3.4.2.

## 2.5 Generic Scheduling

In addition to using the worklet motif to simplify the design of parallel algorithms, the Dax toolkit aims to relieve its users from the details of scheduling work in the execution environment. Although it is straightforward to write a parallel dispatching mechanism for a function with a fixed interface, the Dax worklets are designed to be a free-form expression of an algorithm. This free-form design necessitates a custom scheduler be built for each worklet.

The Dax toolkit uses metatemplate programming to provide this custom scheduling. Each worklet declares its interface through the type of operation it does and a pair of *signatures* that define the semantics of arguments from the control environment and to the execution environment operation. Figure 2.3 shows an example of this declaration, and Section 3.5.1 provides details on how worklets are created.

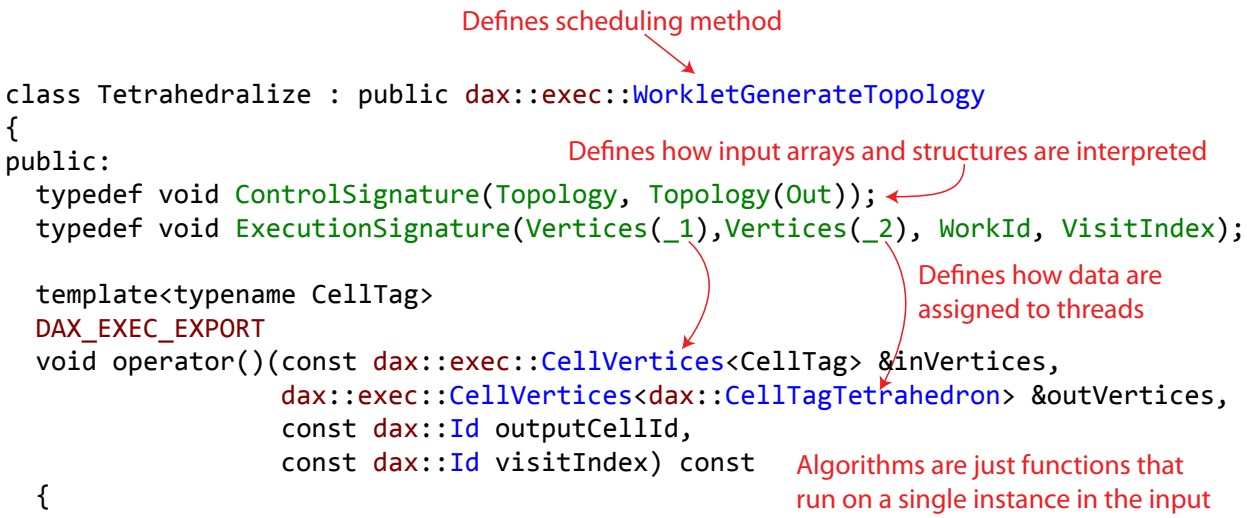



Figure 2.3: An example of a worklet declaration.

Worklets are invoked with an object called a *dispatcher*. The dispatcher uses the signatures provided by a worklet to determine the meaning of input meshes and arrays, transfer the necessary data to the execution environment, launch an appropriate amount of execution threads, and dereference the appropriate data for each invocation of the worklet. Section 3.4.4 describes using the dispatcher classes to invoke worklets.

# Chapter 3

# Dax Toolkit Documentation

This chapter documents the implementation and API of the Dax toolkit. This documentation is primarily in reference to users of the Dax toolkit but also gives some details on the internal implementation.

The Dax toolkit is written in C++ and makes extensive use of templates. The toolkit is implemented as a header library, meaning that all the code is implemented in header files (with extension .h) and completely included in any code that uses it. This is typically necessary of template libraries, which need to be compiled with template parameters that are not known until they are used. This also provides the convenience of allowing the compiler to inline user code for better performance.

When documenting the Dax API, the following conventions are used.

- Filenames are printed in a sans serif font.
- C++ code is printed in a `monospace font`.
- Macros and namespaces from the Dax toolkit are printed in `red`.
- Identifiers from the Dax toolkit are printed in `blue`.
- Signatures, described in Section 2.5, and the tags used in them are printed in `green`.

## 3.1 Package Structure

The Dax toolkit is organized in a hierarchy of nested packages. The Dax toolkit places definitions in *namespaces* that correspond to the package (with the exception that one package may specialized a template defined in a different namespace). Hence, the description and

The base package is named `dax`. All classes within the Dax toolkit are placed either directly in the `dax` package or in a package beneath it. This helps prevent name collisions between the Dax toolkit and any other library.

As described in Section 2.2, the Dax API is divided into two distinct environments: the control environment and the execution environment. The API for these two environments

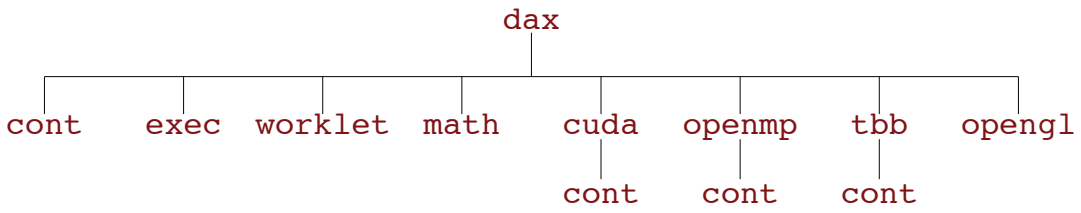



Figure 3.1: Dax package hierarchy.

are located in the dax::cont and dax::exec packages, respectively. Items located in the base dax namespace are available in both environments.

Although it is conventional to spell out names in identifiers (see the coding conventions in Section 3.7), there is an exception to abbreviate control and execution to cont and exec, respectively. This is because it is also part of the coding convention to declare the entire namespace when using an identifier that is part of the corresponding package. The shorter names make the identifiers easier to read, faster to type, and more feasible to pack lines in 80 column displays. These abbreviations are also used instead of more common abbreviations (e.g. ctrl for control) because, as part of actual English words, they are easier to type.

Worklets provided by the Dax toolkit, described in Section 3.3, are contained in the dax::worklet package. Although the operation of a worklet happens exclusively in the execution environment, worklets are typically initialized in the control environment. Thus, the dax::worklet package is not encapsulated in either dax::cont or dax::exec.

The Dax toolkit provides a base set of library functions that are ported to the various systems and compilers on which it is used. These functions are located in the dax::-math package. The features in dax::math are available in both the control and execution environments, but they are typically used in the execution environment.

The Dax toolkit contains code that uses specialized compiler features, such as those with CUDA and OpenMP, or libraries, such as Intel Threading Building Blocks, that will not be available on all machines. Code for these features are encapsulated in their own packages: dax::cuda, dax::openmp, and dax::tbb. Within each one of these packages, there will be cont and exec namespaces as necessary to denote features that are accessible in only one environment or the other.

The Dax toolkit contains OpenGL interoperability that allows data generated with Dax to be efficiently transferred to OpenGL objects. This feature is encapsulated in the dax::-opengl package.

Figure 3.1 provides a diagram of the Dax package hierarchy.

By convention all classes will be defined in a file with the same name as the class name (with a .h extension) located in a directory corresponding to the package name. For example, the dax::cont::ArrayHandle class is found in the dax/cont/ArrayHandle.h header. There are, however, exceptions to this rule. Some smaller classes and types are grouped together

for convenience. These exceptions will be noted as necessary.

Within each namespace there may also be `internal` and `detail` sub-namespaces. The `internal` namespaces contain features that are used internally and may change without notice. The `detail` namespaces contain features that are used by a particular class but must be declared outside of that class. Users should generally ignore classes in these namespaces.

## 3.2 Basic Provisions

This section describes the core facilities provided by the Dax toolkit. These include macros, types, and classes that define the environment in which code is run, the core types of data stored, and template introspection.

### 3.2.1 Function and Method Exports

Any function or method defined by the Dax toolkit must come with an export modifier that determines in which environments the function may be run. These export modifiers are C macros that Dax uses to instruct the compiler for which architectures to compile each method. Most user code outside of the Dax toolkit need not use these macros with the important exception of any classes passed to the Dax toolkit. This occurs when defining new worklets, array containers, and device adapters.

Dax provides three export macros, `DAX_CONT_EXPORT`, `DAX_EXEC_EXPORT`, and `DAX_EXEC_CONT_EXPORT`, which are used to declare functions and methods that can run in the control environment, export environment, and both environments, respectively. These macros get defined by including just about any Dax header file, but including dax/Types.h will ensure they are defined.

The export macro is place after the template declaration, if there is one, and before the return type for the function. Here is a simple example of a function that will square a value. Since most types you would use this function on have operators in both the control and execution environments, the function is exported to both places.

Example 3.1: Usage of export macro.

```
template<class ValueType>
DAX_EXEC_CONT_EXPORT
ValueType Square(const ValueType &inValue)
{
 return inValue * inValue;
}
```

The primary function of the export macros is to interject compiler-specific keywords that specify what architecture to compile code for. For example, when compiling with CUDA, the control exports have `__host__` in them and execution exports have `__device__` in them.

There is one additional export macro that is not used for functions but rather used when declaring a constant data object that is used in the execution environment. This macro is named DAX_EXEC_CONSTANT_EXPORT and is used to declare a constant lookup table used when executing a worklet. Its primary reason for existing is to add a `__constant__` keyword when compiling with CUDA. This export currently has no effect on any other compiler.

### 3.2.2 Core Data Types

Except in rare circumstances where precision is not a concern, the Dax toolkit does not directly use the core C types like `int`, `float`, and `double`. Instead, Dax provides its own core types, which are declared in dax/Types.h.

#### Single Number Types

All floating point values should be declared as type dax::Scalar, and all integer values, generally used for indexing, should be declared as type dax::Id. The chief advantage of using these declared types rather than the core C types is that the precision can easily be changed. By default, both types are 32 bits wide. The CMake configuration options DAX_USE_DOUBLE_PRECISION and DAX_USE_64BIT_IDS can be used to change the dax::Scalar type and dax::Id type, respectively, to be 64 bits wide. The configuration can be overridden by defining the C macro DAX_USE_DOUBLE_PRECISION or DAX_NO_DOUBLE_PRECISION to force dax::Scalar to be either 64 or 32 bits and defining the C macro DAX_USE_64BIT_IDS or DAX_NO_64BIT_IDS to force dax::Id to be either 64 or 32 bits. These macros must be defined before any Dax header files are included to take effect. For convenience, you can include either dax/internal/ConfigureFor32.h or dax/internal/ConfigureFor64.h to force both dax::Scalar and dax::Id to be 32 or 64 bits. The reason Dax uses macros to determine these type widths rather than templates is to reduce the number of template parameters required in the already template-heavy Dax classes and functions.

#### Vector Types

Visualization algorithms also often require operations on short vectors. Arrays indexed in up to three dimensions are common. Data is often defined in 2-space and 3-space, and transformations are typically done in homogeneous coordinates of length 4. To simplify these types of operations, Dax provides several vector data types.

The types dax::Id2 and dax::Id3 are couple and triple values of type dax::Id. The types dax::Vector2, dax::Vector3, and dax::Vector4 are couple, triple, and quadruple values of type dax::Scalar. The elements of these vectors are accessed with the bracket operator, so they syntactically appear like short arrays. They additionally have a constant named NUM_COMPONENTS to specify how many components are in the tuple.

The default constructor of these vector types leaves the values uninitialized. All vectors have a constructor with one arguments that is used to initialize all components. All these vectors also have a constructor that allows you to set the individual components. Likewise, there are a set of dax::make_Id* and dax::make_Vector* functions that build initialized vector types.

Example 3.2: Creating vector types.

```
dax::Vector3 A(1);                          // A is {1, 1, 1}
A[1] = 2;                                   // A is now {1, 2, 1}
dax::Vector3 B(1, 2, 3);                    // B is {1, 2, 3}
dax::Vector3 C = make_Vector3(3, 4, 5); // C is {3, 4, 5}
```

The vector types all support component-wise arithmetic using the operators for plus (+), minus (-), multiply (*), and divide (/). They also support scalar to vector multiplication with the multiply operator. The comparison operators equal (==) is true if every pair of corresponding components are true and not equal (!=) is true otherwise. A special dax::dot function is overloaded to provide a dot product for every type of vector.

Example 3.3: Vector operations.

```
dax::Vector3 A(1, 2, 3);
dax::Vector3 B(4, 5, 6.5);
dax::Vector3 C = A + B;                     // C is {5, 7, 9.5}
dax::Vector3 D = 2 * C;                     // D is {10, 14, 19}
dax::Scalar s = dax::dot(A, B);             // s is 33.5
bool b1 = (A == B);                         // b1 is false
bool b2 = (A == dax::make_Vector3(1, 2, 3); // b2 is true
```

### Tuple

The Dax toolkit provides the templated class dax::Tuple<T,Size>, which is essentially a fixed length array of a given type. dax::Tuple objects behave just like the vector types previously described but with any type and length that you specify.

Example 3.4: The tuple class.

```
dax::Tuple<dax::Scalar, 5> A(2);  // A is {2, 2, 2, 2, 2}
for (int index = 1; index < NUM_COMPONENTS; index++)
  {
  A[index] = A[index-1] * 1.5;
  }
// A is now {2, 3, 4.5, 6.75, 10.125}
```

The same operators that work on the vector types work on dax::Tuple with the caveat that the operator must work on the component type of the tuple. For example, the multiply operator will work fine on objects of type dax::Tuple<char,3>, but the multiply operator will not work on objects of type dax::Tuple<std::string,3> because you cannot multiply objects of type std::string.

A dax::Tuple of the appropriate type can be used interchangeably with a matching vector type. In fact, a vector type is really just a typedef over a dax::Tuple. This is

convenient for a number of things including writing generic functions that work over all types.

Example 3.5: Interchangeability of tuples and vector types.

```
template<typename T, int Size>
DAX_EXEC_CONT_EXPORT
T SumComponents(const dax::Tuple<T,Size> &tuple)
{
  T result = tuple[0];
  for (int index = 1; index < Size; index++)
    {
    result += tuple[index];
    }
  return result;
}

void Foo()
{
  dax::Id a = SumComponents(dax::make_Id3(1, 2, 3));                    // a is 6
  dax::Scalar b = SumComponents(dax::make_Vector4(1.5, 2.5, 3.5, 4.5)); // b is 12
}
```

In addition to generalizing vector operations and making arbitrarily long vectors, `dax::Tuple` is useful for creating any sequence of homogeneous objects. Here is a simple example of using `dax::Tuple` to hold the state of a polygon.

Example 3.6: Usage of a tuple.

```
dax::Tuple<dax::Vector2,3> equilateralTriange(dax::make_Vector2(0.0, 0.0),
                                              dax::make_Vector2(1.0, 0.0),
                                              dax::make_Vector2(0.5, 0.866));
```

### Extents

`dax::Extent3` is a simple structure that holds the extent information for structured data (data defined on a regular grid). It contains to `dax::Id3` fields named `Min` and `Max` that define the minimum and maximum. `dax::Extent3` and several associated helper functions are defined in the dax/Extent.h header.

Example 3.7: Creating and using an `Extent3`.

```
#include <dax/Extent.h>
#include <dax/Types.h>

void ExtentExample()
{
  // Make an extent that defines a grid that has 5x5x3 points and "centered"
  // at index (0,0,0).
  dax::Extent3 extent(dax::make_Id3(-2,-2,-1), dax::make_Id3(2,2,1));

  dax::Id3 minIndices = extent.Min; // Is (-2,-2,-1)
  dax::Id3 maxIndices = extent.Max; // Is (2,2,1)

  dax::Id3 pointDimensions = extentDimensions(extent); // Returns (5,5,3)
  dax::Id3 cellDimensions = extentCellDimensions(extent); // Returns (4,4,2)

  dax::Id3 pointIndexA = flatIndexToIndex3(31, extent); // Returns (-1,-1,0)
  dax::Id3 cellIndexA = flatIndexToIndex3Cell(31, extent); // Returns (1,1,0)
```

```
  dax::Id pointIndexB = index3ToFlatIndex(dax::make_Id3(2,-1,0), extent); // Returns 34
  dax::Id pointIndexB = index3ToFlatIndexCell(dax::make_Id(2,-1,0), extent); // Returns 24
}
```

#### Pair

The Dax toolkit defines a `dax::Pair<T1,T2>` templated object that behaves just like `std::pair` from the standard template library. The difference is that `dax::Pair` will work in both the execution and control environment, whereas the STL `std::pair` does not always work in the execution environment.

The Dax version of `dax::Pair` supports the same types, fields, and operations as the STL version. Dax also provides a `dax::make_Pair` function for convenience.

### 3.2.3 Traits

When using templated types, it is often necessary to get information about the type or specialize code based on general properties of the type. The Dax toolkit uses traits classes to publish and retrieve information about types. A traits class is simply a templated structure that provides typedefs for tag structures, empty types used for identification. The traits classes might also contain constant numbers and helpful static functions. See Mayers [11] for a description of traits classes and their uses.

#### Type Traits

The `dax::TypeTraits<T>` templated class provides basic information about a core type. These type traits are available for all the basic C++ types as well as the core Dax types described in Section 3.2.2. `dax::TypeTraits` contains the following elements.

**`NumericTag`** This type is set to either `dax::TypeTraitsRealTag` or `dax::TypeTraitsIntegerTag` to signal that the type represents either floating point numbers or integers.

**`DimensionalityTag`** This type is set to either `dax::TypeTraitsScalarTag` or `dax::TypeTraitsVectorTag` to signal that the type represents either a single scalar value or a tuple of values.

The definition of `dax::TypeTraits` for `dax::Scalar` could like something like this.

Example 3.8: Definition of `dax::TypeTraits<dax::Scalar>`.

```
namespace dax {
```

```
template<>
struct TypeTraits<dax::Scalar>
{
  typedef TypeTraitsRealTag NumericTag;
  typedef TypeTraitsScalarTag DimensionalityTag;
};

}
```

Here is a simple example of using `dax::TypeTraits` to implement a generic function that behaves like the modulus operator (`%`) for all types including floating points and vectors.

Example 3.9: Using `TypeTraits` for a generic modulus.

```
#include <dax/TypeTraits.h>

template<typename T>
T Modulus(const T &numerator, const T &denominator);

namespace detail {

template<typename T>
T ModulusImpl(const T &numerator,
              const T &denominator,
              dax::TypeTraitsIntegerTag,
              dax::TypeTraitsScalarTag)
{
  return numerator % denominator;
}

template<typename T>
T ModulusImpl(const T &numerator,
              const T &denominator,
              dax::TypeTraitsRealTag,
              dax::TypeTraitsScalarTag)
{
  T quotient = numerator / denominator;
  return (quotient - dax::math::Floor(quotient))*denominator;
}

template<typename T, typename NumericTag>
T ModulusImpl(const T &numerator,
              const T &denominator,
              NumericTag,
              dax::TypeTraitsVectorTag)
{
  T result;
  for (int componentIndex = 0; componentIndex < T::NUM_COMPONENTS; componentIndex++)
    {
    result[componentIndex] = Modulus(numerator[componentIndex],denominator[componentIndex]);
    }
}

} // namespace detail

template<typename T>
T Modulus(const T &numerator, const T &denominator)
{
  return detail::ModulusImpl(numerator,
                             denominator,
                             typename dax::TypeTraits<T>::NumericTag(),
                             typename dax::TypeTraits<T>::DimensionalityTag());
}
```

### Vector Traits

The dax::VectorTraits<T> templated class provides information and accessors to vector and tuple types. It contains the following elements.

**ComponentType** This type is set to the type for each component in the vector. For example, a dax::Vector3 has ComponentType defined as dax::Scalar.

**NUM_COMPONENTS** An integer specifying how many components are contained in the vector.

**HasMultipleComponents** This type is set to either dax::VectorTraitsTagSingleComponent if the vector length is size 1 or dax::VectorTraitsTagMultipleComponents otherwise. This tag can be useful for creating specialized functions when a vector is really just a scalar.

**GetComponent** A static method that takes a vector and returns a particular component.

**SetComponent** A static method that takes a vector and sets are particular component to a given value.

**ToTuple** A static method that converts a vector of the given type to a dax::Tuple.

The definition of dax::VectorTraits for dax::Id3 could like something like this.

Example 3.10: Definition of dax::VectorTraits<dax::Id3>.

```
template<>
struct VectorTraits<dax::Id3>
{
  typedef dax::Id ComponentType;
  static const int NUM_COMPONENTS = 3;
  typedef VectorTraitsTagMultipleComponents HasMultipleComponents;

  DAX_EXEC_CONT_EXPORT
  static dax::Id &GetComponent(dax::Id3 &vector, int component) {
    return vector[component];
  }

  DAX_EXEC_CONT_EXPORT
  static void SetComponent(dax::Id3 &vector, int component, dax::Id value) {
    vector[component] = value;
  }

  DAX_EXEC_CONT_EXPORT
  static dax::Tuple<dax::Id,3> ToTuple(const dax::Id3 &vector) {
    return vector;
  }
};
```

The real power of vector traits is that they simplify creating generic operations on any type that can look like a vector. This includes operations on scalar values as if they were vectors of size one. The following code uses vector traits to simplify the implementation of less functors that define an ordering that can be used for sorting and other operations.

Example 3.11: Using `VectorTraits` for less functors.

```
#include <dax/VectorTraits.h>

// This functor provides a total ordering of vectors. Every compared vector
// will be either less, greater, or equal.
template<typename T>
struct LessTotalOrder
{
  bool operator()(const T &left, const T &right)
  {
    for (int index = 0; index < dax::VectorTraits<T>::NUM_COMPONENTS; index++)
      {
      const T &leftValue = dax::VectorTraits<T>::GetComponent(left, index);
      const T &rightValue = dax::VectorTraits<T>::GetComponent(right, index);
      if (leftValue < rightValue) { return true; }
      if (rightValue < leftValue) { return false; }
      }
    // If we are here, the vectors are equal.
    return false;
  }
};

// This functor provides a partial ordering of vectors. It returns true if and
// only if all components satisfy the less operation. It is possible for
// vectors to be neither less, greater, nor equal, but the transitive closure
// is still valid.
template<typename T>
struct LessTotalOrder
{
  bool operator()(const T &left, const T &right)
  {
    for (int index = 0; index < dax::VectorTraits<T>::NUM_COMPONENTS; index++)
      {
      const T &leftValue = dax::VectorTraits<T>::GetComponent(left, index);
      const T &rightValue = dax::VectorTraits<T>::GetComponent(right, index);
      if (!(leftValue < rightValue)) { return false; }
      }
    // If we are here, all components satisfy less than relation.
    return true;
  }
};
```

## 3.3 Provided Worklets

The Dax toolkit provides several common visualization algorithms encapsulated in worklets that can be executed in parallel on your data. This section describes each of the worklets provided. All worklets provided by Dax are in the `dax::worklet` namespace and defined in header files in the dax/worklet directory.

Much of the support structures for defining data and executing jobs, which you will see in examples, is defined in the Dax control environment. These features are documented in Section 3.4. The Dax toolkit also provides facilities to make it easy to define your own worklet. Descriptions of these features are in Section 3.5.

### 3.3.1 Cell Average

The dax::worklet::CellAverage worklet takes a topology and a field and averages the value of the field in each point. For each cell, it find the field value on each point of the cell and takes the average of those. dax::worklet::CellAverage is a cheap but inaccurate way to integrate the value of a field in each cell. A similar worklet named point data to cell data does a similar operation except that it interpolates the field value to the parametric center of the cell (Section 3.3.8), which may be different than a simple average.

Example 3.12: Cell average worklet.

```
#include <dax/worklet/CellAverage.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherMapCell.h>

template<typename GridType>
DAX_CONT_EXPORT
void RunCellAverage(const GridType &grid,
                    const dax::cont::ArrayHandle<dax::Scalar> &inPointData,
                    dax::cont::ArrayHandle<dax::Scalar> &outCellData)
{
  dax::cont::DispatcherMapCell<dax::worklet::CellAverage> dispatcher;
  dispatcher.Invoke(grid, inPointData, outCellData);
}
```

### 3.3.2 Cell Data to Point Data

The cell data to point data worklet finds all cells incident on each point and then averages the field values of all incident cells to the point.

Running the cell data to point data worklet is a two step process. In the first step, dax::worklet::CellDataToPointDataGenerateKeys extracts point indices for each cell and attaches field values to them. In the second step, dax::worklet::CellDataToPointDataReduceKeys collects field values on a point and averages them.

Example 3.13: Cell data to point data worklet.

```
#include <dax/worklet/CellDataToPointData.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/ArrayHandleConstant.h>
#include <dax/cont/DispatcherGenerateKeysValues.h>
#include <dax/cont/DispatcherReduceKeysValues.h>

#include <dax/CellTraits.h>

template<typename GridType, typename FieldType>
DAX_CONT_EXPORT
void RunCellDataToPointData(const GridType &grid,
                            const dax::cont::ArrayHandle<FieldType> &inPointData,
                            dax::cont::ArrayHandle<FieldType> &outCellData)
{
  dax::cont::ArrayHandleConstant<dax::Id> keyGenCounts =
      dax::cont::make_ArrayHandleConstant<dax::Id>(
            dax::CellTraits<CellTag>::NUM_VERTICES, grid.GetNumberOfCells());
```

```
  dax::cont::DispatcherGenerateKeysValues<
      dax::worklet::CellDataToPointDataGenerateKeys,
      dax::cont::ArrayHandleConstant<dax::Id> > dispatcherGenerateKeys(keyGenCounts);

  dax::cont::ArrayHandle<dax::Id> keyArray;
  dax::cont::ArrayHandle<FieldType> valueArray;

  dispatcherGenerateKeys.Invoke(grid, inPointData, keyArray, valueArray);

  dax::cont::DispatcherReduceKeysValues<dax::worklet::CellDataToPointDataReduceKeys>
      dispatcherReduceKeys(keyArray);

  dispatcherReduceKeys.Invoke(valueArray, outCellData);
}
```

### 3.3.3 Cell Gradient

The `dax::worklet::CellGradient` worklet computes the gradient of a point field at the parametric center of each cell.

Example 3.14: Cell gradient worklet.

```
#include <dax/worklet/CellGradient.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherMapCell.h>

template<typename GridType>
DAX_CONT_EXPORT
void RunCellGradient(const GridType &grid,
                     const dax::cont::ArrayHandle<dax::Scalar> &inPointField,
                     dax::cont::ArrayHandle<dax::Vector3> &outCellGradient)
{
  dax::cont::DispatcherMapCell<dax::worklet::CellGradient> dispatcher;
  dispatcher.Invoke(grid, grid.GetPointCoordinates(), inPointField, outCellGradient);
}
```

### 3.3.4 Cosine

The `dax::worklet::Cosine` worklet computes the cosine of a field. The field can be either a point field or a cell field (or really, just any array).

Example 3.15: Cosine worklet.

```
#include <dax/worklet/Cosine.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherMapField.h>

template<typename FieldType>
DAX_CONT_EXPORT
void RunCosine(const dax::cont::ArrayHandle<FieldType> &inField,
               dax::cont::ArrayHandle<FieldType> &outField)
{
  dax::cont::DispatcherMapField<dax::worklet::Cosine> dispatcher;
  dispatcher.Invoke(inField, outField);
}
```

### 3.3.5 Elevation

The dax::worklet::Elevation worklet find the elevation of points in $R^3$ in relation to a base plane. The orientation of the elevation is determined by a low point location and a high point location. Values lower than the low point and higher than the high point are clamped to the minimum and maximum values. The range of valid values can also be specified.

The elevation worklet is design to be run on the point coordinates of a grid, but in fact could be run on any field or array.

The following example demonstrates finding the elevation of points in a data set oriented along the x axis. Points between $x = -1$ and $x = 1$ are considered. The scale and bias is set to give the distance from the origin along the x-axis in the positive direction.

Example 3.16: Elevation worklet.

```
#include <dax/worklet/Elevation.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherMapField.h>

template<typename GridType>
DAX_CONT_EXPORT
void Elevation(const GridType &grid,
               dax::cont::ArrayHandle<dax::Scalar> &outPointElevation)
{
  dax::worklet::Elevation elevation(dax::make_Vector3(-1.0, 0.0, 0.0),
                                    dax::make_Vector3(1.0, 0.0, 0.0),
                                    dax::make_Vector2(-1.0, 1.0));

  dax::cont::DispatcherMapField<dax::worklet::Elevation> dispatcher(elevation);
  dispatcher.Invoke(grid.GetPointCoordinates(), outPointElevation);
}
```

### 3.3.6 Magnitude

The dax::worklet::Magnitude worklet computes the magnitude of a field of vectors. The field can be either a point field or a cell field (or really, just any array).

Example 3.17: Magnitude worklet.

```
#include <dax/worklet/Magnitude.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherMapField.h>

DAX_CONT_EXPORT
void RunMagnitude(const dax::cont::ArrayHandle<dax::Vector3> &inField,
                  dax::cont::ArrayHandle<dax::Scalar> &outField)
{
  dax::cont::DispatcherMapField<dax::worklet::Magnitude> dispatcher;
  dispatcher.Invoke(inField, outField);
}
```

### 3.3.7 Marching Cubes

The Marching Cubes worklet takes a volume and extracts the contour surface where a field value is equal to a given value.

Running the Marching Cubes worklet is a two step process. In the first step, `dax::worklet::MarchingCubesClassify` identifies how many polygons are going to be generated for every input cell. In the second step, `dax::worklet::MarchingCubesGenerate` creates the triangles that make up the surface.

Example 3.18: Marching Cubes worklet.

```
#include <dax/worklet/MarchingCubes.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherInterpolatedCell.h>
#include <dax/cont/DispatcherMapCell.h>
#include <dax/cont/UnstructuredGrid.h>

template<typename GridType>
DAX_CONT_EXPORT
void RunMarchingCubes(const GridType &inGrid,
                      const dax::cont::ArrayHandle<FieldType> &inPointData,
                      dax::Scalar isovalue,
                      dax::cont::UnstructuredGrid<dax::CellTagTriangle> &outGrid)
{
  dax::cont::ArrayHandle<dax::Id> classification;

  dax::cont::DispatcherMapCell<dax::worklet::MarchingCubesClassify>
      classifyDispatcher(dax::worklet::MarchingCubesClassify(isovalue));
  classifyDispatcher.Invoke(inGrid, inPointData, classification);

  dax::cont::DispatcherInterpolatedCell<dax::worklet::MarchingCubesGenerate>
      generateDispatcher(dax::worklet::MarchingCubesGenerate(isovalue), classification);
  generateDispatcher.Invoke(inGrid, outGrid, inPointData);
}
```

### 3.3.8 Point Data to Cell Data

The `dax::worklet::PointDataToCellData` worklet takes a topology and a field and averages the value of the field in each point. For each cell, it interpolates a point field to the center of the cell. A similar worklet named cell average does a similar operation except that simply averages the field values (Section 3.3.1), which may be different than the interpolation.

The following example uses `dax::worklet::PointDataToCellData` to find the coordinates of each cell center.

Example 3.19: Point data to cell data worklet.

```
#include <dax/worklet/PointDataToCellData.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherMapCell.h>

template<typename GridType>
DAX_CONT_EXPORT
```

```
void RunPointDataToCellData(const GridType &grid,
                            dax::cont::ArrayHandle<dax::Scalar> &outCellCenters)
{
  dax::cont::DispatcherMapCell<dax::worklet::PointDataToCellData> dispatcher;
  dispatcher.Invoke(grid, grid.GetPointCoordinates(), Centers);
}
```

### 3.3.9 Sine

The dax::worklet::Sine worklet computes the sine of a field. The field can be either a point field or a cell field (or really, just any array).

Example 3.20: Sine worklet.

```
#include <dax/worklet/Sine.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherMapField.h>

template<typename FieldType>
DAX_CONT_EXPORT
void RunSine(const dax::cont::ArrayHandle<FieldType> &inField,
             dax::cont::ArrayHandle<FieldType> &outField)
{
  dax::cont::DispatcherMapField<dax::worklet::Sine> dispatcher;
  dispatcher.Invoke(inField, outField);
}
```

### 3.3.10 Slice

The slice worklet takes a volume and intersects it with a plane.

Running the slice worklet is a two step process. In the first step, dax::worklet::SliceClassify identifies how many polygons are going to be generated for every input cell. In the second step, dax::worklet::SliceGenerate creates the triangles that make up the surface that is the intersection of the volume and the plane.

Example 3.21: Slice worklet.

```
#include <dax/worklet/Slice.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherInterpolatedCell.h>
#include <dax/cont/DispatcherMapCell.h>
#include <dax/cont/UnstructuredGrid.h>

template<typename GridType>
DAX_CONT_EXPORT
void RunSlice(const GridType &inGrid,
              const dax::cont::ArrayHandle<FieldType> &inPointData,
              dax::Scalar isovalue,
              dax::cont::UnstructuredGrid<dax::CellTagTriangle> &outGrid)
{
  dax::cont::ArrayHandle<dax::Id> classification;

  dax::cont::DispatcherMapCell<dax::worklet::SliceClassify>
```

```
      classifyDispatcher(dax::worklet::SliceClassify(isovalue));
  classifyDispatcher.Invoke(inGrid, inPointData, classification);

  dax::cont::DispatcherInterpolatedCell<dax::worklet::SliceGenerate>
      generateDispatcher(dax::worklet::SliceGenerate(isovalue), classification);
  generateDispatcher.Invoke(inGrid, outGrid, inPointData);
}
```

### 3.3.11 Square

The dax::worklet::Square worklet computes the square of all the values in a field. (It finds a component-wise square in the case of vector types.) The field can be either a point field or a cell field (or really, just any array).

Example 3.22: Square worklet.

```
#include <dax/worklet/Square.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherMapField.h>

template<typename FieldType>
DAX_CONT_EXPORT
void RunSquare(const dax::cont::ArrayHandle<FieldType> &inField,
               dax::cont::ArrayHandle<FieldType> &outField)
{
  dax::cont::DispatcherMapField<dax::worklet::Square> dispatcher;
  dispatcher.Invoke(inField, outField);
}
```

### 3.3.12 Tetrahedralize

The dax::worklet::Tetrahedralize takes a data set and divides each cell into a group of simplices (tetrahedra) that comprise the volume.

Example 3.23: Tetrahedralize worklet.

```
#include <dax/worklet/Tetrahedralize.h>

#include <dax/cont/ArrayHandleConstant.h>
#include <dax/cont/DispatcherGenerateTopology.h>
#include <dax/cont/UnstructuredGrid.h>

template<typename GridType>
DAX_CONT_EXPORT
void RunTetrahedralize(const GridType &inGrid,
                       dax::cont::UnstructuredGrid<dax::CellTagTetrahedron> &outGrid)
{
  typedef dax::cont::ArrayHandleConstant<dax::Id>
      classification(5, inGrid.GetNumberOfCells());

  dax::cont::DispatcherGenerateTopology<
      dax::worklet::Tetrahedralize,dax::cont::ArrayHandleConstant<dax::Id> >
        dispatcher(classification);
  dispatcher.SetRemoveDuplicatePoints(false);

  dispatcher.Invoke(inGrid, outGrid);
```

```
}
```

### 3.3.13 Threshold

The threshold worklet takes a grid and extracts all cells with field values within a range specified by a minimum and maximum value.

Running the threshold worklet is a two step process. In the first step, `dax::worklet::ThresholdClassify` identifies how many cells are going to be generated for every input cell (0 or 1). In the second step, `dax::worklet::ThresholdTopology` creates a new grid with the passed cells.

Example 3.24: Threshold worklet.

```
#include <dax/worklet/Threshold.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherGenerateTopology.h>
#include <dax/cont/DispatcherMapCell.h>
#include <dax/cont/UnstructuredGrid.h>

template<typename CellType, typename FieldType>
DAX_CONT_EXPORT
void RunThreshold(const dax::cont::UnstructuredGrid<CellType> &inGrid,
                  const dax::cont::ArrayHandle<FieldType> &inPointField,
                  FieldType thresholdMin,
                  FieldType thresholdMax,
                  dax::cont::UnstructuredGrid<CellType> &outGrid,
                  dax::cont::ArrayHandle<FieldType> &outPointField)
{
  typedef dax::cont::ArrayHandle<dax::Id> classification;

  typedef dax::worklet::ThresholdClassify<FieldType> ClassifyWorkletType;
  dax::cont::DispatcherMapCell<ClassifyWorkletType>
      classifyDispatcher(ClassifyWorkletType(thresholdMin, thresholdMax));
  classifyDispatcher.Invoke(inGrid, inPointField, classification);

  dax::cont::DispatcherGenerateTopology<dax::worklet::ThresholdTopology>
      generateDispatcher(classification);

  generateDispatcher.Invoke(inGrid, outGrid);

  generateDispatcher.CompactPointField(inPointField, outPointField);
}
```

## 3.4 Control Environment

The control environment is where code interfaces with applications and I/O devices. The associated API is designed for users that want to use the Dax toolkit to analyze their data using provided or supplied worklets. Code for the control environment is designed to run on a single thread (or one single thread per process in an MPI job).

Most users of the Dax toolkit will have some interaction with the Dax toolkit, for you cannot define data structures or execute any algorithms without it.

### 3.4.1 Device Adapter Tag

The Dax toolkit uses a feature called a device adapter to define what type of device will be used to run algorithms. The device adapter encapsulates the device-specific code required to port to various devices. More information on the function of the device adapter are given in Section 2.3.

The device adapter is identified by a *device adapter tag*. This tag, which is simply an empty struct type, is used as the template parameter for several classes in the Dax control environment and causes these classes to direct their work to a particular device.

There are two ways to select a device adapter. The first is to make a global selection of a default device adapter. The second is to specify a specific device adapter as a template parameter.

#### Default Device Adapter

A default device adapter tag is specified in dax/cont/DeviceAdapter.h (although it can also by specified in many other Dax headers via header dependencies). If no other information is given, Dax attempts to choose a default device adapter that is a best fit for the system it is compiled on. Dax currently select the default device adapter with the following sequence of conditions.

- If the source code is being compiled by CUDA, the CUDA device is used.
- If the CUDA compiler is not being used and the current compiler supports OpenMP, then the OpenMP device is used.
- If the compiler supports neither CUDA nor OpenMP and the Dax Toolkit was configured to use Intel Threading Building Blocks, then that device is used.
- If no parallel device adapters are found, then the Dax Toolkit falls back to a serial device.

You can also set the default device adapter specifically by setting the `DAX_DEVICE_ADAPTER` macro. This macro must be set *before* including any Dax header files. You can set `DAX_DEVICE_ADAPTER` to any one of the following.

`DAX_DEVICE_ADAPTER_SERIAL` Performs all computation on the same single thread as the control environment. This device is useful for debugging. This device is always available.

DAX_DEVICE_ADAPTER_CUDA Uses a CUDA capable GPU device. For this device to work, Dax must be configured to use CUDA and the code must be compiled by the CUDA nvcc compiler.

DAX_DEVICE_ADAPTER_OPENMP Uses OpenMP compiler extensions to run algorithms on multiple threads. For this device to work, Dax must be configured to use OpenMP and the code must be compiled with a compiler that supports OpenMP pragmas.

DAX_DEVICE_ADAPTER_TBB Uses the Intel Threading Building Blocks library to run algorithms on multiple threads. For this device to work, Dax must be configured to use TBB and the executable must be linked to the TBB library.

These macros provide a useful mechanism for quickly reconfiguring code to run on different devices. The following example shows a typical block of code at the top of a source file that could be used for quick reconfigurations.

Example 3.25: Macros to port Dax code among different devices

```
// Uncomment one (and only one) of the following to reconfigure the Dax
// code to use a particular device. Comment them all to automatically pick a
// device.
//#define DAX_DEVICE_ADAPTER DAX_DEVICE_ADAPTER_SERIAL
#define DAX_DEVICE_ADAPTER DAX_DEVICE_ADAPTER_CUDA
//#define DAX_DEVICE_ADAPTER DAX_DEVICE_ADAPTER_OPENMP
//#define DAX_DEVICE_ADAPTER DAX_DEVICE_ADAPTER_TBB

#include <dax/cont/DeviceAdapter.h>
```

The default device adapter can always be overridden by specifying a device adapter tag, as described in the next section. There is actually one more internal default device adapter named DAX_DEVICE_ADAPTER_ERROR that will cause a compile error if any component attempts to use the default device adapter. This feature is most often used in testing code to check when device adapters should be specified.

### Specifying Device Adapter Tags

In addition to setting a global default device adapter, it is possible to explicitly set which device adapter to use in any feature provided by Dax. This is done by providing a device adapter tag as a template argument to Dax templated objects. The following device adapter tags are available in Dax.

dax::cont::DeviceAdapterTagSerial Performs all computation on the same single thread as the control environment. This device is useful for debugging. This device is always available. This tag is defined in dax/cont/DeviceAdapterSerial.h.

dax::cuda::cont::DeviceAdapterTagCuda Uses a CUDA capable GPU device. For this device to work, Dax must be configured to use CUDA and the code must be compiled by the CUDA nvcc compiler. This tag is defined in dax/cuda/cont/DeviceAdapterCuda.h.

**dax::openmp::cont::DeviceAdapterTagOpenMP** Uses OpenMP compiler extensions to run algorithms on multiple threads. For this device to work, Dax must be configured to use OpenMP and the code must be compiled with a compiler that supports OpenMP pragmas. This tag is defined in dax/openmp/cont/DeviceAdapterOpenMP.h.

**dax::tbb::cont::DeviceAdapterTagTBB** Uses the Intel Threading Building Blocks library to run algorithms on multiple threads. For this device to work, Dax must be configured to use TBB and the executable must be linked to the TBB library. This tag is defined in dax/tbb/cont/DeviceAdapterTBB.h.

The following example invokes the elevation worklet much like shown in Example 3.16 on page 39 but also specifies using the Intel Threading Building blocks device adapter. In particular, consider the template parameter of the dax::cont::DispatcherMapField class.

Example 3.26: Calling the Elevation worklet with a specific device adapter.

```
dax::worklet::Elevation elevation(dax::make_Vector3(-1.0, 0.0, 0.0),
                                  dax::make_Vector3(1.0, 0.0, 0.0),
                                  dax::make_Vector2(-1.0, 1.0));

dax::cont::DispatcherMapField<dax::worklet::Elevation, dax::tbb::cont::DeviceAdapterTagTBB>
    dispatcher(elevation);
dispatcher.Invoke(grid.GetPointCoordinates(), outPointElevation);
```

When structuring your code to always specify a particular device adapter, consider setting the default device adapter (with the DAX_DEVICE_ADAPTER macro) to DAX_DEVICE_ADAPTER_ERROR. This will cause the compiler to produce an error if any object is instantiated with the default device adapter, which checks to make sure the code properly specifies every device adapter parameter.

The Dax toolkit also defines a macro named DAX_DEFAULT_DEVICE_ADAPTER_TAG that can be used in place of an explicit device adapter tag to use the default tag. This macro is used to create new templates that have template parameters for device adapters that can use the default. The following example has a (rather artificial) declaration of a helper class for executing the elevation worklet.

Example 3.27: Declaring a template with a default device adapter.

```
template<typename DeviceAdapter = DAX_DEFAULT_DEVICE_ADAPTER_TAG>
class MyElevationDispatcher
{
public:
  void DoInvoke()
  {
    dax::cont::DispatcherMapField<dax::Worklet::Elevation,DeviceAdapter> dispatcher;
    dispatcher.Invoke(this->Grid.GetPointCoordinates(), this->OutPointElevation);
```

### 3.4.2 Array Handle

An *array handle*, implemented with the dax::cont::ArrayHandle class, manages an array of data that can be accessed or manipulated by Dax algorithms. It is typical to construct an array handle in the control environment to pass data to an algorithm running in the execution environment. It is also typical for an algorithm running in the execution environment to allocate and populate an array handle, which can then be read back in the control environment. It is also possible for an array handle to manage data created by one Dax algorithm and passed to another, remaining in the execution environment the whole time and never copied to the control environment.

The array handle may have up to two copies of the array, one for the control environment and one for the execution environment. However, depending on the device and how the array is being used, the array handle will only have one copy when possible. Copies between the environments are implicit and lazy. They are copied only when an operation needs data in an environment where the data is not.

dax::cont::ArrayHandle behaves like a shared smart pointer in that when the C++ object is copied, each copy holds a reference to the same array. These copies are reference counted so that when all copies of the dax::cont::ArrayHandle are destroyed, any allocated memory is released.

#### Creating Array Handles

dax::cont::ArrayHandle is a templated class with three template parameters. The first template parameter is the only one required and specifies the base type of the entries in the array. The second template parameter specifies the container used when storing data in the control environment. Containers are discussed later in this section, and for now we will use the default value. The third template parameter is a device adapter tag that specifies what device is used in the execution environment. Device adapter tags are described in Section 3.4.1. Most of the examples here will use the default device adapter.

Example 3.28: Declaration of the dax::cont::ArrayHandle templated class.

```
template<
    typename T,
    typename ArrayContainerControlTag = DAX_DEFAULT_ARRAY_CONTAINER_CONTROL_TAG,
    typename DeviceAdapterTag = DAX_DEFAULT_DEVICE_ADAPTER_TAG>
class ArrayHandle;
```

There are multiple ways to create and populate an array handle. The default dax::cont::ArrayHandle constructor will create an empty array with nothing allocated in either the control or execution environment. This is convenient for creating arrays used as the output for algorithms.

Example 3.29: Creating an ArrayHandle for output data.

```
dax::cont::ArrayHandle<dax::Scalar> outputArray;
```

Constructing an dax::cont::ArrayHandle that points to a provided C array or std::vector is straightforward with the dax::cont::make_ArrayHandle functions. These functions will make an array handle that points to the array data that you provide.

Example 3.30: Creating an ArrayHandle that points to a provided C array.

```
dax::Scalar dataBuffer[50];
// Populate dataBuffer with meaningful data. Perhaps read data from a file.

dax::cont::ArrayHandle<dax::Scalar> inputArray = dax::cont::make_ArrayHandle(dataBuffer,50);
```

Example 3.31: Creating an ArrayHandle that points to a provided std::vector.

```
std::vector<dax::Scalar> dataBuffer;
// Populate dataBuffer with meaningful data. Perhaps read data from a file.

dax::cont::ArrayHandle<dax::Scalar> inputArray = dax::cont::make_ArrayHandle(dataBuffer);
```

*Be aware* that dax::cont::make_ArrayHandle makes a shallow pointer copy. This means that if you change or delete the data provided, the internal state of dax::cont::ArrayHandle becomes invalid and undefined behavior can ensue. The most common manifestation of this error happens when a std::vector goes out of scope. This subtle interaction will cause the dax::cont::ArrayHandle to point to an unallocated portion of the memory heap. For example, if the code in Example 3.31 where to be placed within a callable function or method, it could cause the dax::cont::ArrayHandle to become invalid.

Example 3.32: Invalidating an ArrayHandle by letting the source std::vector leave scope.

```
DAX_CONT_EXPORT
dax::cont::ArrayHandle<dax::Scalar> BadDataLoad()
{
  std::vector<dax::Scalar> dataBuffer;
  // Populate dataBuffer with meaningful data. Perhaps read data from a file.

  dax::cont::ArrayHandle<dax::Scalar> inputArray = dax::cont::make_ArrayHandle(dataBuffer);

  return inputArray;
  // THIS IS WRONG! At this point dataBuffer goes out of scope and deletes its memory.
  // However, inputArray has a pointer to that memory, which becomes an invalid pointer
  // in the returned object. Bad things will happen when the ArrayHandle is used.
}

DAX_CONT_EXPORT
dax::cont::ArrayHandle<dax::Scalar> SafeDataLoad()
{
  std::vector<dax::Scalar> dataBuffer;
  // Populate dataBuffer with meaningful data. Perhaps read data from a file.

  dax::cont::ArrayHandle<dax::Scalar> tmpArray = dax::cont::make_ArrayHandle(dataBuffer);

  // This copies the data from one ArrayHandle to another (in the execution environment).
  // Although it is an extraneous copy, it is usually pretty fast on a parallel device.
  // Another option is to make sure that the buffer in the std::vector never goes out
  // of scope before all the ArrayHandle references, but this extra step allows the
  // ArrayHandle to manage its own memory and ensure everything is valid.
  dax::cont::ArrayHandle<dax::Scalar> inputArray;
  dax::cont::DeviceAdapterAlgorithm<DAX_DEFAULT_DEVICE_ADAPTER_TAG>::Copy(
      tmpArray, inputArray);

  return inputArray;
  // This is safe.
}
```

### Retrieving Data from an Array Handle

An array handle does not provide direct access to its underlying data by design. The most straightforward way to get data from an array handle is to use the `CopyInto` method. `CopyInto` takes an STL-compatible forward iterator and copies all the data into that iterator. It is assumed that the iterator can be advanced enough to copy all data into the target array. The number of entries in the array handle can be retrieved with the `GetNumberOfValues` method, and the target array should be at least that big.

Example 3.33: Retrieving `ArrayHandle` data with `CopyInto`.

```
dax::cont::ArrayHandle<dax::Scalar> outArray;
// Do something that fills outArray

std::vector<dax::Scalar> resultBuffer(outArray.GetNumberOfValues());
outArray.CopyInto(resultBuffer.begin());
```

There are two other ways data can be retrieved from an array handle. The first is to request an array portal to the data and the second is to define a new container that points to a particular data structure. Both of these methods are discussed in more detail in later sections.

### Array Portals

An array handle defines auxiliary structures called *array portals* that provide direct access into its data. An array portal is a simple object that is somewhat functionally equivalent to an STL-type iterator, but with a much simpler interface. Array portals can be read-only (const) or read-write and they can be accessible from either the control environment or the execution environment. All these variants have similar interfaces although some features that are not applicable can be left out.

An array portal object contains each of the following:

**`ValueType`** A `typedef` of the type for each item in the array.

**`GetNumberOfValues`** A method that returns the number of entries in the array.

**`Get`** A method that returns the value at a given index.

**`Set`** A method that changes the value at a given index. This method does not need to exist for read-only (const) array portals.

**`IteratorType`** A `typedef` of an STL-compatible random-access iterator that can be used for alternative access. This method does not need to exist in the execution environment.

**`GetIteratorBegin`** A method that returns an STL-compatible iterator of type `IteratorType` that points to the beginning of the array. This method does not need exist in the execution environment.

**GetIteratorEnd** A method that returns an STL-compatible iterator of type `IteratorType` that points to the beginning of the array. This method does not need to exist in the execution environment.

The following code example defines an array portal for a simple C array of scalar values. This definition has no practical value (it is covered by the more general `dax::cont::internal::ArrayPortalFromIterators`), but demonstrates the function of each component.

Example 3.34: A simple array portal implementation.

```
#include <dax/Types.h>

class SimpleScalarArrayPortal
{
public:
  typedef dax::Scalar ValueType;

  // There is no specification for creating array portals, but they generally
  // need a constructor like this to be practical.
  DAX_EXEC_CONT_EXPORT
  SimpleScalarArrayPortal(ValueType *array, dax::Id numberOfValues)
    : Array(array), NumberOfValues(numberOfValues) {  }

  DAX_EXEC_CONT_EXPORT
  SimpleScalarArrayPortal() : Array(NULL), NumberOfValues(0) {  }

  DAX_EXEC_CONT_EXPORT
  dax::Id GetNumberOfValues() const { return this->GetNumberOfValues; }

  DAX_EXEC_CONT_EXPORT
  ValueType Get(dax::Id index) const { return this->Array[index]; }

  DAX_EXEC_CONT_EXPORT
  void Set(dax::Id index, ValueType value) const { this->Array[index] = value; }

  typename ValueType *IteratorType;

  DAX_CONT_EXPORT
  IteratorType GetIteratorBegin() const { return this->Array; }

  DAX_CONT_EXPORT
  IteratorType GetIteratorEnd() const { return this->Array + this->GetNumberOfValues(); }

private:
  ValueType *Array;
  dax::Id NumberOfValues;
};
```

`dax::cont::ArrayHandle` contains four `typedef`s for array portal types that are capable of interfacing with the underlying data: two for use in the control environment and two for use in the execution environment. The two used in the control environment are `PortalControl` and `PortalConstControl`, which define read-write and read-only (const) array portals, respectively. Likewise, the two used in the execution environment are `PortalExecution` and `PortalConstExecution`.

Because `dax::cont::ArrayHandle` is an control environment object, it provides the methods `GetPortalControl` and `GetPortalConstControl` to get the associated array portal objects. These methods also have the side effect of refreshing the control environment copy

of the data, so this can be a way of synchronizing the data. Be aware that when an `dax::cont::ArrayHandle` is created with a pointer or `std::vector`, it is put in a read-only mode, and `GetPortalControl` can fail (although `GetPortalConstControl` will still work). Also be aware that calling `GetPortalControl` will invalidate any copy in the execution environment, meaning that any subsequent use will cause the data to be copied back again.

In reality, `GetPortalControl` and `GetPortalConstControl` are only really used for testing purposes or quick access to a particular value. Modifications to an array are better performed in the execution environment. Data is best retrieved by providing a container (described later) that deposits the data directly into your own structures or using the `CopyInto` method (described earlier). Thus, the following example is a bit artificial.

Example 3.35: Using portals from an `ArrayHandle`.

```
#include <dax/cont/ArrayHandle.h>

#include <algorithm>

template<typename T>
void SortCheckArrayHandle(dax::cont::ArrayHandle<T> arrayHandle)
{
  typedef typename dax::cont::ArrayHandle<T>::PortalControl PortalType;
  typedef typename dax::cont::ArrayHandle<T>::PortalConstControl PortalConstType;

  PortalType readwritePortal = arrayHandle.GetPortalControl();
  // This is actually pretty dumb. Sorting would be generally faster in parallel in
  // the execution environment using the device adapter algorithms.
  dax::sort(readwritePortal.GetIteratorBegin(), readwritePortal.GetIteratorEnd());

  PortalConstType readPortal = arrayHandle.GetPortalConstControl();
  for (dax::Id index = 1; index < readPortal.GetNumberOfValues(); index++)
    {
    if (readPortal.Get(index-1) > readPortal.Get(index))
      {
      std::cout << "Sorting is wrong!" << std::endl;
      break;
      }
    }
}
```

### Interface to Execution Environment

One of the main functions of the array handle is to allow an array to be defined in the control environment and then be used in the execution environment. When using an `ArrayHandle` with worklets, this transition is handled automatically. However, it is also possible to invoke the transfer for use in your own scheduled algorithms.

The `ArrayHandle` class manages the transition from control to execution with a set of three methods that allocate, transfer, and ready the data in one operation. These methods all start with the prefix `Prepare` and are meant to be called before some operation happens in the execution environment. The methods are as follows.

**`PrepareForInput`** Copies data from the control to the execution environment, if necessary, and readies the data for read-only access.

**PrepareForInPlace** Copies the data from the control to the execution environment, if necessary, and readies the data for both reading and writing.

**PrepareForOutput** Allocates space (the size of which is given as a parameter) in the execution environment, if necessary, and readies the space for writing.

Each of these methods returns an array portal that can be used in the execution environment. `PrepareForInput` returns an object of type `PortalConstExecution` (defined in the `ArrayHandle`) whereas `PrepareForInPlace` and `PrepareForOutput` each return an object of type `PortalExecution`.

Although these `Prepare` methods are called in the control environment, the returned array portal can only be used in the execution environment. Thus, the portal must be passed to an invocation of the execution environment. Typically this is done with a call to `Schedule` in `dax::cont::DeviceAdapterAlgorithm`. This and other device adapter algorithms are described in detail in Section 3.4.7, but here is a quick example of using these execution array portals in a simple functor.

Example 3.36: Using an execution array portal from an `ArrayHandle`.

```
#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DeviceAdapter.h>

#include <dax/exec/internal/WorkletBase>

template<typename InputPortalType, typename OutputPortalType>
struct DoubleFunctor
{
  DAX_CONT_EXPORT
  DoubleFunctor(InputPortalType inputPortal, OutputPortalType outputPortal)
    : InputPortal(inputPortal), OutputPortal(outputPortal) {  }

  DAX_EXEC_EXPORT
  void operator()(dax::Id index) const {
    this->OutputPortal.Set(index, 2*this->InputPortal.Get(index));
  }

  InputPortalType InputPortal;
  OutputPortalType OutputPortal;
};

template<typename InputArrayType, typename OutputArrayType>
DAX_CONT_EXPORT
void DoubleArray(InputArrayType inputArray, OutputArrayType outputArray)
{
  dax::Id numValues = inputArray.GetNumberOfValues();

  DoubleFunctor<typename InputArrayType::PortalConstExecution,
                typename OutputArrayType::PortalExecution>
    functor(inputArray.PrepareForInput(),
            outputArray.PrepareForOutput());

  typedef typename InputArrayType::DeviceAdapterTag DeviceAdapter;

  dax::cont::DeviceAdapterAlgorithm<DeviceAdapter>::Schedule(functor, numValues);
}
```

It should be noted that the array handle will expect any use of the execution array portal to finish before the next call to any `ArrayHandle` method. Since these `Prepare` methods are

typically used in the process of scheduling an algorithm in the execution environment, this is seldom an issue.

### Basic Container

As previously discussed, dax::cont::ArrayHandle takes three template arguments.

Example 3.37: Declaration of the dax::cont::ArrayHandle templated class (again).

```
template<
    typename T,
    typename ArrayContainerControlTag = DAX_DEFAULT_ARRAY_CONTAINER_CONTROL_TAG,
    typename DeviceAdapterTag = DAX_DEFAULT_DEVICE_ADAPTER_TAG>
class ArrayHandle;
```

The first argument is the only one required and has been demonstrated multiple times before. The third (optional) argument specifies the device adapter, as described in detail in Section 3.4.1. The second (optional) argument specifies something called a container, which provides the interface between the generic dax::cont::ArrayHandle class and a specific storage mechanism in the control environment.

In this and the following sections we describe these control environment containers. A default container is specified in much the same way as a default device adapter is defined. It is done by setting the DAX_ARRAY_CONTAINER_CONTROL macro. This macro must be set before including any Dax header files. Currently the only practical container provided by the Dax toolkit is the basic container, which simply allocates a continuous section of memory of the given base type. This container can be explicitly specified by setting DAX_ARRAY_CONTAINER_CONTROL to DAX_ARRAY_CONTAINER_CONTROL_BASIC although the basic container will also be used as the default if no other container is specified (which is typical).

The default array container can always be overridden by specifying an array container tag. The tag for the basic container is located in the dax/cont/ArrayContainerControl.h header file and is named dax::cont::ArrayContainerControlTagBasic. Here is an example of specifying the container type when declaring an array handle.

Example 3.38: Specifying the container type for an ArrayHandle.

```
dax::cont::ArrayHandle<
  dax::Scalar,
  dax::cont::ArrayContainerControlTagBasic> arrayHandle1;
dax::cont::ArrayHandle<
  dax::Scalar,
  dax::cont::ArrayContainerControlTagBasic,
  dax::cont::DeviceAdapterTagSerial> arrayHandle2;
```

The Dax toolkit also defines a macro named DAX_DEFAULT_ARRAY_CONTAINER_CONTROL_TAG that can be used in place of an explicit array container tag to use the default tag. This macro is used to create new templates that have template parameters for array containers that can use the default or to create array handles with the default container but a specific device adapter.

Example 3.39: An `ArrayHandle` with default container and explicit device.

```
dax::cont::ArrayHandle<
  dax::Scalar,
  DAX_DEFAULT_ARRAY_CONTAINER_CONTROL_TAG,
  dax::cont::DeviceAdapterTagSerial> arrayHandle;
```

### Adapting Data Structures

The intention of the container parameter for `dax::cont::ArrayHandle` is to implement the strategy design pattern [6] to enable the Dax toolkit to interface directly with the data of any third party code source. The Dax toolkit is designed to work with data originating in other libraries or applications. By creating a new type of array container, the entire Dax toolkit can be adapted to new kinds of data structures.

In this section we demonstrate the steps required to adapt the array handle to a data structure provided by a third party. For the purposes of the example, let us say that some fictitious library named "foo" has a simple structure named `FooFields` that holds the field values for a particular part of a mesh, and then maintain the field values for all locations in a mesh in a `std::deque` object.

Example 3.40: Fictitious field storage used in custom array container examples.

```
#include <deque>

struct FooFields {
  float Pressure;
  float Temperature;
  float Velocity[3];
  // And so on...
};

typedef std::deque<FooFields> FooFieldsDeque;
```

The Dax toolkit expects separate arrays for each of the fields rather than a single array containing a structure holding all of the fields. However, rather than copy each field to its own array, we can create a container for each field that points directly to the data in a `FooFieldsDeque` object.

The first step in creating an adapter container is to create a control environment array portal to the data. This is described in more detail starting on page 49 and is generally straightforward for simple containers like this. Here is an example implementation for our `FooFieldsDeque` container.

Example 3.41: Array portal to adapt a third-party container to Dax.

```
#include <dax/cont/Assert.h>
#include <dax/cont/internal/IteratorFromArrayPortal.h>

// DequeType expected to be FooFieldsDeque or const FooFieldsDeque
template<typename DequeType>
class ArrayPortalFooPressure
{
public:
```

```
  typedef dax::Scalar ValueType;

  DAX_CONT_EXPORT
  ArrayPortalFooPressure(DequeType *container) : Container(container) {  }

  DAX_CONT_EXPORT
  dax::Id GetNumberOfValues() const {
    return static_cast<dax::Id>(this->Container->size());
  }

  DAX_CONT_EXPORT
  dax::Scalar Get(dax::Id index) const {
    DAX_ASSERT_CONT(index >= 0);
    DAX_ASSERT_CONT(index < this->GetNumberOfValues());
    return static_cast<dax::Scalar>((*this->Container)[index].Pressure);
  }

  DAX_CONT_EXPORT
  dax::Scalar Set(dax::Id index, dax::Scalar value) const {
    DAX_ASSERT_CONT(index >= 0);
    DAX_ASSERT_CONT(index < this->GetNumberOfValues());
    (*this->Container)[index].Pressure = value;
  }

  typedef dax::cont::internal::IteratorFromArrayPortal<ArrayPortalFooPressure> IteratorType;

  DAX_CONT_EXPORT
  IteratorType GetIteratorBegin() const {
    return IteratorType(*this, 0);
  }

  DAX_CONT_EXPORT
  IteratorType GetIteratorEnd() const {
    return IteratorType(*this, this->GetNumberOfValues());
  }

private:
  DequeType *Container;
};
```

The next step in creating an adapter container is to define a tag for the adapter. We shall call ours `ArrayContainerControlTagFooPressure`. Then, we need to create a specialization of the templated `dax::cont::internal::ArrayContainerControl` class. The `ArrayHandle` will instantiate an object using the array container tag we give it, and we define our own specialization so that it runs our interface into the code.

`dax::cont::internal::ArrayContainerControl` has two template arguments: the base type of the array and the array container tag.

Example 3.42: Prototype for `dax::cont::internal::ArrayContainerControl`.

```
namespace dax {
namespace cont {
namespace internal {

template<typename T, typename ArrayContainerControlTag>
class ArrayContainerControl;

}
}
}
```

The `dax::cont::internal::ArrayContainerControl` must define the following items.

**`ValueType`** A `typedef` of the type for each item in the array. This is the same type as the first template argument.

**`PortalType`** The type of an array portal that can be used to access the underlying data. This array portal needs to work only in the control environment.

**`PortalConstType`** A read-only (const) version of `PortalType`.

**`GetPortal`** A method that returns an array portal of type `PortalType` that can be used to access the data manged in this container.

**`GetPortalConst`** Same as `GetPortal` except it returns a read-only (const) array portal.

**`GetNumberOfValues`** A method that returns the number of values the container is currently allocated for.

**`Allocate`** A method that allocates the array to a given size. An values stored in the previous allocation may be destroyed.

**`Shrink`** A method like `Allocate` with two differences. First, the size of the allocation must be smaller than the existing allocation when the method is called. Second, any values currently stored in the array will be valid after the array is resized. This constrained form of allocation allows the array to be resized and values valid without ever having to copy data.

**`ReleaseResources`** A method that instructs the container to free all of its memory.

The following provides an example implementation of our adapter to a `FooFieldsDeque`. It relies on the `ArrayPortalFooPressure` provided in Example 3.41.

Example 3.43: Array container to adapt a third-party container to Dax.

```
// Includes or definition for ArrayPortalFooPressure

struct ArrayContainerControlTagFooPressure {  };

namespace dax {
namespace cont {
namespace internal {

template<>
class ArrayContainerControl<dax::Scalar, ArrayContainerControlTagFooPressure>
{
public:
  typedef dax::Scalar ValueType;

  typedef ArrayPortalFooPressure<FooFieldsDeque> PortalType;
  typedef ArrayPortalFooPressure<const FooFieldsDeque> PortalConstType;

  DAX_CONT_EXPORT
  ArrayContainerControl(FooFieldsDeque *container) : Container(container) {  }

  DAX_CONT_EXPORT
  PortalType GetPortal() { return PortalType(this->Container); }

  DAX_CONT_EXPORT
  PortalConstType GetPortalConst() const { return PortalConstType(this->Container); }
```

```
  DAX_CONT_EXPORT
  dax::Id GetNumberOfValues() const {
    return static_cast<dax::Id>(this->Container->size());
  }

  DAX_CONT_EXPORT
  void Allocate(dax::Id numberOfValues) { this->Container->resize(numberOfValues); }

  DAX_CONT_EXPORT
  void Shrink(dax::Id numberOfValues) { this->Container->resize(numberOfValues); }

  DAX_CONT_EXPORT
  void ReleaseResources() { this->Container->clear(); }

private:
  FooFieldsDeque *Container;
};

}
}
} // namespace dax::cont::internal
```

The final step to make a container adapter is to make a mechanism to construct an `ArrayHandle` that points to a particular container. This can be done by creating a trivial subclass of `dax::cont::ArrayHandle` that simply constructs the array handle to the state of an existing container.

Example 3.44: Array handle to adapt a third-party container to Dax.

```
template<typename DeviceAdapter>
class ArrayHandleFooPressure
    : public dax::cont::ArrayHandle<
                dax::Scalar, ArrayContainerControlTagFooPressure, DeviceAdapter>
{
private:
  typedef dax::cont::internal
      ::ArrayContainerControl<dax::Scalar,ArrayContainerControlTagFooPressure>
      ArrayContainerControlType;
  typedef dax::cont::internal
      ::ArrayTransfer<T,ArrayContainerControlTagFooPressure,DeviceAdapter>
      ArrayTransferType;
public:
  typedef dax::cont::ArrayHandle<
      dax::Scalar, ArrayContainerControlTagFooPressure, DeviceAdapter> Superclass;

  ArrayHandleFooPressure(FooFieldsDeque *container)
    : Superclass(ArrayContainerControlType(container), true, ArrayTransferType(), false)
  {  }
};
```

With this new version of `ArrayHandle`, the Dax toolkit can now read to and write from the `FooFieldsDeque` structure directly. Note, however, that when writing to an array handle, it is necessary to call `GetPortalControl` or `GetPortalConstControl` to flush data from the execution environment to the control environment.

Example 3.45: Using an `ArrayHandle` with custom container.

```
template<typename GridType>
DAX_CONT_EXPORT
void GetElevationAirPressure(const GridType &grid, FooFieldsDeque *fields)
{
```

```
  dax::worklet::Elevation elevation(dax::make_Vector3(0.0, 0.0, 0.0),
                                    dax::make_Vector3(0.0, 0.0, 10.0),
                                    dax::make_Vector2(0.02, 0.0));

  // Make an array handle that points to the pressure values in fields.
  ArrayHandleFooPressure pressureHandle(fields);

  // Run the elevation worklet.
  dax::cont::DispatcherMapField<dax::worklet::Elevation> dispatcher(elevation);
  dispatcher.Invoke(grid.GetPointCoordinates(), pressureHandle);

  // Make sure values are flushed back to the control environment.
  pressureHandle.GetPortalConstControl();

  // Now the pressure fields are field in the fields container.
};
```

### Implicit Containers

The generic array handle and array container templating in the Dax toolkit allows for any type of operations to retrieve a particular value. Typically this is used to convert an index to some location or locations in memory. However, it is also possible to compute a value directly from an index rather than look up some value in memory. Such an array is completely functionally and requires no storage in memory at all. Such a functional array is specified with an *implicit container*

Specifying a functional or implicit array in the Dax toolkit is straightforward. The Dax toolkit comes with a generic implicit container that can be templated to any function you like. In this section we demonstrate the steps required to create an implicit container. For the purposes of the example, let us say we want an array of even numbers. That is, the array has the values $[0, 2, 4, 6, \ldots]$ up to some given size. Although we could easily create this array in memory, we can save space and possibly time by computing these values on demand.

The first step to creating an implicit container is to build a read-only array portal that computes the desired value in the `Get` method. The portal must work in both the control and execution environments (although the iterators only need to work in the control environment), and no `Set` method is necessary because the array is assumed to be read-only (since it is functional). The array portal may have a small amount of state, but the class itself must be copyable as a raw data structure. That is, using `memcpy` on the structure should work.

Example 3.46: Implicit array portal for an implicit array of even numbers.

```
#include <dax/cont/ArrayContainerControlImplicit.h>
#include <dax/cont/ArrayHandle.h>
#include <dax/cont/internal/IteratorFromArrayPortal.h>

class ArrayPortalEvenNumbers
{
public:
  typedef dax::Id ValueType;
```

```
  DAX_EXEC_CONT_EXPORT
  ArrayPortalEvenNumbers() : NumberOfValues(0) {  }

  DAX_EXEC_CONT_EXPORT
  ArrayPortalEvenNumbers(dax::Id numValues) : NumberOfValues(numValues) {  }

  DAX_EXEC_CONT_EXPORT
  dax::Id GetNumberOfValues() const { return this->NumberOfValues; }

  DAX_EXEC_CONT_EXPORT
  ValueType Get(dax::Id index) const { return 2*index; }

  typedef dax::cont::internal::IteratorFromArrayPortal<ArrayPortalEvenNumbers> IteratorType;

  DAX_CONT_EXPORT
  IteratorType GetIteratorBegin() const
  {
    return IteratorType(*this);
  }

  DAX_CONT_EXPORT
  IteratorType GetIteratorEnd() const
  {
    return IteratorType(*this, this->NumberOfValues);
  }

private:
  dax::Id NumberOfValues;
};
```

Note that this array portal uses the template `dax::cont::internal::IteratorFromArrayPortal`, which can convert any array portal to STL-compatible iterators.

Once the implicit array portal is built, an implicit array container is defined using the `dax::cont::ArrayContainerControlTagImplicit` tag. This tag is templated, and the template parameter is the implicit array portal.

Example 3.47: Defining the container tag for an implicit array of even numbers.

```
typedef dax::cont::ArrayContainerControlTagImplicit<ArrayPortalEvenNumbers>
    ArrayContainerControlTagEvenNumbers;
```

An array handle can be created directly with this tag as the container template parameter to `dax::cont::ArrayHandle`. However, it is common to create a trivial subclass of `dax::cont::ArrayHandle` that simply constructs the array handle to an implicit array portal of a given state. The following example, which builds on Examples 3.46 and 3.47 demonstrates the convenience `dax::cont::ArrayHandle` subclass.

Example 3.48: Implicit array handle of even numbers.

```
template<typename DeviceAdapter>
class ArrayHandleEvenNumbers
    : public dax::cont::ArrayHandle<
                dax::Id, ArrayContainerControlTagEvenNumbers, DeviceAdapter>
{
  typedef dax::cont::ArrayHandle<
                dax::Id, ArrayContainerControlTagEvenNumbers, DeviceAdapter> Superclass;

public:
  ArrayHandleEvenNumbers(dax::Id length)
    : Superclass(ArrayPortalEvenNumbers(length)) {  }
};
```

The Dax toolkit comes with two examples of implicit containers. The first is dax::cont::ArrayHandleConstant, which returns the same value for every index in the array. The constant array is useful when an algorithm that can work on a variable field is used on a constant value. The second is dax::cont::ArrayHandleCounting, which returns the index as the value with a possible offset. The counting array is useful for generating fields of identifiers or for indexing operations. The Dax toolkit also provides dax::cont::make_ArrayHandleConstant and dax::cont::make_ArrayHandleCounting convenience functions to simplify building these arrays.

### Derived Containers

So far, we have discussed using the array container mechanism to adapt to particular memory layout and to create implicit arrays. Yet another option is to create a *derived container*. A derived container shares attributes with both adaptive containers and implicit containers. A derived container takes one or more other arrays and changes their behavior in some way. Their implementation is similar to adapting a memory layout, but some of the details are different.

In this section we will demonstrate the steps required to create a derived container. For the purposes of the example, let us say we want to array handles to behave as one array with the contents concatenated together. We could of course actually copy the data, but we can also do it in place.

As before, the first step to creating a derived container is to build an array portal that will take portals from arrays being derived. The portal must work in both the control and execution environment (or have a separate version for control and execution).

Example 3.49: Derived array portal for concatenated arrays.

```
#include <dax/cont/ArrayContainerControlImplicit.h>
#include <dax/cont/ArrayPortal.h>
#include <dax/cont/Assert.h>
#include <dax/cont/internal/IteratorFromArrayPortal.h>

template<template P1, template P2>
class ArrayPortalConcatenate
{
public:
  typedef P1 PortalType1;
  typedef P2 PortalType2;
  typedef typename PortalType1::ValueType ValueType;

  DAX_EXEC_CONT_EXPORT
  ArrayPortalConcatenate() : FirstPortal(), Portal2() {   }

  DAX_EXEC_CONT_EXPORT
  ArrayPortalConcatenate(const PortalType1 &firstPortal,
                         const PortalType2 &secondPortal)
    : Portal1(firstPortal), Portal2(secondPortal) {  }

  /// Copy constructor for any other ArrayPortalConcatenate with an iterator
  /// type that can be copied to this iterator type. This allows us to do any
  /// type casting that the iterators do (like the non-const to const cast).
  template<class OtherP1, class OtherP2>
```

```
  DAX_CONT_EXPORT
  ArrayPortalConcatenate(const ArrayPortalConcatenate<OtherP1,OtherP2> &src)
    : Portal1(src.GetPortal1()), Portal2(src.GetPortal2()) {  }

  DAX_EXEC_CONT_EXPORT
  dax::Id GetNumberOfValues() const {
    return this->Portal1.GetNumberOfValues() + this->Portal2.GetNumberOfValues();
  }

  DAX_EXEC_CONT_EXPORT
  ValueType Get(dax::Id index) const {
    if (index < this->Portal1.GetNumberOfValues())
      {
      return this->Portal1.Get(index);
      }
    else
      {
      return this->Portal2.Get(index);
      }
  }

  DAX_EXEC_CONT_EXPORT
  ValueType Set(dax::Id index, const ValueType &value) const {
    if (index < this->Portal1.GetNumberOfValues())
      {
      return this->Portal1.Set(index, value);
      }
    else
      {
      return this->Portal2.Set(index, value);
      }
  }

  typedef dax::cont::internal::IteratorFromArrayPortal<
      ArrayPortalConcatenate<PortalType1,PortalType2> > IteratorType;

  DAX_CONT_EXPORT
  IteratorType GetIteratorBegin() const {
    return IteratorType(*this);
  }

  DAX_CONT_EXPORT
  IteratorType GetIteratorEnd() const {
    return IteratorType(*this, this->GetNumberOfValues());
  }

  DAX_EXEC_CONT_EXPORT
  const PortalType1 &GetPortal1() const { return this->Portal1; }
  DAX_EXEC_CONT_EXPORT
  const PortalType2 &GetPortal2() const { return this->Portal2; }

private:
  PortalType1 Portal1;
  PortalType2 Portal2;
};
```

Like in an adapter container, the next step in creating a derived container is to define a tag for the adapter. We shall call ours `ArrayContainerControlTagConcatenate` and it will be templated on the two array handle types that we are deriving. Then, we need to create a specialization of the templated `dax::cont::internal::ArrayContainerControl` class. The implementation for an `ArrayContainerControl` for a derived container is usually trivial compared to an adapter container because the majority of the work is deferred to the derived arrays.

Example 3.50: `ArrayContainerControl` for derived container of concatenated arrays.

```
template<typename ArrayHandleType1, typename ArrayHandleType2>
struct ArrayContainerControlTagConcatenate {  };

namespace dax {
namespace cont {
namespace internal {

template<typename T, typename Container1, typename Container2, typename DeviceAdapter>
class ArrayContainerControl<
    T,
    ArrayContainerControlTagConcatenate<
        dax::cont::ArrayHandle<T, Container1, DeviceAdapter>
        dax::cont::ArrayHandle<T, Container2, DeviceAdapter> > >
{
  typedef dax::cont::ArrayHandle<T, Container1, DeviceAdapter> ArrayHandleType1;
  typedef dax::cont::ArrayHandle<T, Container2, DeviceAdapter> ArrayHandleType2;

public:
  typedef T ValueType;

  typedef ArrayPortalConcatinate<
      typename ArrayHandleType1::PortalControl,
      typename ArrayHandleType2::PortalControl> PortalType;
  typedef ArrayPortalConcatinate<
      typename ArrayHandleType1::PortalConstControl,
      typename ArrayHandleType2::PortalConstControl> PortalConstType;

  DAX_CONT_EXPORT
  ArrayContainerControl() : Valid(false) {  }

  DAX_CONT_EXPORT
  ArrayContainerControl(const ArrayHandleType1 firstArrayHandle,
                        const ArrayHandle2 secondArrayHandle)
    : Array1(firstArrayHandle), Array2(secondArrayHandle) {  }

  DAX_CONT_EXPORT
  PortalType GetPortal() {
    DAX_ASSERT_CONT(this->Valid);
    return PortalType(this->Array1.GetPortalControl(), this->Array2.GetPortalControl());
  }

  DAX_CONT_EXPORT
  PortalConstType GetPortalConst() const {
    DAX_ASSERT_CONT(this->Valid);
    return PortalType(this->Array1.GetPortalConstControl(),
                      this->Array2.GetPortalConstControl());
  }

  DAX_CONT_EXPORT
  dax::Id GetNumberOfValues() const {
    DAX_ASSERT_CONT(this->Valid);
    return this->Array1.GetNumberOfValues() + this->Array2.GetNumberOfValues();
  }

  DAX_CONT_EXPORT
  void Allocate(dax::Id numberOfValues) {
    DAX_ASSERT_CONT(this->Valid);
    // This implementation of allocate, which allocates the same amount in both arrays, is
    // arbitrary. It could, for example, leave the size of Array1 alone and change the size
    // of Array2. Or, probably most likely, it could simply throw an error and state that
    // this operation is invalid.
    dax::Id half = numberOfValues/2;
    // PrepareForOutput is the only accessible way to resize an ArrayHandle.
    this->Array1.PrepareForOutput(numberOfValues-half);
    this->Array2.PrepareForOutput(half);
  }
```

```
  DAX_CONT_EXPORT
  void Shrink(dax::Id numberOfValues) {
    DAX_ASSERT_CONT(this->Valid);
    if (numberOfValues < this->Array1.GetNumberOfValues())
      {
      this->Array1.Shrink(numberOfValues);
      this->Array2.Shrink(0);
      }
    else
      {
      this->Array2.Shrink(numberOfValues - this->Array1.GetNumberOfValues());
      }
  }

  DAX_CONT_EXPORT
  void ReleaseResources() {
    DAX_ASSERT_CONT(this->Valid);
    this->Array1.ReleaseResources();
    this->Array2.ReleaseResources();
  }

private:
  ArrayHandleType1 Array1;
  ArrayHandleType2 Array2;
  bool Valid;
};

}
}
} // namespace dax::cont::internal
```

One of the responsibilities of an array handle is to copy data between the control and execution environments. The default behavior is to request the device adapter to copy data items from one environment to another. This might involve transferring data between a host and device. For an array of data resting in memory, this is necessary. However, implicit containers (described in the previous section) override this behavior to pass nothing but the functional array portal. Likewise, it is undesirable to do a raw transfer of data with derived containers. The underlying arrays being derived may be used in other contexts, and it would be good to share the data wherever possible. It is also sometimes more efficient to copy data independently from the arrays being derived than from the derived container itself.

The mechanism that controls how a particular control array container gets transferred to and from the execution environment is encapsulated in the templated `dax::cont::internal::ArrayTransfer` class. By creating a specialization of `dax::cont::internal::ArrayTransfer`, we can modify the transfer behavior to instead transfer the arrays being derived and use the respective copies in the control and execution environments.

`dax::cont::internal::ArrayTransfer` has three template arguments: the base type of the array, the array container tag, and the device adapter tag.

Example 3.51: Prototype for `dax::cont::internal::ArrayTransfer`.

```
namespace dax {
namespace cont {
namespace internal {

template<typename T, class ArrayContainerControlTag, class DeviceAdapterTag>
class ArrayTransfer;
```

```
}
}
}
```

The `dax::cont::internal::ArrayTransfer` must define the following items.

**`ValueType`** A `typedef` of the type for each item in the array. This is the same type as the first template argument.

**`PortalControl`** The type of an array portal that is used to access the underlying data in the control environment.

**`PortalConstControl`** A read-only (const) version of `PortalControl`.

**`PortalExecution`** The type of an array portal that is used to access the underlying data in the execution environment.

**`PortalConstExecution`** A read-only (const) version of `PortalExecution`.

**`GetNumberOfValues`** A method that returns the number of values currently allocated in the execution environment. The results may be undefined if none of the load or allocate methods have yet been called.

**`LoadDataForInput`** A method that takes an array portal of type `PortalConstControl`, allocates enough space in the execution environment, and copies the given data to that array. The allocated array can later be accessed via the `GetPortalConstExecution` method. The data is assumed to be read-only.

**`LoadDataForInPlace`** A method that takes an array portal of type `PortalControl`, allocates enough space in the execution environment, and copies the given data to that array. The allocated array can later be accessed via the `GetPortalExecution` and `GetPortalConstExecution` methods. The data can be read and written.

**`AllocateArrayForOutput`** A method that takes an array container and a size and allocates an array in the execution environment of the specified size. The initial memory is uninitialized and can be accessed via the `GetPortalExecution` method. The container argument can be used to allocate data when the control and execution share arrays, but this argument is often ignored.

**`RetrieveOutputData`** This method takes an array container, allocates memory in the control environment, and copies data from the execution environment into it.

**`CopyInto`** This method takes an STL-compatible iterator and copies data from the execution environment into it.

**`Shrink`** A method that adjusts the size of the array in the execution environment to something that is a smaller size. All the data up to the new length must remain valid. Typically, no memory is actually reallocated. Instead, a different end is marked.

**GetPortalExecution** A method that returns an array portal that can be used in the execution environment. The portal was defined in either `LoadDataForInPlace` or `AllocateArrayForOutput`.

**GetPortalConstExecution** A method that returns a read-only (const) array portal that can be used in the execution environment. The portal was defined in one of the load or allocate methods.

**ReleaseResources** A method that frees any resources (typically memory) in the execution environment.

Continuing our example derived container that concatenates two arrays started in Examples 3.49 and 3.50, the following provides an `ArrayTransfer` appropriate for the derived container.

Example 3.52: `ArrayTransfer` for derived container of concatenated arrays.

```
namespace dax {
namespace cont {
namespace internal {

template<class ArrayHandleType1,
         class ArrayHandleType2,
         class DeviceAdapter>
class ArrayTransfer<
    typename ArrayHandleType1::ValueType,
    ArrayContainerControlTagConcatenate<ArrayHandleType1,ArrayHandleType2>,
    DeviceAdapter>
{
public:
  typedef typename ArrayHandleType1::ValueType ValueType;

private:
  typedef
  ArrayContainerControlTagConcatenate<ArrayHandleType1,ArrayHandleType2>
    ContainerTag;
  typedef dax::cont::internal::ArrayContainerControl<ValueType, ContainerTag> ContainerType;

public:
  typedef typename ContainerType::PortalType PortalControl;
  typedef typename ContainerType::PortalConstType PortalConstControl;

  typedef ArrayPortalConcatinate<
      typename ArrayHandleType1::PortalExecution,
      typename ArrayHandleType2::PortalExecution> PortalExecution;
  typedef ArrayPortalConcatinate<
      typename ArrayHandleType1::PortalConstExecution,
      typename ArrayHandleType2::PortalConstExecution> PortalConstExecution;

  DAX_CONT_EXPORT
  ArrayTransfer()
    : ArraysValid(false),
      ExecutionPortalConstValid(false),
      ExecutionPortalValid(false)
  {  }

  DAX_CONT_EXPORT
  ArrayTransfer(ArrayHandleType1 firstArray,
                ArrayHandleType2 secondArray)
    : Array1(firstArray),
      Array2(secondArray),
      ArraysValid(true),
```

```
    ExecutionPortalConstValid(false),
    ExecutionPortalValid(false)
{  }

DAX_CONT_EXPORT
dax::Id GetNumberOfValues() const {
  DAX_ASSERT_CONT(this->ArraysValid);
  return this->Array1.GetNumberOfValues() + this->Array2.GetNumberOfValues();
}

DAX_CONT_EXPORT
void LoadDataForInput(PortalConstControl daxNotUsed(portal)) {
  // Assuming portal was created from a container with the same two arrays.
  DAX_ASSERT_CONT(this->ArraysValid);
  this->ExecutionPortalConst = PortalConstExecution(this->Array1.PrepareForInput(),
                                                    this->Array2.PrepareForInput());
  this->ExecutionPortalConstValid = true;
  this->ExecutionPortalValid = false;
}

DAX_CONT_EXPORT
void LoadDataForInPlace(PortalControl daxNotUsed(portal)) {
  // Assuming portal was created from a container with the same two arrays.
  DAX_ASSERT_CONT(this->ArraysValid);
  this->ExecutionPortal = PortalExecution(this->Array1.PrepareForInPlace(),
                                          this->Array2.PrepareForInPlace());

  this->ExecutionPortalConst = this->ExecutionPortal;
  this->ExecutionPortalConstValid = true;
  this->ExecutionPortalValid = true;
}

DAX_CONT_EXPORT
void AllocateArrayForOutput(ContainerType &daxNotUsed(controlArray),
                            dax::Id numberOfValues) {
  // Assuming controlArray uses the same arrays as this.
  DAX_ASSERT_CONT(this->ArraysValid);

  // This implementation of allocate, which allocates the same amount in both arrays, is
  // arbitrary. It could, for example, leave the size of Array1 alone and change the size
  // of Array2. Or, probably most likely, it could simply throw an error and state that
  // this operation is invalid.
  dax::Id half = numberOfValues/2;
  this->ExecutionPortal
      = PortalExecution(this->Array1.PrepareForOutput(numberOfValues-half),
                        this->Array2.PrepareForOutput(half));
  this->ExecutionPortalValid = true;
  this->ExecutionPortalConstValid = false;
}

DAX_CONT_EXPORT
void RetrieveOutputData(ContainerType &daxNotUsed(controlArray)) const {
  // Implementation of this method should be unnecessary. The internal
  // first and second array handles should automatically retrieve the
  // output data as necessary.
}

template<typename IteratorTypeControl>
DAX_CONT_EXPORT
void CopyInto(IteratorTypeControl dest) const {
  DAX_ASSERT_CONT(this->ArraysValid);
  this->Array1->CopyInto(dest);
  this->Array2->CopyInto(dest + this->Array1.GetNumberOfValues());
}

DAX_CONT_EXPORT
void Shrink(dax::Id numberOfValues) {
  DAX_ASSERT_CONT(this->ArraysValid);
```

```
    if (numberOfValues < this->Array1.GetNumberOfValues())
      {
      this->Array1.Shrink(numberOfValues);
      this->Array2.Shrink(0);
      }
    else
      {
      this->Array2.Shrink(numberOfValues - this->Array1.GetNumberOfValues());
      }
  }

  DAX_CONT_EXPORT
  PortalExecution GetPortalExecution() {
    DAX_ASSERT_CONT(this->ExecutionPortalValid);
    return this->ExecutionPortal;
  }

  DAX_CONT_EXPORT
  PortalConstExecution GetPortalConstExecution() const {
    DAX_ASSERT_CONT(this->ExecutionPortalConstValid);
    return this->ExecutionPortalConst;
  }

  DAX_CONT_EXPORT
  void ReleaseResources() {
    DAX_ASSERT_CONT(this->ArraysValid);
    this->Array1.ReleaseResourcesExecution();
    this->Array2.ReleaseResourcesExecution();
    this->ExecutionPortalValid = false;
    this->ExecutionPortalConstValid = false;
  }

private:
  ArrayHandleType1 Array1;
  ArrayHandleType2 Array2;
  bool ArraysValid;
  PortalConstExecution ExecutionPortalConst;
  bool ExecutionPortalConstValid;
  PortalExecution ExecutionPortal;
  bool ExecutionPortalValid;
};

}
}
} // namespace dax::cont::internal
```

The final step to make a derived container is to create a mechanism to construct an `ArrayHandle` with a container derived from the desired arrays. This can be done by creating a trivial subclass of `dax::cont::ArrayHandle` that simply constructs the array handle to the state of an existing container. It uses a protected constructor of `dax::cont::ArrayHandle` that accepts a constructed container, array transfer, and flags on the status of the control and execution arrays.

Example 3.53: `ArrayHandle` for derived container of concatenated arrays.

```
template<typename ArrayHandleType1, typename ArrayHandleType2>
class ArrayHandleConcatenate
  : public dax::cont::ArrayHandle<
      typename ArrayHandleType1::ValueType,
      ArrayContainerControlTagConcatenate<ArrayHandleType1,ArrayHandleType2>,
      typename ArrayHandleType1::DeviceAdapterTag>
{
  typedef ArrayContainerControlTagConcatenate<ArrayHandleType1,ArrayHandleType2>
      ContainerTag;
```

```
  typedef dax::cont::ArrayHandle<
      typename ArrayHandleType1::ValueType,
      ContainerTag,
      typename ArrayHandleType1::DeviceAdapterTag> Superclass;
  typedef dax::cont::internal::ArrayContainerControl<T,ContainerTag> ContainerType;
  typedef dax::cont::internal::ArrayTransfer<
      typename ArrayHandleType1::ValueType,
      ContainerTag,
      typename ArrayHandleType1::DeviceAdapterTag> TransferType;

public:
  ArrayHandleConcatenate(const ArrayHandleType1 &array1, const ArrayHandleType2 &array2)
    : Superclass(ContainerType(array1, array2),
                 true,
                 TransferType(array1, array2),
                 false)
  { }
};
```

### 3.4.3 Grid Structures

The Dax toolkit provides containers for topologies. The topologies are built on the previously described data structures (mostly array handles) and are intentionally simplistic to simplify the adaptation to other structures.

The grid structures are independent classes. They have no common superclass. However, they do have some elements that are expected to be common across all grids classes, which can be used in a templated environment.

All grid structures have methods named `GetNumberOfPoints` and `GetNumberOfCells`. These methods, of course, return the number of points or cells in the grid structure.

All grid structures have a method called `GetPointCoordinates`. This method returns an array handle that contains the spatial coordinates for all the points in the mesh. Topologies with implicit connections might return an array with an implicit or derived container (meaning that the data is functionally defined rather than stored in memory), but the arrays behave the same regardless. The type of the array returned by `GetPointCoordinates` is specified by the type `PointCoordinatesType` defined in the grid class. This will be either a `typedef` of an `ArrayHandle` with specific template parameters or a subclass of `ArrayHandle`.

It is also possible to query the point coordinates for any given point with the `ComputePointCoordinates` method. This method is mainly provided for testing purposes. Most point coordinate operations should be performed in the execution environment.

The `GetPointCoordinates` method is most useful for invoking an operation on the point coordinates as a field on points. We have seen this method used on the examples of the elevation worklet.

Example 3.54: Processing point coordinates from an unknown grid type.

```
template<typename GridType>
DAX_CONT_EXPORT
```

```
void Elevation(const GridType &grid,
               dax::cont::ArrayHandle<dax::Scalar> &outPointElevation)
{
  dax::cont::DispatcherMapField<dax::worklet::Elevation> dispatcher;
  dispatcher.Invoke(grid.GetPointCoordinates(), outPointElevation);
}
```

Each grid structure contains a particular type of cell. Each grid structure defines a type named `CellTag` that identifies the type of cell stored. Cell types and operations that can be performed in the execution environment are described in Section 3.5.4.

All grid structures also have the facilities to pack information to be sent to the execution environment. There is a type defined in the grid class called `TopologyStructConstExecution` for read-only input data and a `PrepareForInput` method to build the structure. Likewise, there is a `TopologyStructExecution` type and `PrepareForOutput` method for output data.

The execution structures, however, differ significantly. Typically, these facilities are handled internally within Dax to pass data to worklets.

### Uniform Grid

A uniform grid is stored in a `dax::cont::UniformGrid` class. A uniform grid is a topology structure where its points form a regular 3D array. The 3D array of points are axis aligned, and the spacing is uniform along each dimension. Adjacent are connected together in *voxel* cells, which are simply axis aligned hexahedra.

The topology of a uniform grid is completely implicit and specified with three pieces of information. First, the extent, stored in a `dax::Extent3` structure, specifies the minimum and maximum indices of the array. Second, the origin, stored in a `dax::Vector3`, gives the point coordinates of the point at index $[0, 0, 0]$ (which may not actually be in the extent of the grid). Third, the spacing, stored in a `dax::Vector3`, gives the amount of space between adjacent points in each dimension.

The uniform grid class is templated on the device adapter for which it is being used. Its prototype looks as follows.

Example 3.55: Prototype for `dax::cont::UniformGrid`.

```
template <class DeviceAdapterTag = DAX_DEFAULT_DEVICE_ADAPTER_TAG>
class UniformGrid;
```

The `dax::cont::UniformGrid` class provides the following features.

**`CellTag`** A type that identifies what kind of cell is stored in this class. Always set to `dax::CellTagVoxel`.

**`GetExtent`** A method that returns a `dax::Extent3` specifying the extent of the 3 dimensional indices.

**SetExtent** A method that sets the extent of the 3 dimensional indices. There are two versions of SetExtent: one that accepts a dax::Extent3 object and another that accepts two dax::Id3 objects specifying the minimum and maximum indices.

**GetOrigin** A method that returns a dax::Vector3 specifying the coordinates for the origin of the grid.

**SetOrigin** A method that accepts a dax::Vector3 as a parameter to set the coordinates for the origin of the grid.

**GetSpacing** A method that returns a dax::Vector3 specifying the spacing between adjacent points along each dimension.

**SetSpacing** A method that accepts a dax::Vector3 as a parameter to set the spacing between adjacent points along each dimension.

**GetNumberOfPoints** A method that returns the number of points in the grid.

**GetNumberOfCells** A method that returns the number of cells in the grid.

**ComputePointIndex** A convenience method that takes a dax::Id3 representing the 3 dimensional coordinates of a point and returns the one dimensional index for the point. The 1 dimensional index corresponds to the index for point field arrays contained in dax::cont::ArrayHandle objects.

**ComputeCellIndex** A convenience method that takes a dax::Id3 representing the 3 dimensional coordinates of a cell and returns the one dimensional index for the cell. The 1 dimensional index corresponds to the index for cell field arrays contained in dax::cont::ArrayHandle objects.

**ComputePointLocation** A convenience method that takes a 1 dimensional point index and returns the corresponding 3 dimensional index as a dax::Id3. This method performs the inverse operation of ComputePointIndex.

**ComputeCellLocation** A convenience method that takes a 1 dimensional cell index and returns the corresponding 3 dimensional index as a dax::Id3. This method performs the inverse operation of ComputeCellIndex.

**ComputePointCoordinates** A convenience method that returns the spatial coordinates for a given point. This method is overloaded to accept either a 1 dimensional index as a dax::Id or a 3 dimensional index as a dax::Id3.

**GetPointCoordinates** Returns a dax::cont::ArrayHandle containing spatial coordinates for each point. This array can be used as a field when invoking worklets. The array is implicit.

**PointCoordinatesType** The type returned by GetPointCoordinates. It is a specialization of ArrayHandle.

**TopologyStructConstExecution** A memory copyable structure holding the state of the uniform grid that can be used in the execution environment.

**PrepareForInput** A method that returns a TopologyStructConstExecution object to pass to the execution environment. This method is typically only used internally within the Dax toolkit.

### Unstructured Grid

An unstructured grid is stored in a dax::cont::UnstructuredGrid class. An unstructured grid is a topology with a collection of cells connected in arbitrary ways. It first defines a list of points. It then has a connection list that specifies for each cell the points that comprise the vertices for each cell. The dax::cont::UnstructuredGrid class is limited to containing cells of only one type.

The topology of an unstructured grid is defined with a point coordinates array and a cell connections array. The point coordinates array is an array of dax::Vertex3 values containing one for each point. The cell connections array is an array of dax::Id values. The length of this array is the number of cells times the number of vertices per cell. The connections for a particular cell are grouped together in adjacent array values. The cell connections are given in CGNS order [15]. An example cell connection array is given in Figure 3.2.

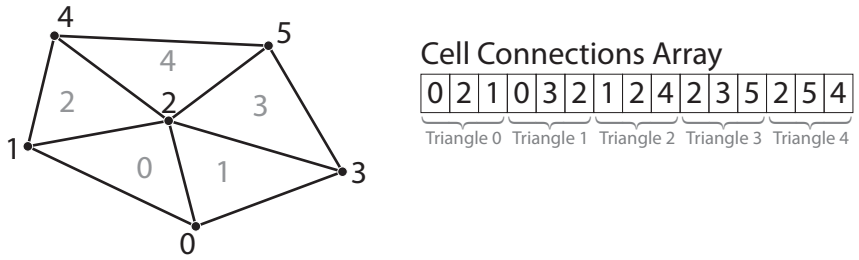



Figure 3.2: The cell connection array for a simple triangle mesh.

The unstructured grid class is templated on the cell type (dax::CellTagHexahedron, dax::CellTagLine, dax::CellTagQuadrilateral, dax::CellTagTetrahedron, dax::CellTagTriangle, dax::CellTagVertex, or dax::CellTagWedge) the container for cell connections, the container for the point coordinate array, and the device adapter. Its prototype looks as follows.

Example 3.56: Prototype for dax::cont::UnstructuredGrid.

```
template <
    typename CellT,
    class CellConnectionsContainerControlTag = DAX_DEFAULT_ARRAY_CONTAINER_CONTROL_TAG,
    class PointsArrayContainerControlTag = DAX_DEFAULT_ARRAY_CONTAINER_CONTROL_TAG,
    class DeviceAdapterTag = DAX_DEFAULT_DEVICE_ADAPTER_TAG>
class UnstructuredGrid;
```

The dax::cont::UnstructuredGrid class provides the following features.

**CellTag** A type that identifies what kind of cell is stored in this class. Always set to the first template parameter.

**CellConnectionsType** The type of the dax::cont::ArrayHandle used to store cell connection indices.

**PointCoordinatesType** The type of the dax::cont::ArrayHandle used to store point coordinates.

**GetCellConnections** A method used to get the array handle for the cell connections.

**SetCellConnetions** A method used to set the array handle for the cell connections.

**GetPointCoordinates** A method used to get the array handle for point coordinates.

**SetPointCoordinates** A method used to set the array handle for point coordinates.

**ComputePointCoordinates** A convenience method that takes a point index and returns the point coordinates at that index. The actual value is pulled from the point coordinates array.

**GetNumberOfPoints** A method that returns the number of points in the grid.

**GetNumberOfCells** A method that returns the number of cells in the grid.

**TopologyStructExecution** A memory copyable structure holding the state of the uniform grid that can be used in the execution environment.

**TopologyStructConstExecution** A read-only (const) form of TopologyStructuExecution.

**PrepareForInput** A method that returns a TopologyStructConstExecution object to pass to the execution environment. This method is typically only used internally within the Dax toolkit.

**PrepareForOutput** A method that returns a TopologyStructExecution object to pass to the execution environment. This method is typically only used internally within the Dax toolkit.

### 3.4.4 Dispatchers

Worklets, both those provided by the Dax toolkit as listed in Section 3.3 and ones created by a user as described in Section 3.5.1, are instantiated in the control environment and run in the execution environment. This means that the control environment must have a means to *invoke* worklets that start running in the execution environment.

This invocation is done through a set of *dispatcher* objects. A dispatcher object is an object in the control environment that has an instance of a worklet and can invoke that worklet with a set of arguments. There are multiple types of dispatcher objects, each corresponding to a type of worklet object. All dispatcher objects have at least two template parameters: the worklet class being invoked, which is always the first argument, and the device adapter tag, which is always the last argument and will be set to the default device adapter if not specified.

All dispatcher classes have a method named `Invoke` that launches the worklet in the execution environment. The arguments to `Invoke` must match those in the control signature of the worklet held by the dispatcher.

The following is a list of the dispatchers defined in the Dax toolkit. The dispatcher classes corresponded to list of worklet types as specified in Section 3.5.1 starting on page 88. See that section for more details and examples of using these dispatcher classes.

**dax::cont::DispatcherMapField** The dispatcher used in conjunction with a worklet that subclasses dax::exec::WorkletMapField. The class has two template arguments: the worklet type and the device adapter (optional).

**dax::cont::DispatcherMapCell** The dispatcher used in conjunction with a worklet that subclasses dax::WorkletMapCell. The class has two template arguments: the worklet type and the device adapter (optional).

**dax::cont::DispatcherGenerateTopology** The dispatcher used in conjunction with a worklet that subclasses dax::WorkletGenerateTopology. The class has three template arguments: the worklet type, the type of array handle containing the count of the number of cells being generated (optional), and the device adapter (optional). The default type of the count array handle is dax::cont::ArrayHandle<dax::Id>. An instance of the count array handle must be provided in the constructor of dax::cont::DispatcherGenerateTopology.

**dax::cont::DispatcherInterpolatedCell** The dispatcher used in conjunction with a worklet that subclasses dax::WorkletInterpolatedCell. The class has three template arguments: the worklet type, the type of array handle containing the count of the number of cells being generated (optional), and the device adapter (optional). The default type of the count array handle is dax::cont::ArrayHandle<dax::Id>. An instance of the count array handle must be provided in the constructor of dax::cont::DispatcherInterpolatedCell.

**dax::cont::DispatcherGenerateKeysValues** The dispatcher used in conjunction with a worklet that subclasses dax::WorkletGenerateKeysValues. The class has three template arguments: the worklet type, the type of array handle containing the count of the number of key-values being generated (optional), and the device adapter (optional). The default type of the count array handle is dax::cont::ArrayHandle<dax::Id>. An instance of the count array handle must be provided in the constructor of dax::cont::DispatcherGenerateKeysValues.

**dax::cont::DispatcherReduceKeysValues** The dispatcher used in conjunction with a worklet that subclasses dax::WorkletReduceKeysValues. The class has three template arguments: the worklet type, the type of array handle containing the keys (optional), and the device adapter (optional). The default type of the key array handle is dax::cont::ArrayHandle<dax::Id>. An instance of the key array handle must be provided in the constructor of dax::cont::DispatcherReduceKeysValues.

### 3.4.5 Timers

It is often the case that you need to measure the time it takes for an operation to happen. This could be for performing measurements for algorithm study or it could be to dynamically adjust scheduling.

Performing timing in a multi-threaded environment can be tricky because operations happen asynchronously. In the Dax control environment timing is simplified because the control environment operates on a single thread. However, operations invoked in the execution environment may run asynchronously to operations in the control environment.

To ensure that accurate timings can be made, Dax provides a dax::cont::Timer class that is templated on the device adapter to provide an accurate measurement of operations that happen on the device. The timing starts when the Timer is constructed. The time elapsed can be retrieved with a call to the GetElapsedTime method. This method will block until all operations in the execution environment complete so as to return an accurate time. The timer can be restarted with a call to the Reset method.

Example 3.57: Using dax::cont::Timer.

```
dax::cont::UniformGrid<> grid;
grid.SetExtent(dax::make_Id3(0, 0, 0), dax::make_Id3(99, 99, 99));
grid.SetOrigin(dax::make_Vector3(0.0, 0.0, 0.0));
grid.SetSpacing(dax::make_Vector3(1.0, 1.0, 1.0));

dax::cont::ArrayHandle<dax::Scalar> results;
dax::cont::DispatchMapField<dax::worklet::Elevation> dispatcher;

dax::cont::Timer<> timer;
dispatcher.Invoke(grid.GetPointCoordinates(), results);
// This call makes sure data is pulled back to the host in a host/device architecture.
results.GetPortalConstControl();
dax::Scalar elapsedTime = timer.GetElapsedTime();

std::cout << "Time to run elevation: " << elapsedTime << std::endl;
```

### 3.4.6 Error Handling

The Dax toolkit uses exceptions to report errors. All exceptions thrown by Dax will be a subclass of dax::cont::Error. For simple error reporting, it is possible to simply catch a dax::cont::Error and report the error message string reported by the GetMessage method.

Example 3.58: Simple error reporting.

```
#include <dax/cont/Error.h>

int main(int argc, char **argv)
{
  try
    {
    // Do something cool with Dax
    // ...
    }
  catch (dax::cont::Error error)
    {
    std::cout << error.GetMessage() << std::endl;
    return 1;
    }
  return 0;
}
```

There are two subclasses to `dax::cont::Error`. These are `dax::cont::ErrorExecution` and `dax::cont::ErrorControl`, and they represent errors that happen in the respective execution and control environments.

Readers familiar with parallel programming will probably note the difficulty in raising errors in multi-threaded execution like what happens in the execution environment. In fact some devices, like CUDA devices, do not support exceptions at all. Dax handles the error reporting in the execution environment by flagging an error when it occurs and then throwing an error in the control environment after all threads have terminated. This means that the amount of execution that happens after an error is flagged is indeterminate and any output values should be considered incorrect.

The `dax::cont::ErrorControl` class is also broken down into several subclasses that can be independently caught to handle different types of errors. The following control errors exist and may be thrown.

**`dax::cont::ErrorControlAssert`** Thrown when an assertion fails, meaning a Dax operation reached an unexpected state. The header file dax/cont/Assert.h defines a macro named `DAX_ASSERT_CONT` that behaves much like the POSIX C assert macro except that a `ErrorControlAssert` is thrown rather than killing the application outright.

**`dax::cont::ErrorControlBadValue`** Thrown when a Dax function or method encounters an invalid value that inhibits progress.

**`dax::cont::ErrorControlInternal`** Thrown when Dax detects an internal state that should never be reached. This error usually indicates a bug in Dax or, at best, Dax failed to detect an invalid input it should have.

**`dax::cont::ErrorControlOutOfMemory`** Thrown when a Dax function or method tries to allocate an array and fails.

### 3.4.7 Device Adapter Algorithms

The Dax toolkit comes with the templated class dax::cont::DeviceAdapterAlgorithm that provides a set of algorithms that can be invoked in the control environment and are run on the execution environment. The template has a single argument that specifies the device adapter tag.

Example 3.59: Prototype for dax::cont::DeviceAdapterAlgorithm.

```
namespace dax {
namespace cont {

template<class DeviceAdapterTag>
struct DeviceAdapterAlgorithm;

}
} // namespace dax::cont
```

DeviceAdapterAlgorithm contains no state. It only has a set of static methods that implement its algorithms. The following methods are available.

**Copy** Copies data from an input array to an output array. The copy takes place in the execution environment.

**LowerBounds** The LowerBounds method takes three arguments. The first argument is an ArrayHandle of sorted values. The second argument is another ArrayHandle of items to find in the first array. LowerBounds find the index of the first item that is greater than or equal to the target value, much like the std::lower_bound STL algorithm. The results are returned in an ArrayHandle given in the third argument.

There are two specializations of LowerBounds. The first takes an additional comparison function that defines the less-than operation. The second takes only two parameters. The first is an ArrayHandle of sorted dax::Ids and the second is an ArrayHandle of dax::Ids to find in the first list. The results are written back out to the second array. This second specialization is useful for inverting index maps.

**ScanInclusive** The ScanInclusive method takes an input and an output ArrayHandle and performs a running sum on the input array. The first value in the output is the same as the first value in the input. The second value in the output is the sum of the first two values in the input. The third value in the output is the sum of the first three values of the input, and so on. ScanInclusive returns the sum of all values in the input.

**ScanExclusive** The ScanExclusive method takes an input and an output ArrayHandle and performs a running sum on the input array. The first value in the output is always 0. The second value in the output is the same as the first value in the input. The third value in the output is the sum of the first two values in the input. The fourth value in the output is the sum of the first three values of the input, and so on. ScanExclusive returns the sum of all values in the input.

**Schedule** The Schedule method takes a functor as its first argument and invokes it a number of times specified by the second argument. It should be assumed that each invocation of Schedule occurs on a separate thread although in practice there could be some thread sharing.

There are two versions of the Schedule method. The first version takes a dax::Id and invokes the functor that number of times. The second version takes a dax::Id3 and invokes the functor once for every entry in a 3D array of the given dimensions.

The functor is expected to be an object with a const overloaded parentheses operator. The operator takes as a parameter the index of the invocation, which is either a dax::Id or a dax::Id3 depending on what version of Schedule is being used. The functor must also provide a method named SetErrorMessageBuffer that accepts an argument of type dax::exec::internal::ErrorMessageBuffer. If any errors occur during the invocations of the functor, it should call the RaiseError method of the ErrorMessageBuffer. That will cause the Schedule method to (eventually) throw a dax::cont::ErrorExecution exception.

**Sort** The Sort method provides an unstable sort of an array. There are two forms of the Sort method. The first takes an ArrayHandle and sorts the values in place. The second takes an additional argument that is a functor that provides the comparison operation for the sort.

**SortByKey** The SortByKey method works similarly to the Sort method except that it takes two ArrayHandles: an array of keys and a corresponding array of values. The sort orders the array of keys in ascending values and also reorders the values so they remain paired with the same key. Like Sort, SortByKey has a version that sorts by the default less-than operator and a version that accepts a custom comparison functor.

**StreamCompact** The StreamCompact method selectively removes values from an array. The first argument is an ArrayHandle to be compacted and the second argument is an ArrayHandle of equal size with flags indicating whether the corresponding input value is to be copied to the output. The third argument is an output ArrayHandle whose length is set to the number of true flags in the stencil and the passed values are put in order to the output array.

There is also a second form of StreamCompact that only has the stencil and output as arguments. In this version, the output gets the corresponding index of where the input should be taken from.

**Synchronize** The Synchronize method waits for any asynchronous operations running on the device to complete and then returns.

**Unique** The Unique method removes all duplicate values in an ArrayHandle. The method will only find duplicates if they are adjacent to each other in the array. The easiest way to ensure that duplicate values are adjacent is to sort the array first.

There are two versions of Unique. The first uses the equals operator to compare entries. The second accepts a binary functor to perform the comparisons.

**UpperBounds** The `UpperBounds` method takes three arguments. The first argument is an `ArrayHandle` of sorted values. The second argument is another `ArrayHandle` of items to find in the first array. `UpperBounds` find the index of the first item that is greater than to the target value, much like the `std::upper_bound` STL algorithm. The results are returned in an `ArrayHandle` given in the third argument.

There are two specializations of `UpperBounds`. The first takes an additional comparison function that defines the less-than operation. The second takes only two parameters. The first is an `ArrayHandle` of sorted `dax::Id`s and the second is an `ArrayHandle` of `dax::Id`s to find in the first list. The results are written back out to the second array. This second specialization is useful for inverting index maps.

### 3.4.8 Implementing Device Adapters

The Dax toolkit comes with several implementations of device adapters so that it may be ported to a variety of platforms. It is also possible to provide new device adapters to support yet more devices, compilers, and libraries. A new device adapter provides a tag, a class to manage arrays in the execution environment, a collection of algorithms that run in the execution environment, and (optionally) a timer.

Although not strictly necessary, the implementation of device adapters within the Dax toolkit are divided into 3 header files with the names DeviceAdapterTag∗.h, ArrayManagerExecution∗.h and DeviceAdapterAlgorithm∗.h. The DeviceAdapter∗.h that most code includes is a trivial header that simply includes these other three files. For example, the dax/tbb/cont/DeviceAdapterTBB.h for the Intel Threading Building Blocks (TBB) device adapter simply contains the following (with minutia like include guards removed).

Example 3.60: Contents of dax/tbb/cont/DeviceAdapterTBB.h file.

```
#include <dax/tbb/cont/internal/DeviceAdapterTagTBB.h>
#include <dax/tbb/cont/internal/ArrayManagerExecutionTBB.h>
#include <dax/tbb/cont/internal/DeviceAdapterAlgorithmTBB.h>
```

The reason the Dax toolkit breaks up the code for its device adapters this way is that there is an interdependence between the implementation of each device adapter and the mechanism to pick a default device adapter. Breaking up the device adapter code in this way maintains an acyclic dependence among header files.

#### Tag

The device adapter tag, as described in Section 3.4.1 is a simple empty type that is used as a template parameter to identify the device adapter. Every device adapter implementation provides one. The device adapter tag is typically defined in an internal header file with a prefix of DeviceAdapterTag. Here is the implementation for the TBB device adapter.

Example 3.61: Implementation of the TBB device adapter tag.

```
namespace dax {
namespace tbb {
namespace cont {

struct DeviceAdapterTagTBB {  };

}
}
} // namespace dax::tbb::cont
```

### Array Manager Execution

The Dax toolkit defines a template named `dax::cont::internal::ArrayManagerExecution` that is responsible for allocating memory in the execution environment and copying data between the control and execution environment. The execution array manager is typically defined in an internal header file with a prefix of ArrayManagerExecution.

Example 3.62: Prototype for `dax::cont::internal::ArrayManagerExecution`.

```
namespace dax {
namespace cont {
namespace internal {

template<typename T, class ArrayContainerControlTag, class DeviceAdapterTag>
class ArrayManagerExecution;

}
}
} // namespace dax::cont::internal
```

A device adapter must provide a partial specialization of `ArrayManagerExecution` for its device adapter tag. The implementation for `ArrayManagerExecution` is expected to manage the resources for a single array, and it must provide the following elements.

**`ValueType`** A `typedef` of the type for each item in the array. This is the same type as the first template argument.

**`PortalType`** The type of an array portal that can be used in the execution environment to access the array.

**`PortalConstType`** A read-only (const) version of `PortalType`.

**`GetNumberOfValues`** A method that returns the number of values stored in the array. The results are undefined if the data has not been loaded or allocated.

**`LoadDataForInput`** A method that takes an array portal in the control environment, allocates a large enough array in the execution environment, and copies the data into that array. The data in the execution array is not expected to be changed. The allocated array can later be accessed via the `GetPortalConst` method.

**LoadDataForInPlace** A method that takes an array portal in the control environment, allocates a large enough array in the execution environment, and copies the data into that array. The data in the execution array is expected to be read and changed. The allocated array can later be accessed via the `GetPortal` and `GetPortalConst` methods.

**AllocateArrayForOutput** A method that takes an array container and a size and allocates an array in the execution environment of the specified size. The initial memory is uninitialized and can be accessed via the `GetPortal` method. The container argument can be used to allocate data when the control and execution share arrays, but this argument is often ignored.

**RetrieveOutputData** This method takes an array container, allocates memory in the control environment, and copies data from the execution environment into it.

**CopyInto** This method takes an STL-compatible iterator and copies data from the execution environment into it.

**Shrink** A method that adjusts the size of the array in the execution environment to something that is a smaller size. All the data up to the new length must remain valid. Typically, no memory is actually reallocated. Instead, a different end is marked.

**GetPortal** A method that returns an array portal that can be used in the execution environment. The portal was defined in either `LoadDataForInPlace` or `AllocateArrayForOutput`.

**GetPortalConst** A method that returns a read-only (const) array portal that can be used in the execution environment. The portal was defined in one of the load or allocate methods.

**ReleaseResources** A method that frees any resources (typically memory) in the execution environment.

Specializations of this template typically take on one of two forms. If the control and execution environments have separate memory spaces, then this class behaves by copying memory in methods such as `PrepareForInput` and `RetrieveOutputData`. This might require creating buffers in the control environment to efficiently move data from control array portals.

However, if the control and execution environments share the same memory space, the execution array manager can, and should, delegate all of its operations to the `ArrayContainerControl` it is used with. The Dax toolkit comes with a class called `dax::cont::internal::ArrayManagerExecutionShareWithControl` that provides the implementation for an execution array manager that shares a memory space with the control environment. In this case, making the `ArrayManagerExecution` specialization be a trivial subclass is sufficient. For example, here is the implementation of `ArrayManagerExecution` for TBB.

Example 3.63: Specialization of ArrayManagerExecution for TBB.

```
#include <dax/tbb/cont/internal/DeviceAdapterTagTBB.h>

#include <dax/cont/internal/ArrayManagerExecution.h>
#include <dax/cont/internal/ArrayManagerExecutionShareWithControl.h>

namespace dax {
namespace cont {
namespace internal {

template <typename T, class ArrayContainerTag>
class ArrayManagerExecution
    <T, ArrayContainerTag, dax::tbb::cont::DeviceAdapterTagTBB>
    : public dax::cont::internal::ArrayManagerExecutionShareWithControl
        <T, ArrayContainerTag>
{
};

}
}
} // namespace dax::cont::internal
```

### Algorithms

A device adapter implementation must also provide a specialization of dax::cont::DeviceAdapterAlgorithm, which is documented in Section 3.4.7. The implementation for the device adapter algorithms is typically placed in a header file with a prefix of DeviceAdapterAlgorithm.

Although there are many methods in DeviceAdapterAlgorithms, it is seldom necessary to implement them all. Instead, the Dax toolkit comes with dax::cont::internal::DeviceAdapterAlgorithmGeneral that provides generic implementation for most of the required algorithms. By deriving the specialization of DeviceAdapterAlgorithm from DeviceAdapterAlgorithmGeneral, only the implementations for Schedule and Synchronize need to be implemented. All other algorithms can be derived from those.

That said, not all of the algorithms implemented in DeviceAdapterAlgorithmGeneral are optimized for all types of devices. Thus, it is worthwhile to provide algorithms optimized for the specific device when possible. In particular, it is best to provide specializations for the sort and scan algorithms.

The following example is a minimal implementation of the TBB device adapter algorithms. The actual version that comes with the Dax toolkit contains more enhancements.

Example 3.64: Abbreviated implementation of DeviceAdapterAlgorithm for TBB.

```
#include <dax/tbb/cont/internal/DeviceAdapterTagTBB.h>
#include <dax/tbb/cont/internal/ArrayManagerExecutionTBB.h>

#include <dax/cont/internal/DeviceAdapterAlgorithmGeneral.h>
#include <dax/exec/internal/IJKIndex.h>

#include <tbb/blocked_range.h>
#include <tbb/blocked_range3d.h>
#include <tbb/parallel_for.h>
```

```
namespace dax {
namespace cont {

template<>
struct DeviceAdapterAlgorithm<dax::tbb::cont::DeviceAdapterTagTBB> :
    dax::cont::internal::DeviceAdapterAlgorithmGeneral<
        DeviceAdapterAlgorithm<dax::tbb::cont::DeviceAdapterTagTBB>,
        dax::tbb::cont::DeviceAdapterTagTBB>
{
private:
  static const dax::Id TBB_GRAIN_SIZE = 128;

  template<class FunctorType>
  class ScheduleKernel
  {
  public:
    DAX_CONT_EXPORT ScheduleKernel(const FunctorType &functor)
      : Functor(functor)
    {  }

    DAX_CONT_EXPORT void SetErrorMessageBuffer(
        const dax::exec::internal::ErrorMessageBuffer &errorMessage)
    {
      this->ErrorMessage = errorMessage;
      this->Functor.SetErrorMessageBuffer(errorMessage);
    }

    DAX_EXEC_EXPORT
    void operator()(const ::tbb::blocked_range<dax::Id> &range) const {
      // The TBB device adapter causes array classes to be shared between
      // control and execution environment. This means that it is possible for
      // an exception to be thrown even though this is typically not allowed.
      // Throwing an exception from here is bad because there are several
      // simultaneous threads running. Get around the problem by catching the
      // error and setting the message buffer as expected.
      try
        {
        for (dax::Id index = range.begin(); index < range.end(); index++)
          {
          this->Functor(index);
          }
        }
      catch (dax::cont::Error error)
        {
        this->ErrorMessage.RaiseError(error.GetMessage().c_str());
        }
      catch (...)
        {
        this->ErrorMessage.RaiseError(
            "Unexpected error in execution environment.");
        }
    }
  private:
    FunctorType Functor;
    dax::exec::internal::ErrorMessageBuffer ErrorMessage;
  };

public:
  template<class FunctorType>
  DAX_CONT_EXPORT
  static void Schedule(FunctorType functor, dax::Id numInstances)
  {
    const dax::Id MESSAGE_SIZE = 1024;
    char errorString[MESSAGE_SIZE];
    errorString[0] = '\0';
    dax::exec::internal::ErrorMessageBuffer
        errorMessage(errorString, MESSAGE_SIZE);
```

```
    ScheduleKernel<FunctorType> kernel(functor);
    kernel.SetErrorMessageBuffer(errorMessage);

    ::tbb::blocked_range<dax::Id> range(0, numInstances, TBB_GRAIN_SIZE);

    ::tbb::parallel_for(range, kernel);

    if (errorMessage.IsErrorRaised())
      {
      throw dax::cont::ErrorExecution(errorString);
      }
  }

private:
  template<class FunctorType>
  class ScheduleKernelId3
  {
  public:
    DAX_CONT_EXPORT ScheduleKernelId3(const FunctorType &functor,
                                      const dax::Id3& dims)
      : Functor(functor),
        Dims(dims)
      {  }

    DAX_CONT_EXPORT void SetErrorMessageBuffer(
        const dax::exec::internal::ErrorMessageBuffer &errorMessage)
    {
      this->ErrorMessage = errorMessage;
      this->Functor.SetErrorMessageBuffer(errorMessage);
    }

    DAX_EXEC_EXPORT
    void operator()(const ::tbb::blocked_range3d<dax::Id> &range) const {
      try
        {
        dax::exec::internal::IJKIndex index(this->Dims);
        for( dax::Id k=range.pages().begin(); k!=range.pages().end(); ++k)
          {
          index.SetK(k);
          for( dax::Id j=range.rows().begin(); j!=range.rows().end(); ++j)
            {
            index.SetJ(j);
            for( dax::Id i=range.cols().begin(); i!=range.cols().end(); ++i)
              {
              index.SetI(i);
              this->Functor(index);
              }
            }
          }
        }
      catch (dax::cont::Error error)
        {
        this->ErrorMessage.RaiseError(error.GetMessage().c_str());
        }
      catch (...)
        {
        this->ErrorMessage.RaiseError(
            "Unexpected error in execution environment.");
        }
    }
  private:
    FunctorType Functor;
    dax::Id3 Dims;
    dax::exec::internal::ErrorMessageBuffer ErrorMessage;
  };

public:
```

```
  template<class FunctorType>
  DAX_CONT_EXPORT
  static void Schedule(FunctorType functor,
                       dax::Id3 rangeMax)
  {
    //we need to extract from the functor that uniform grid information
    const dax::Id MESSAGE_SIZE = 1024;
    char errorString[MESSAGE_SIZE];
    errorString[0] = '\0';
    dax::exec::internal::ErrorMessageBuffer
        errorMessage(errorString, MESSAGE_SIZE);

    //memory is generally setup in a way that iterating the first range
    //in the tightest loop has the best cache coherence.
    ::tbb::blocked_range3d<dax::Id> range(0, rangeMax[2],
                                          0, rangeMax[1],
                                          0, rangeMax[0]);

    ScheduleKernelId3<FunctorType> kernel(functor,rangeMax);
    kernel.SetErrorMessageBuffer(errorMessage);

    ::tbb::parallel_for(range, kernel);

    if (errorMessage.IsErrorRaised())
      {
      throw dax::cont::ErrorExecution(errorString);
      }
  }

  DAX_CONT_EXPORT static void Synchronize()
  {
    // Nothing to do. This device schedules all of its operations using a
    // split/join paradigm. This means that the if the control thread is
    // calling this method, then nothing should be running in the execution
    // environment.
  }

};

}
} // namespace dax::cont
```

### Timer Implementation

The Dax timer, described in Section 3.4.5, delegates to an internal class named `dax::cont::DeviceAdapterTimerImplementation`. The interface for this class is the same as that for `dax::cont::Timer`. A default implementation of this templated class uses the system timer and the `Synchronize` method in the device adapter algorithms.

However, some devices might provide alternate or better methods for implementing timers. For example, the TBB library comes with a high resolution timer that has better accuracy than the standard system timers. Thus, the device adapter can optionally provide a specialization of `DeviceAdapterTimerImplementation`, which is typically placed in the same header file as the device adapter algorithms.

The following example is the implementation of the TBB timer implementation.

Example 3.65: Implementation of `DeviceAdapterTimerImplementation` for TBB.

```
#include <dax/cont/DeviceAdapter.h>
#include <dax/tbb/cont/internal/DeviceAdapterTagTBB.h>

#include <tbb/tick_count.h>

namespace dax {
namespace cont {

template<>
class DeviceAdapterTimerImplementation<dax::tbb::cont::DeviceAdapterTagTBB>
{
public:
  DAX_CONT_EXPORT DeviceAdapterTimerImplementation()
  {
    this->Reset();
  }
  DAX_CONT_EXPORT void Reset()
  {
    dax::cont::DeviceAdapterAlgorithm<dax::tbb::cont::DeviceAdapterTagTBB>::Synchronize();
    this->StartTime = ::tbb::tick_count::now();
  }
  DAX_CONT_EXPORT dax::Scalar GetElapsedTime()
  {
    dax::cont::DeviceAdapterAlgorithm<dax::tbb::cont::DeviceAdapterTagTBB>::Synchronize();
    ::tbb::tick_count currentTime = ::tbb::tick_count::now();
    ::tbb::tick_count::interval_t elapsedTime = currentTime - this->StartTime;
    return static_cast<dax::Scalar>(elapsedTime.seconds());
  }

private:
  ::tbb::tick_count StartTime;
};

}
} // namespace dax::cont
```

A word of warning about implementing timers. Although `GetElapsedTime` returns a `dax::Scalar`, it is advisable to store the internal timing in its native data format until the elapsed time is recorded. This is because the times may be biased by a large value, and the floating point number might not hold enough precision to get a precise measurement between the start and end of the timer.

### Testing

The implementation of a device adapter contains many components. To ensure that all of its device adapters are working properly, the Dax toolkit contains a complete test of all the components in dax/cont/testing/TestingDeviceAdapter.h. Here is the implementation for the TBB device adapter test, which plugs into the CMake testing framework.

Example 3.66: Test code for the TBB device adapter.

```
#include <dax/tbb/cont/DeviceAdapterTBB.h>

#include <dax/cont/testing/TestingDeviceAdapter.h>

int UnitTestDeviceAdapterTBB(int, char *[])
{
  return dax::cont::testing::TestingDeviceAdapter
      <dax::tbb::cont::DeviceAdapterTagTBB>::Run();
```

```
}
```

## 3.5 Execution Environment

The execution environment is exposed to developers that write worklets for different visualization algorithms. In addition to providing all the mechanisms for building the worklet object itself, the execution environment contains supporting code that can be useful to the implementations of visualization algorithms.

The data structures in the execution environment provide information and operations for a single element. This is in contrast to the control environment, where data structures are built on arrays providing information for large collections of data. These respective data structures reflect the nature of the two environments. The control environment manages the stores of data whereas the execution environment performs large parallel processing through fine operations.

### 3.5.1 Creating Worklets

A worklet in Dax is most simply a functor that operates on an element of data. Thus, it is a `class` or `struct` that has an overloaded parenthesis operator (which must be declared `const` for thread safety). It also must inherit from one of the predefined abstract worklet classes, which will declare the correct dispatcher to use when invoking the worklet. Finally, it must declare a pair of *signatures* that define what information must be presented when invoking the worklet and how this information gets passed to each worklet invocation. Figure 3.3 demonstrates all of the required components of a worklet.

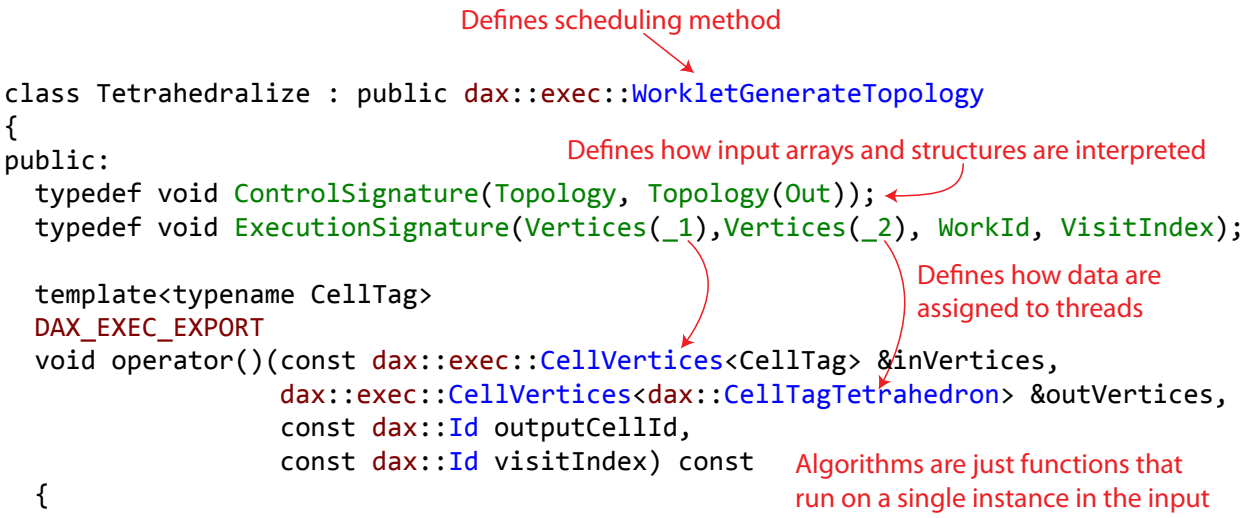



Figure 3.3: Annotated example of a worklet declaration.

### Control Signature

The control signature of a worklet is the `typedef` of a function prototype named `ControlSignature`. The function prototype matches the calling specification used with the dispatcher invoke function.

The return type of the function prototype is always `void` because the dispatcher invoke functions do not return values. The parameters of the function prototype are *tags* that identify the type of data that is expected to be passed to invoke. For example, a `Field` tag declares that the worklet will operate on field data, typically held in a `dax::cont::ArrayHandle` whereas a `Topology` tag declares that the worklet will operate on a grid structure like those documented in Section 3.4.3.

Tags can also have modifiers on them, which are attached with parenthesis. For example, a `Field` can be declared as either `Field(Point)` or `Field(Cell)`. Likewise, a field could be modified to be either `In` (the default) or `Out`.

The signature tags and their modifiers are described in greater detail in the following section on worklet types.

### Execution Signature

Like the control signature, the execution signature of a worklet is the `typedef` of a function prototype named `ExecutionSignature`. The function prototype must match the parenthesis operator in terms of arity and argument semantics.

The arguments of the `ExecutionSignature`'s function prototype are tags that define where the data comes from. The most common tags are an underscore followed by a number, such as `_1`, `_2`, etc. These numbers refer back to the corresponding argument in the `ControlSignature`. For example, `_1` means data from the first control signature argument, `_2` means data from the second control signature argument, etc.

Unlike the control signature, the execution signature optionally can declare a return type if the parenthesis operator returns a value. If this is the case, the return value should be one of the numeric tags (i.e. `_1`, `_2`, etc.) to refer to one of the data structures of the control signature. If the parenthesis operator does not return a value, then `ExecutionSignature` should declare the return type as `void`.

In addition to the numeric tags, there are other execution signature tags to represent other types of data. For example, the `WorkId` tag identifies the instance of the worklet invocation. Each call to the worklet function will have a unique `WorkId`. Other such tags exist and are described in the following section on worklet types where appropriate.

### Worklet Types

There are multiple worklet types provided by the Dax toolkit, each designed to support a particular type of operation. This section will define each of the worklet types, identify the generic superclass that a worklet instance should derive, identify the signature tags and their meanings, and give an example of the worklet in use.

**Field Map** A worklet deriving `dax::exec::WorkletMapField` performs a basic mapping operation that applies a function (the operator in the worklet) on all the field values at a single point or cell and creates a new field value at that same location. Although the intention is to operate on some variable over the mesh, a `dax::exec::WorkletMapField` actually be applied to any array.

A `WorkletMapField` subclass is invoked with a `dax::cont::DispatcherMapField`. This dispatcher has two template arguments. The first argument is the type of the worklet subclass. The second argument, which is optional, is a device adapter tag.

A field map worklet has only one type of `ControlSignature` tag: `Field`. This tag corresponds to a `dax::cont::ArrayHandle` passed to invoke, and each invocation of the worklet gets or sets a single value in this array. The `Field` tag can be modified to be either `In` (the default) or `Out`.

A field map worklet supports the standard tags in its `ExecutionSignature`. These are the numeric tags (e.g. `_1`, `_2`, etc.) and the `WorkId` tag, which uniquely identifies the invocation instance of the worklet.

Field maps most commonly perform basic calculator arithmetic, as demonstrated in the following example.

Example 3.67: Declaration and use of a field map worklet.

```
#include <dax/exec/WorkletMapField.h>
#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherMapField.h>
#include <dax/math/VectorAnalysis.h>

class Magnitude : public dax::exec::WorkletMapField
{
public:
  typedef void ControlSignature(Field(In), Field(Out));
  typedef void ExecutionSignature(_1,_2);

  DAX_EXEC_EXPORT
  void operator()(const dax::Vector3 &inValue,
                  dax::Scalar &outValue) const
  {
    outValue = dax::math::Magnitude(inValue);
  }
};

DAX_CONT_EXPORT
dax::cont::ArrayHandle<dax::Scalar>
InvokeMagnitude(dax::cont::ArrayHandle<dax::Vector3> input)
{
```

```
  dax::cont::ArrayHandle<dax::Scalar> output;

  dax::cont::DispatcherMapField<Magnitude> dispatcher;
  dispatcher.Invoke(input, output);

  return output;
}
```

**Cell Map** A worklet deriving `dax::exec::WorkletMapCell` performs a mapping operation that applies a function (the operator in the worklet) on field values on a single cell and creates a new field value at that location. The function has access to all field values local to that cell. So if an input is a point field, the operation will have access to all the values of that field.

A `WorkletMapCell` subclass is invoked with a `dax::cont::DispatcherMapCell`. This dispatcher has two template arguments. The first argument is the type of the worklet subclass. The second argument, which is optional, is a device adapter tag.

A cell map worklet supports the following tags in the parameters of its `ControlSignature`.

`Topology` This tag corresponds to one of the grid structures described in Section 3.4.3 passed to invoke that holds the topology on which to apply the map.

If the `Topology` argument is referenced with a numeric tag in the `ExecutionSignature` (e.g. with `_1`), then the worklet operator receives the cell-type tag (such as `dax::CellTagTriangle` or `dax::CellTagVoxel`). This is sometimes useful for specializing based on the cell type, but usually unnecessary.

If the `Topology` argument is referenced by a `Vertices` tag wrapping a numeric tag (e.g. with `Vertices(_1)`), then the worklet function is passed a `dax::exec::CellVertices` object that contains the point indices for all the vertices of the cell.

`Field` This tag corresponds to a `dax::cont::ArrayHandle` passed to invoke that holds the sample values for a field at all points or all cells. The `Field` tag can be modified to be either `In` (the default) or `Out`. Input `Field` tags can be further modified to be attached to `Point`s or `Cell`s. The size of the input `dax::cont::ArrayHandle` must match the number of points or cells in the grid structure passed in as a `Topology` argument. Output fields are always attached to the cells, and the corresponding `dax::cont::ArrayHandle` will be resized as necessary.

A cell field has a one-to-one mapping between `dax::cont::ArrayHandle` entries and worklet function parameters. Thus, when the `ExecutionSignature` references a `ControlSignature Field` parameter (e.g. with `_2`), the parameter is the same as the basic type as the values in the array (typically something like `dax::Scalar` or `dax::Vector3`).

A point field has a many-to-one mapping between `dax::cont::ArrayHandle` entries and worklet function parameters because each cell can touch multiple points. So when

a dax::cont::ArrayHandle is translated to the worklet invocation, its values get passed in a dax::exec::CellField object, which behaves like a dax::Tuple with a size matching the number of vertices in a cell.

A cell map worklet supports the following tags in the parameters of its ExecutionSignature.

_1, _2,... These reference the corresponding parameter in the ControlSignature.

Vertices When modified by one of the numeric tags (e.g. Vertices(_1)), passes a dax::exec::CellVertices to the worklet representing the point indices for each vertex of the cell.

WorkId Produces a dax::Id that uniquely identifies the invocation instance of the worklet.

Cell maps most commonly perform operations on interpolated fields. They often use the cell operations provided by the Dax toolkit and described in Section 3.5.4. The following example shows a cell map worklet that simply averages all the point values it touches. A more serious worklet would probably perform interpolations, derivatives, or integrations over the cell.

Example 3.68: Declaration and use of a cell map worklet.

```
#include <dax/exec/WorkletMapCell.h>

#include <dax/exec/CellField.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherMapCell.h>
#include <dax/cont/UniformGrid.h>

class CellAverage : public dax::exec::WorkletMapCell
{
public:
  typedef void ControlSignature(Topology, Field(Point), Field(Out));
  typedef _3 ExecutionSignature(_1,_2);

  template<class CellTag>
  DAX_EXEC_EXPORT
  dax::Scalar operator()(
    CellTag, const dax::exec::CellField<dax::Scalar,CellTag> &values) const
  {
    dax::Scalar sum = values[0];
    for (int index = 1; index < values.NUM_VERTICES; index++)
      {
      sum += values[index];
      }
    return sum/values.NUM_VERTICES;
  }
};

DAX_CONT_EXPORT
void InvokeCellAverage()
{
  const dax::Id DIM = 100;

  // Make a grid structure.
```

```
  dax::cont::UniformGrid<> grid;
  grid.SetExtent(dax::make_Id3(0, 0, 0), dax::make_Id3(DIM-1, DIM-1, DIM-1));

  // Make input.
  // (A real application would make more interesting data and do it more efficiently.)
  dax::Scalar inputBuffer[DIM*DIM*DIM];
  for (dax::Id index = 0; index < DIM*DIM*DIM; index++)
    {
    inputBuffer[index] = index;
    }
  dax::cont::ArrayHandle<dax::Scalar> input =
      dax::cont::make_ArrayHandle(inputBuffer, DIM*DIM*DIM);

  dax::cont::ArrayHandle<dax::Scalar> output;

  dax::cont::DispatcherMapCell<CellAverage> dispatcher;
  dispatcher.Invoke(grid, input, output);

  // Do something with output.
}
```

**Generate Topology** A worklet deriving from dax::exec::WorkletGenerateTopology generates a cell connectivity. When invoked, the dispatcher is given an array containing the number of output cells derived from each input cell. Each invocation of a dax::exec::WorkletGenerateTopology produces exactly one cell, so the dispatcher then invokes the generate topology worklet multiple times per cell if multiple cells are derived.

A WorkletGenerateTopology subclass is invoked with a dax::cont::DispatcherGenerateTopology. This dispatcher has three template arguments. The first argument is the type of the worklet subclass. The second argument is a type of array handle (defaults to dax::cont::ArrayHandle<dax::Id>) that holds the count of cells to be generated per input value. The third argument, which is optional, is a device adapter tag.

Generate topology operations are used when one topology is derived from another's points. A generate topology is often proceeded by a field map or cell map that counts how many cells will be derived from each input cell. These counts are stored in an array and passed to the dax::cont::DispatcherGenerateTopology that invokes the worklet.

A generate topology worklet supports the following tags in the parameters of its ControlSignature.

Topology This tag corresponds to one of the grid structures described in Section 3.4.3 passed to invoke that holds the topology on which to derive a new topology or to write the new topology into. The Topology tag can be modified to be either In (the default) or Out.

If the Topology argument is referenced with a numeric tag in the ExecutionSignature (e.g. with _1), then the worklet operator receives the cell-type tag (such as dax::CellTagTriangle or dax::CellTagVoxel). This is sometimes useful for specializing based on the cell type, but usually unnecessary.

If the Topology argument is referenced by a Vertices tag wrapping a numeric tag (e.g.

with Vertices(_1)), then the worklet function is passed a dax::exec::CellVertices object that contains the point indices for all the vertices of the cell.

**Field** This tag corresponds to a dax::cont::ArrayHandle passed to invoke that holds the sample values for a field at all points or all cells. All fields for generate topology worklets are input. The Field tag can be modified to be attached to Points or Cells. The size of the dax::cont::ArrayHandle must match the number of points or cells in the grid structure passed in as a Topology argument.

A cell field has a one-to-one mapping between dax::cont::ArrayHandle entries and worklet function parameters. Thus, when the ExecutionSignature references a ControlSignature Field parameter (e.g. with _2), the parameter is the same as the basic type as the values in the array (typically something like dax::Scalar or dax::Vector3).

A point field has a many-to-one mapping between dax::cont::ArrayHandle entries and worklet function parameters because each cell can touch multiple points. So when a dax::cont::ArrayHandle is translated to the worklet invocation, its values get passed in a dax::exec::CellField object, which behaves like a dax::Tuple with a size matching the number of vertices in a cell.

A generate topology worklet supports the following tags in the parameters of its ExecutionSignature.

**_1, _2,...** These reference the corresponding parameter in the ControlSignature.

**Vertices** When modified by one of the numeric tags (e.g. Vertices(_1)), passes a dax::exec::CellVertices to the worklet representing the point indices for each vertex of the cell. The numeric tag must point to a ControlSignature parameter of type Topology.

**VisitId** Produces a dax::Id that uniquely identifies the invocation instance for the particular cell being visited. For example, if dividing hexahedra into tetrahedra, each hexahedra produces 5 or 6 tetrahedra, but each invocation of the generate topology worklet generates just one of this. The VisitId identifies which of the tetrahedra to produce.

**WorkId** Produces a dax::Id that uniquely identifies the invocation instance of the worklet.

The following example converts a uniform grid of voxels into the a collection of quadrilaterals that make up the faces. The worklet leverages the implicit topology of a uniform grid to ensure that each face is represented exactly once.

Example 3.69: Declaration and use of a generate topology worklet.

```
#include <dax/exec/WorkletGenerateTopology.h>

#include <dax/Extent.h>
```

```
#include <dax/exec/CellVertices.h>
#include <dax/exec/WorkletMapCell.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherGenerateTopology.h>
#include <dax/cont/DispatcherMapCell.h>
#include <dax/cont/UniformGrid.h>
#include <dax/cont/UnstructuredGrid.h>

DAX_EXEC_CONSTANT_EXPORT
const unsigned char VoxelFaces[6][4] = {
  { 0, 3, 7, 4 },
  { 0, 4, 5, 1 },
  { 0, 1, 2, 3 },
  { 1, 2, 6, 5 },
  { 2, 3, 7, 6 },
  { 4, 5, 6, 7 }
};

class CountFaceOut : public dax::exec::WorkletMapCell
{
public:
  typedef void ControlSignature(Topology, Field(Out), Field(Out));
  typedef _3 ExecutionSignature(WorkId, _2);

  DAX_CONT_EXPORT
  CountFaceOut(dax::Id3 dimensions) : Dimensions(dimensions) {  }

  DAX_EXEC_EXPORT
  dax::Id operator()(dax::Id workId, dax::Tuple<unsigned char,6> &faceToOutput) const
  {
    dax::Id3 index3D;
    dax::Id flatIndex = workId;
    for (i = 0; i < 3; i++)
      {
      index3D[i] = flatIndex % this->Dimensions[i];
      flatIndex /= this->Dimensions[i];
      }

    dax::Id count = 0;
    // First three faces output on all cells.
    faceToOutput[count] = 0;  count++;
    faceToOutput[count] = 1;  count++;
    faceToOutput[count] = 2;  count++;

    // Second three faces output only on cells at maximum boundary.
    if (flatIndex[0] == this->Dimensions[0]-1) { faceToOutput[count] = 3;  count++; }
    if (flatIndex[1] == this->Dimensions[1]-1) { faceToOutput[count] = 4;  count++; }
    if (flatIndex[2] == this->Dimensions[2]-1) { faceToOutput[count] = 5;  count++; }

    return count;
  }

private:
  dax::Id3 Dimensions;
};

class ExtractFace : public dax::exec::WorkletGenerateTopology
{
public:
  typedef void ControlSignature(Topology, Topology(Out), Field);
  typedef void ExecutionSignature(Vertices(_1), Vertices(_2), _3, VisitIndex);

  DAX_EXEC_EXPORT
  void operator()(const dax::exec::CellVertices<dax::CellTagVoxel> &inVertices,
                  dax::exec::CellVertices<dax::CellTagQuadrilateral> &outVertices,
                  const dax::Tuple<unsigned char,6> outputFaces,
                  dax::Id visitIndex) const
```

```
  {
    unsigned char faceId = outputFaces[visitIndex];
    outVertices[0] = inVertices[VoxelFaces[faceId][0]];
    outVertices[1] = inVertices[VoxelFaces[faceId][1]];
    outVertices[2] = inVertices[VoxelFaces[faceId][2]];
    outVertices[3] = inVertices[VoxelFaces[faceId][3]];
  }
};

DAX_CONT_EXPORT
dax::cont::UnstructuredGrid<dax::CellTagQuadrilateral>
InvokeExtraceFaces(const dax::cont::UniformGrid<> &inputGrid)
{
  dax::Id3 dimensions = dax::extentCellDimensions(inputGrid.GetExtent());

  dax::cont::ArrayHandle<dax::Tuple<unsigned char,6> > faces;
  dax::cont::ArrayHandle<dax::Id> counts;

  dax::cont::DispatcherMapCell<CountFaceOut> countDispatcher(CountFaceOut(dimensions));
  countDispatcher.Invoke(inputGrid, faces, counts);

  dax::cont::UnstructuredGrid<dax::CellTagQuadrilateral> outputGrid;

  dax::cont::DispatcherGenerateTopology<ExtractFace> extractFaceDispatcher(counts);
  extractFaceDispatcher.SetRemoveDuplicatePoints(false); // All points will be used.
  extractFaceDispatcher.Invoke(inputGrid, outputGrid, faces);

  return outputGrid;
}
```

**Interpolated Cell** A worklet deriving from `dax::exec::WorkletInterpolatedCell` generates a new geometry comprising both new points at new coordinates and cell connections among those points. When invoked, the dispatcher is given an array containing the number of cells produced. (The cell type must be homogeneous.) Each invocation of a `dax::exec::WorkletInterpolatedCell` produces exactly one cell and its associated points, so the dispatcher then invokes the interpolated cell worklet multiple times per cell if multiple cells are derived.

A `WorkletInterpolatedCell` subclass is invoked with a `dax::cont::DispatcherInterpolatedCell`. This dispatcher has three template arguments. The first argument is the type of the worklet subclass. The second argument is a type of array handle (defaults to `dax::cont::ArrayHandle<dax::Id>`) that holds the count of cells to be generated per input value. The third argument, which is optional, is a device adapter tag.

Interpolated cell operations are used when one topology is derived from another, but the new topology can build cells in unconstrained ways. An interpolated cell is often proceeded by a field map or cell map that counts how many cells will be derived from each input cell. These counts are stored in an array and passed to the `dax::cont::DispatcherInterpolatedCell` that invokes the worklet.

An interpolated cell worklet supports the following tags in the parameters of its `ControlSignature`.

`Topology` This tag corresponds to one of the grid structures described in Section 3.4.3 passed

to invoke that holds the topology on which to derive a new topology.

If the Topology argument is referenced with a numeric tag in the ExecutionSignature (e.g. with _1), then the worklet operator receives the cell-type tag (such as dax::CellTagTriangle or dax::CellTagVoxel). This is sometimes useful for specializing based on the cell type, but usually unnecessary.

If the Topology argument is referenced by a Vertices tag wrapping a numeric tag (e.g. with Vertices(_1)), then the worklet function is passed a dax::exec::CellVertices object that contains the point indices for all the vertices of the cell.

**Geometry** This tag corresponds to one of the grid structures described in Section 3.4.3 passed to invoke. Parameters of this type access the full geometry of the grid including both point locations and cell connections. The Geometry tag is always used in the output of an interpolated cell worklet, and so should be modified with Out.

When the ExecutionSignature references a ControlSignature Geometry parameter (e.g. with _2), the parameter is a dax::exec::InterpolatedCellPoints object. The worklet operator should pass this parameter by reference so that it may be filled and the results returned.

**Field** This tag corresponds to a dax::cont::ArrayHandle passed to invoke that holds the sample values for a field at all points or all cells. All fields for generate topology worklets are input. The Field tag can be modified to be attached to Points or Cells. The size of the dax::cont::ArrayHandle must match the number of points or cells in the grid structure passed in as a Topology argument.

A cell field has a one-to-one mapping between dax::cont::ArrayHandle entries and worklet function parameters. Thus, when the ExecutionSignature references a ControlSignature Field parameter (e.g. with _2), the parameter is the same as the basic type as the values in the array (typically something like dax::Scalar or dax::Vector3).

A point field has a many-to-one mapping between dax::cont::ArrayHandle entries and worklet function parameters because each cell can touch multiple points. So when a dax::cont::ArrayHandle is translated to the worklet invocation, its values get passed in a dax::exec::CellField object, which behaves like a dax::Tuple with a size matching the number of vertices in a cell.

An interpolated cell worklet supports the following tags in the parameters of its ExecutionSignature.

**_1, _2,...** These reference the corresponding parameter in the ControlSignature.

**Vertices** When modified by one of the numeric tags (e.g. Vertices(_1)), passes a dax::exec::CellVertices to the worklet representing the point indices for each vertex of the cell. The numeric tag must point to a ControlSignature parameter of type Topology.

`VisitId` Produces a `dax::Id` that uniquely identifies the invocation instance for the particular cell being visited. For example, if dividing hexahedra into tetrahedra, each hexahedra produces 5 or 6 tetrahedra, but each invocation of the generate topology worklet generates just one of this. The `VisitId` identifies which of the tetrahedra to produce.

`WorkId` Produces a `dax::Id` that uniquely identifies the invocation instance of the worklet.

The following example performs a slice on a uniform grid using a plane that is aligned with the x axis (parallel with the y-z plane). With these constraints, we know that the intersection of every cell will be a quadrilateral.

Example 3.70: Declaration and use of an interpolated cell worklet.

```
#include <dax/exec/WorkletInterpolatedCell.h>

#include <dax/exec/CellField.h>
#include <dax/exec/CellVertices.h>
#include <dax/exec/InterpolatedCellPoints.h>
#include <dax/exec/WorkletMapCell.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherInterpolatedCell.h>
#include <dax/cont/DispatcherMapCell.h>
#include <dax/cont/UniformGrid.h>
#include <dax/cont/UnstructuredGrid.h>

class CountXSliceOut : public dax::exec::WorkletMapCell
{
public:
  typedef void ControlSignature(Topology, Field(Point), Field(Out));
  typedef _3 ExecutionSignature(_2);

  DAX_CONT_EXPORT
  CountXSliceOut(dax::Scalar xIntercept) : XIntercept(xIntercept) {  }

  DAX_EXEC_EXPORT
  dax::Id operator()(
      const dax::exec::CellField<dax::Vector3, dax::CellTagVoxel> &pointCoordinates) const
  {
    dax::Scalar minX = pointCoordinates[0][0];
    dax::Scalar maxX = pointCoordinates[6][0];

    return ((minX <= this->XIntercept) && (this->XIntercept < maxX)) ? 1 : 0;
  }

private:
  dax::Scalar XIntercept;
};

class XSlice : public dax::exec::WorkletInterpolatedCell
{
public:
  typedef void ControlSignature(Topology, Geometry(Out), Field(Point));
  typedef void ExecutionSignature(Vertices(_1), _2, _3);

  DAX_CONT_EXPORT
  XSlice(dax::Scalar xIntercept) : XIntercept(xIntercept) {  }

  DAX_EXEC_EXPORT
  void operator()(
      const dax::exec::CellVertices<dax::CellTagVoxel> &inVertices,
      dax::exec::InterpolatedCellPoints<dax::CellTagQuadrilateral> &outVertices,
```

```
      const dax::exec::CellField<dax::Vector3, dax::CellTagVoxel> &pointCoordinates) const
  {
    dax::Scalar minX = pointCoordinates[0][0];
    dax::Scalar maxX = pointCoordinates[6][0];
    dax::Scalar interpolant = (this->XIntercept - minX)/(maxX - minX);

    outVertices.SetInterpolationPoint(0, inVertices[0], inVertices[1], interpolant);
    outVertices.SetInterpolationPoint(1, inVertices[3], inVertices[2], interpolant);
    outVertices.SetInterpolationPoint(2, inVertices[4], inVertices[5], interpolant);
    outVertices.SetInterpolationPoint(3, inVertices[7], inVertices[6], interpolant);
  }

private:
  dax::Scalar XIntercept;
};

DAX_CONT_EXPORT
dax::cont::UnstructuredGrid<dax::CellTagQuadrilateral>
InvokeXSlice(const dax::cont::UniformGrid<> &inputGrid, dax::Scalar xIntercept)
{
  dax::cont::ArrayHandle<dax::Id> counts;

  dax::cont::DispatcherMapCell<CountXSliceOut> countDispatcher(CountXSliceOut(xIntercept));
  countDispatcher.Invoke(inputGrid, inputGrid.GetPointCoordinates(), counts);

  dax::cont::UnstructuredGrid<dax::CellTagQuadrilateral> outputGrid;

  dax::cont::DispatcherInterpolatedCell<XSlice> sliceDispatcher(count, XSlice(xIntercept));
  sliceDispatcher.SetRemoveDuplicatePoints(true);
  sliceDispatcher.Invoke(inputGrid, outputGrid, inputGrid.GetPointCoordinates());

  return outputGrid;
}
```

**Generate Keys and Values** A worklet deriving from dax::exec::WorkletGenerateKeysValues, which is designed to be used in conjunction with the reduce keys and values worklet, is an experimental type of worklet that can be applied to a variety of visualization algorithms. They allow an algorithm with a lot of interdependence to operate with lots of concurrency by storing and deferring the interdependent operation.

In operation the generate keys and values worklet works very much like a cell map worklet except that it is able to produce a variable amount of field values per cell. Each invocation of a dax::exec::WorkletGenerateKeysValues generates one set of keys and values, so the dispatcher then invokes the worklet multiple times per cell if multiple key-value sets are needed.

A WorkletGenerateKeysValues subclass is invoked with a dax::cont::DispatcherGenerateKeysValues. This dispatcher has three template arguments. The first argument is the type of the worklet subclass. The second argument is a type of array handle (defaults to dax::cont::ArrayHandle<dax::Id>) that holds the count of cells to be generated per input value. The third argument, which is optional, is a device adapter tag.

When invoking, the dispatcher needs to know how many key-values to produce per cell. These counts are stored in an array and passed to the dax::cont::DispatcherGenerateKeysValues that invokes the worklet. If all cells produced the same number of key-values,

then the implicit dax::cont::ArrayHandleConstant can be used.

Although the dax::exec::WorkletGenerateKeysValues worklet is expected to generate keys and values which have distinct semantics, the worklet itself does not distinguish between them. Instead, both keys and values are simply considered output fields.

A generate keys and values worklet supports the following tags in the parameters of its ControlSignature.

**Topology** This tag corresponds to one of the grid structures described in Section 3.4.3 passed to invoke that holds the topology on which to apply the map.

If the Topology argument is referenced with a numeric tag in the ExecutionSignature (e.g. with _1), then the worklet operator receives the cell-type tag (such as dax::CellTagTriangle or dax::CellTagVoxel). This is sometimes useful for specializing based on the cell type, but usually unnecessary.

If the Topology argument is referenced by a Vertices tag wrapping a numeric tag (e.g. with Vertices(_1)), then the worklet function is passed a dax::exec::CellVertices object that contains the point indices for all the vertices of the cell.

**Field** This tag corresponds to a dax::cont::ArrayHandle passed to invoke that holds the sample values for a field at all points or all cells. The Field tag can be modified to be either In (the default) or Out. Input Field tags can be further modified to be attached to Points or Cells. The size of the input dax::cont::ArrayHandle must match the number of points or cells in the grid structure passed in as a Topology argument. Output fields are always attached to the cells, and the corresponding dax::cont::ArrayHandle will be resized as necessary.

A cell field has a one-to-one mapping between dax::cont::ArrayHandle entries and worklet function parameters. Thus, when the ExecutionSignature references a ControlSignature Field parameter (e.g. with _2), the parameter is the same as the basic type as the values in the array (typically something like dax::Scalar or dax::Vector3).

A point field has a many-to-one mapping between dax::cont::ArrayHandle entries and worklet function parameters because each cell can touch multiple points. So when a dax::cont::ArrayHandle is translated to the worklet invocation, its values get passed in a dax::exec::CellField object, which behaves like a dax::Tuple with a size matching the number of vertices in a cell.

A generate keys and values worklet supports the following tags in the parameters of its ExecutionSignature.

**_1, _2,...** These reference the corresponding parameter in the ControlSignature.

**Vertices** When modified by one of the numeric tags (e.g. Vertices(_1)), passes a dax::exec::CellVertices to the worklet representing the point indices for each vertex of the cell.

`VisitId` Produces a `dax::Id` that uniquely identifies the invocation instance for the particular cell being visited. For example, if performing an operation on all cell values incident to a point, these values can be collected by generating keys on the point index. Each cell will generate one key-value per vertex, and the `VisitId` identifies which of the vertices to key on.

`WorkId` Produces a `dax::Id` that uniquely identifies the invocation instance of the worklet.

An example of defining and using a generate keys and values worklet is given in the next section in conjunction with a reduce keys and values worklet.

**Reduce Keys and Values** A worklet deriving from `dax::exec::WorkletReduceKeysValues` is an experimental type of worklet that can be applied to a variety of visualization algorithms. They allow an algorithm with a lot of interdependence to operate with lots of concurrency by storing and deferring the interdependent operation.

A `WorkletReduceKeysValues` subclass is invoked with a `dax::cont::DispatcherReduceKeysValues`. This dispatcher has three template arguments. The first argument is the type of the worklet subclass. The second argument is a type of array handle (defaults to `dax::cont::ArrayHandle<dax::Id>`) that holds the count of cells to be generated per input value. The third argument, which is optional, is a device adapter tag.

When invoking a `dax::exec::WorkletReduceKeysValues`, the dispatcher groups values based on their associated keys and calls a single instance of the worklet for every unique key given. The keys are given to the dispatcher. The values are passed as parameters and are automatically grouped by key before being passed to the worklet.

A reduce keys and values worklet supports only one tags in the parameters of its `ControlSignature`: `Value`. A `Value` corresponds to a `dax::cont::ArrayHandle` passed into the invoke method. The `Value` tag can be modified to be either `In` or `Out`. The semantics of the input and output values are a bit different.

A `Value(In)` corresponds to an input `dax::cont::ArrayHandle` with the same number of entries as there are keys. This type of parameter must be referenced in the `ExecutionSignature` using the `KeyGroup` tag modified by the numeric tag (for example, `KeyGroup(_1)`). The values of the group are passed in through a `dax::exec::KeyGroup` object. A `KeyGroup` object has a `GetNumberOfValues` method that returns the number of values in the group and a `Get` method that retrieves the value with a given group index. `KeyGroup` objects also have an overloaded bracket operator so that they can be referenced like an array or tuple.

A `Value(Out)` corresponds to an output `dax::cont::ArrayHandle`. The dispatcher will resize this array to the number of unique keys, and each instance of the worklet produces one entry into this array. This type of parameter is referenced in the `ExecutionSignature` simply with a numeric tag (such as `_2`).

The following example shows a pair of generate keys-values and reduce keys-values worklets that, for each point, simply averages all the cell values it touches.

Example 3.71: Declaration and use of generation and reduction of keys and values.

```
#include <dax/exec/WorkletGenerateKeysValues.h>
#include <dax/exec/WorkletReduceKeysValues.h>

#include <dax/CellTraits.h>

#include <dax/exec/CellVertices.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/ArrayHandleConstant.h>
#include <dax/cont/DispatcherGenerateKeysValues.h>
#include <dax/cont/DispatcherReduceKeysValues.h>
#include <dax/cont/UnstructuredGrid.h>

class PointAverageGenerateKeys : public dax::exec::WorkletGenerateKeysValues
{
public:
  typedef void ControlSignature(Topology, Field(Cell), Field(Out), Field(Out));
  typedef void ExecutionSignature(Vertices(_1), _2, _3, _4, VisitIndex);

  template<typename CellTag>
  DAX_EXEC_EXPORT
  void operator()(const dax::exec::CellVertices<CellTag> &cellVertices,
                  dax::Scalar fieldValue,
                  dax::Id &outKey,
                  dax::Scalar &outValue,
                  dax::Id visitIndex) const
  {
    outKey = cellVertices[visitIndex];
    outValue = fieldValue;
  }
};

class PointAverageReduceKeys : public dax::exec::WorkletReduceKeysValues
{
public:
  typedef void ControlSignature(Values(In), Values(Out));
  typedef _2 ExecutionSignature(KeyGroup(_1));

  template<typename KeyGroupType>
  DAX_EXEC_EXPORT
  dax::Scalar operator()(KeyGroupType keyGroup) const
  {
    dax::Scalar sum = keyGroup[0];
    for (dax::Id index = 1; index < keyGroup.GetNumberOfValues(); index++)
      {
      sum += keyGroup[index];
      }
    return sum/keyGroup.GetNumberOfValues();
  }
};

template<typename CellTag>
DAX_CONT_EXPORT
dax::cont::ArrayHandle<dax::Scalar>
InvokePointAverage(dax::cont::UnstructuredGrid<CellTag> grid,
                   dax::cont::ArrayHandle<dax::Scalar> inputCellField)
{
  typedef dax::cont::ArrayHandleConstant<dax::Id> CountArrayType;
  CountArrayType counts(dax::CellTraits<CellTag>::NUM_VERTICES, grid.GetNumberOfCells());

  dax::cont::ArrayHandle<dax::Id> keys;
  dax::cont::ArrayHandle<dax::Scalar> values;
```

```
  dax::cont::DispatcherGenerateKeysValues<PointAverageGenerateKeys,CountArrayType>
      generateKeysDispatcher(counts);
  generateKeysDispatcher.Invoke(grid, inputCellField, keys, values);

  dax::cont::ArrayHandle<dax::Scalar> outputPointField;

  dax::cont::DispatcherReduceKeysValues<
      PointAverageReduceKeys,dax::cont::ArrayHandle<dax::Scalar> >
        reduceKeysDispatcher(keys);
  reduceKeysDispatcher.Invoke(values, outputPointField);

  return outputPointField;
}
```

### Execution Objects

In the previous discussion, there is one `ControlSignature` tag that is available in all types of worklets that has not been mentioned: the `ExecObject` tag. The `ExecObject` means that the corresponding parameter to the dispatcher's invoke method will be an execution object. It is an object that is passed directly to every invocation of the worklet. Execution objects are helpful for implementing search structures and lookup tables. They also provide a mechanism for implementing features not yet available in the Dax toolkit.

The execution object must be a subclass of `dax::exec::ExecutionObjectBase`, and the instance is passed to all invocations of the worklet. This means that the execution object must be possible to copy the object from the control environment to the execution environment. It also means that any method used in the worklet must be declared with `DAX_EXEC_EXPORT` or `DAX_EXEC_CONT_EXPORT`.

An execution object can refer to an array, but the array reference must be through an array portal for the execution environment. This can be retrieved from the `PrepareForInput` method in `dax::cont::ArrayHandle`, as described in Section 3.4.2.

The following is a contrived example of the use of an execution object. Let's say we want to measure the quality of triangles in a mesh. A common method for doing this is using the equation

$$q = \frac{4a\sqrt{3}}{h_1^2 + h_2^2 + h_3^2}$$

where $a$ is the area of the triangle and $h_1$, $h_2$, and $h_3$ are the lengths of the sides. We can easily compute this in a cell map, but what if we want to speed up the computations by reducing precision? After all, we probably only care if the triangle is good, reasonable, or bad. So instead, let's embed a lookup table in an execution object that can quickly retrieve the triangle quality based on its sides.

Example 3.72: Creating and using an executive object that references arrays.

```
#include <dax/exec/ExecutionObjectBase.h>

#include <dax/cont/ArrayHandle.h>
#include <dax/cont/ErrorControlBadValue.h>
```

```
#include <dax/cont/DispatcherMapCell.h>
#include <dax/cont/DispatcherMapField.h>
#include <dax/cont/UniformGrid.h>
#include <dax/cont/UnstructuredGrid.h>

#include <dax/exec/WorkletMapCell.h>
#include <dax/exec/WorkletMapField.h>

#include <dax/math/Compare.h>
#include <dax/math/Exp.h>
#include <dax/math/VectorAnalysis.h>

DAX_EXEC_EXPORT
dax::Vector3 TriangleSideLengths(const dax::Vector3 &point1,
                                 const dax::Vector3 &point2,
                                 const dax::Vector3 &point3)
{
  return dax::make_Vector3(dax::math::Magnitude(point1-point2),
                           dax::math::Magnitude(point2-point3),
                           dax::math::Magnitude(point3-point1));
}

DAX_EXEC_EXPORT
dax::Scalar TriangleQuality(const dax::Vector3 &sideLengths)
{
  // Heron's formula for triangle area.
  dax::Scalar semiparameter = (sideLengths[0]+sideLengths[1]+sideLengths[2])/2;
  dax::Scalar area = dax::math::Sqrt(semiparameter*
                                     (semiparameter - sideLengths[0])*
                                     (semiparameter - sideLengths[0])*
                                     (semiparameter - sideLengths[0]));

  // Formula for triangle quality.
  return 4*area*dax::math::Sqrt(3)/dax::math::MagnitudeSquared(sideLengths);
}

class BuildTriangleQualityArray : public dax::exec::WorkletMapField
{
public:
  typedef void ControlSignature(Field, Field(Out));
  typedef _2 ExecutionSignature(_1);

  DAX_EXEC_EXPORT
  dax::Scalar operator()(const dax::Vector3 &sideLengths)
  {
    return TriangleQuality(sideLengths);
  }
};

template<typename ArrayHandleType>
class TriangleQualityTableExecution : dax::exec::ExecutionObjectBase
{
  typedef typename ArrayHandleType::PortalConstExecution PortalType;

  DAX_CONT_EXPORT
  TriangleQualityTableExecution(ArrayHandleType array,
                                dax::Id arrayDimensions,
                                dax::Scalar spacing)
    : Portal(array.PrepareForInput()), ArrayDimensions(arrayDimensions), Scale(1/spacing)
  {
    if (array.GetNumberOfValues() != arrayDimensions*arrayDimensions*arrayDimensions)
      {
      throw dax::cont::ErrorControlBadValue("Array size was not what was expected.");
      }
  }

  DAX_EXEC_EXPORT
  dax::Scalar operator()(const dax::Vector3 &point1,
```

```
                        const dax::Vector3 &point2,
                        const dax::Vector3 &point3)
  {
    dax::Vector3 lengths = TriangleSideLengths(point1, point2, point3);
    dax::Vector3 indices = lengths*this->Scale;

    dax::Id index = 0;
    for (int i = 2; i >= 0; i--)
      {
      index *= this->ArrayDimensions;
      dax::Id dimIndex = static_cast<dax::Id>(indices[i]);
      index += dax::math::Min(dimIndex, this->ArrayDimensions-1);
      }

    return this->Portal.Get(index);
  }

private:
  PortalType Portal;
  dax::Id ArrayDimensions;
  dax::Scalar Scale;
};

class TriangleQualityTableControl
{
  typedef dax::cont::ArrayHandle<dax::Scalar> ArrayHandleType;
public:
  typedef TriangleQualityTableExecution<ArrayHandleType> ExecutionObjectType;

  DAX_CONT_EXPORT
  TriangleQualityTableControl(dax::Id arrayDimensions, dax::Scalar maxLength)
    : ArrayDimensions(arrayDimensions), Spacing(maxLength/(arrayDimensions-1))
  {
    this->BuildArray();
  }

  DAX_CONT_EXPORT
  ExecutionObjectType GetExecutionObject() const
  {
    return ExecutionObjectType(this->Array, this->ArrayDimensions, this->Spacing);
  }

private:
  ArrayHandleType Array;
  dax::Id ArrayDimensions;
  dax::Scalar Spacing;

  DAX_CONT_EXPORT
  void BuildArray()
  {
    // For convenience, create a uniform grid with point coordinates that match the side
    // lengths we want to compute a table for.
    dax::cont::UniformGrid<> grid;
    grid.SetExtent(dax::make_Id3(0, 0, 0), dax::make_Id3(this->ArrayDimensions-1,
                                                         this->ArrayDimensions-1,
                                                         this->ArrayDimensions-1));
    grid.SetSpacing(dax::make_Vector3(this->Spacing,this->Spacing,this->Spacing));
    grid.SetOrigin(dax::make_Vector3(0,0,0));

    dax::cont::DispatcherMapField<BuildTriangleQualityArray> dispatcher;
    dispatcher.Invoke(grid.GetPointCoordinates(), this->Array);
  }
};

class TriangleQualityWorklet : public dax::exec::WorkletMapCell
{
public:
  typedef void ControlSignature(Topology, Field(Point), ExecObject, Field(Out));
```

```
  typedef _4 ExecutionSignature(_2, _3);

  template<typename TriangleQualityTableType>
  DAX_EXEC_EXPORT
  dax::Scalar operator()(
      dax::exec::CellField<dax::Vector3,dax::CellTagTriangle> &pointCoords,
      const TriangleQualityTableType &triangleQuality)
  {
    return triangleQuality(pointCoords[0], pointCoords[1], pointCoords[2]);
  }
};

DAX_CONT_EXPORT
dax::cont::ArrayHandle<dax::Scalar>
ComputeTriangleQuality(dax::cont::UnstructuredGrid<dax::CellTagTriangle> grid,
                       TriangleQualityTableControl triangleQualityLookup)
{
  dax::cont::ArrayHandle<dax::Scalar> quality;

  dax::cont::DispatcherMapCell<TriangleQualityWorklet> dispatcher;
  dispatcher.Invoke(grid,
                    grid.GetPointCoordinates(),
                    triangleQualityLookup.GetExecutionObject(),
                    quality);

  return quality;
}
```

### 3.5.2 Error Handling

It is sometimes the case during the execution of an algorithm that an error condition can occur that causes the computation to become invalid. At such a time, it is important to raise an error to alert the calling code of the problem. Since Dax uses an exception mechanism to raise errors, we want an error in the execution environment to throw an exception.

However, throwing exceptions in a parallel algorithm is problematic. Some accelerator architectures, like CUDA, do not even support throwing exceptions. Even on architectures that do support exceptions, throwing them in a thread block can cause problems. An exception raised in one thread may or may not be thrown in another, which increases the potential for deadlocks, and it is unclear how uncaught exceptions progress through thread blocks.

The Dax toolkit handles this problem by using a flag and check mechanism. When a worklet encounters an error, it can call its `RaiseError` method to flag the problem and record a message for the error. Once all the threads terminate, the scheduler checks for the error and if one exists throws a `dax::cont::ErrorExecution` exception in the control environment. Thus, calling `RaiseError` looks like an exception was thrown from the perspective of the control environment code that invoked it.

Example 3.73: Raising an error in the execution environment.

```
#include <dax/cont/ArrayHandle.h>
#include <dax/cont/DispatcherMapField.h>
#include <dax/cont/ErrorExecution.h>
```

```
#include <dax/exec/WorkletMapField.h>

#include <dax/math/Exp.h>

class SquareRoot : public dax::exec::WorkletMapField
{
public:
  typedef void ControlSignature(Field, Field(Out));
  typedef _2 ExecutionSignature(_1);

  DAX_EXEC_EXPORT
  dax::Scalar operator()(dax::Scalar x) const
  {
    if (x < 0)
      {
      this->RaiseError("Cannot take the square root of a negative number.");
      }
    return dax::math::Sqrt(x);
  }
};

DAX_CONT_EXPORT
dax::cont::ArrayHandle<dax::Scalar>
InvokeSquareRoot(dax::cont::ArrayHandle<dax::Scalar> input)
{
  dax::cont::ArrayHandle<dax::Scalar> output;

  try
    {
    dax::cont::DispatcherMapField<SquareRoot> dispatcher;
    dispatcher.Invoke(input, output);
    }
  catch (dax::cont::ErrorExecution error)
  {
    std::cout << "An error occurred when taking square root: "
              << error.GetMessage() << std::endl;
  }
}
```

As a convenience, the dax/exec/Assert.h header file contains a macro named DAX_ASSERT_EXEC. It behaves essentially like the POSIX C assert macro except that it takes two arguments, the second being a worklet to call `RaiseError` with. Thus, the conditional in the worklet of Example 3.73 could be replaced with

```
DAX_ASSERT_EXEC(0 <= x, *this);
```

to get relatively the same error checking. However, the error message is going to be less useful to end users and a release build might remove the assert check, so this method should only be used if the errant condition is really unexpected.

Be aware that there are limitations to the execution environment's error handling mechanism because the exception throwing is deferred until after the threads complete. First, it is not possible to catch and handle errors within the worklet, so once an error is raised it is inevitable that the overall worklet operation will fail. Second, calling `RaiseError` does not actually halt any execution. Thus, the error handling will not prevent an invalid block of code from executing. The error flag may or may not terminate execution early, so raising an error should not be counted on to shorten the time of execution.

It is also possible to raise errors within functors launched with the `Schedule` method in the device adapter algorithms (described in Section 3.4.7). The functor for `Schedule` must have a method named `SetErrorMessageBuffer` that accepts an argument of type `dax::exec::internal::ErrorMessageBuffer`. Calling the `RaiseError` on the `ErrorMessageBuffer` will raise an error in the same way as calling `RaiseError` on a worklet.

### 3.5.3 Math

The Dax toolkit comes with several basic math functions. Some of these functions replicate the behavior of the basic POSIX math functions. These functions can vary subtly on different accelerators, and these functions provide cross platform support. Other functions provide convenient implementations of other common math operations that are likely to be helpful in visualization algorithms.

All math functions are located in the `dax::math` package. The functions are most useful in the execution environment, but they can also be used in the control environment when needed.

The math functions are grouped into several different header files based on the type of operation they perform. The following subsections document each of the groups, the header file that defines them, and the contents of each one.

#### Comparisons

The comparison functions are located in dax/math/Compare.h. They help provide ordering of numbers, vectors, and other elements. The following functions are provided.

`dax::math::Max` Takes two arguments and returns the argument that is greater. If called with a vector type, returns a component-wise maximum.

`dax::math::Min` Takes two arguments and returns the argument that is lesser. If called with a vector type, returns a component-wise minimum.

In addition, dax/math/Compare.h provides the pair of templated functors `dax::math::SortLess` and `dax::math::SortGreater` functors. Both functors provide an operation that takes two values of the given type and returns a Boolean. When templated on a scalar type, `SortLess` and `SortGreater` return whether the first value is less than or greater than, respectively, the second value. For types that behave like vectors, these two functors provide a total ordering by comparing the first component, then the second if the first are equal, then the third if the first two are equal, and so on. `SortLess` and `SortGreater` are useful for providing comparison operators to algorithms like `Sort`.

### Exponents

Functions that perform various type of exponential functions are located in dax/math/Exp.h. The following functions are provided.

`dax::math::Cbrt` Takes one argument and returns the cube root of that argument. If called with a vector type, returns a component-wise cube root.

`dax::math::Exp` Computes $e^x$ where $x$ is the argument to the function and $e$ is Euler's number (approximately 2.71828). If called with a vector type, returns a component-wise exponent.

`dax::math::Exp10` Computes $10^x$ where $x$ is the argument. If called with a vector type, returns a component-wise exponent.

`dax::math::Exp2` Computes $2^x$ where $x$ is the argument. If called with a vector type, returns a component-wise exponent.

`dax::math::ExpM1` Computes $e^x - 1$ where $x$ is the argument to the function and $e$ is Euler's number (approximately 2.71828). The accuracy of this function is good even for very small values of $x$. If called with a vector type, returns a component-wise exponent.

`dax::math::Log` Computes the natural logarithm (i.e. logarithm to the base $e$) of the single argument. If called with a vector type, returns a component-wise logarithm.

`dax::math::Log10` Computes the logarithm to the base 10 of the single argument. If called with a vector type, returns a component-wise logarithm.

`dax::math::Log1P` Computes $\ln(1+x)$ where $x$ is the single argument and ln is the natural logarithm (i.e. logarithm to the base $e$). The accuracy of this function is good for very small values. If called with a vector type, returns a component-wise logarithm.

`dax::math::Log2` Computes the logarithm to the base 2 of the single argument. If called with a vector type, returns a component-wise logarithm.

`dax::math::Pow` Takes two arguments and returns the first argument raised to the power of the second argument. This function is only defined for `dax::Scalar`.

`dax::math::RCbrt` Takes one argument and returns the cube root of that argument. The result of this function is equivalent to `1/Cbrt(x)`. However, on some devices it is faster to compute the reciprocal cube root than the regular cube root. Thus, you should use this function whenever dividing by the cube root.

`dax::math::RSqrt` Takes one argument and returns the square root of that argument. The result of this function is equivalent to `1/Sqrt(x)`. However, on some devices it is faster to compute the reciprocal square root than the regular square root. Thus, you should use this function whenever dividing by the square root.

**dax::math::Sqrt** Takes one argument and returns the square root of that argument. If called with a vector type, returns a component-wise square root.

### Matrices

Linear algebra operations on small matrices that are done on a single thread are located in dax/math/Matrix.h.

This header defines the dax::math::Matrix templated class. The template parameters are first the type of component, then the number of rows, then the number of columns. The overloaded parentheses operator can be used to retrieve values based on row and column indices. Likewise, the bracket operators can be used to reference the Matrix as a 2D array (indexed by row first). The following example builds a Matrix that contains the values

$$\begin{vmatrix} 0 & 1 & 2 \\ 10 & 11 & 12 \end{vmatrix}$$

Example 3.74: Creating a Matrix.

```
dax::math::Matrix<dax::Scalar, 2, 3> matrix;
// Using parenthesis notation.
matrix(0, 0) = 0;
matrix(0, 1) = 1;
matrix(0, 2) = 2;
// Using bracket notation.
matrix[1][0] = 10;
matrix[1][1] = 11;
matrix[1][2] = 12;
```

There are also three convenience classes for common matrices. These are dax::math::Matrix2x2, dax::math::Matrix3x3, and dax::math::Matrix4x4. These are all equivalent to a Matrix with a component type of dax::Scalar and of the respective size.

The dax/math/Matrix.h header also defines the following functions that operate on matrices.

**dax::math::MatrixColumn** Given a Matrix and a column index, returns a dax::Tuple of that column. This function might not be as efficient as dax::math::MatrixRow. (It performs a copy of the column).

**dax::math::MatrixDeterminant** Takes a square Matrix as its single argument and returns the determinant of that matrix.

**dax::math::MatrixIdentity** Returns the identity matrix. If given no arguments, it creates an identity matrix and returns it. (In this form, the component type and size must be explicitly set.) If given a single square matrix argument, fills that matrix with the identity.

**dax::math::MatrixInverse** Finds and returns the inverse of a given matrix. The function takes two arguments. The first argument is the matrix to invert. The second argument is a reference to a Boolean that is set to true if the inverse is found or false if the matrix is singular and the returned matrix is incorrect.

**dax::math::MatrixMultiply** Performs a matrix-multiply on its two arguments. Overloaded to work for matrix-matrix, vector-matrix, or matrix-vector multiply.

**dax::math::MatrixRow** Given a Matrix and a row index, returns a dax::Tuple of that row.

**dax::math::MatrixSetColumn** Given a Matrix, a column index, and a dax::Tuple, sets the column of that index to the values of the Tuple.

**dax::math::MatrixSetRow** Given a Matrix, a row index, and a dax::Tuple, sets the row of that index to the values of the Tuple.

**dax::math::MatrixTranspose** Takes a Matrix and returns its transpose.

**dax::math::SolveLinearSystem** Solves the linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ and returns $\boldsymbol{x}$. The function takes three arguments. The first two arguments are the matrix $\boldsymbol{A}$ and the vector $\boldsymbol{b}$, respectively. The third argument is a reference to a Boolean that is set to true if a single solution is found, false otherwise.

### Numerical Methods

Numerical methods are located in dax/math/Numerical.h. They contain small single threaded algorithms of the type you would see in the numerical recipes books by Press et al. [13]. The functions currently available are as follows.

**dax::math::NewtonsMethod** Uses Newton's method (also known as the Newton-Raphson method) to solve a nonlinear system of equations. This function assumes that the number of variables equals the number of equations. Newton's method operates on an iterative evaluate and search. Evaluations are performed using the functors passed into the NewtonsMethod. The function takes the following 6 parameters (three of which are optional).

1. A functor whose operation takes a dax::Tuple and returns the math function's Jacobian vector at that point.
2. A functor whose operation takes a dax::Tuple and returns the evaluation of the math function at that point as another dax::Tuple.
3. The dax::Tuple that represents the desired output of the function.
4. A dax::Tuple to use as the initial guess. If not specified, the origin is used.

5. The convergence distance. If the iterative method changes all values less than this amount, then it considers the solution found. If not specified, set to $10^{-3}$.
6. The maximum amount of iterations to run before giving up and returning the best solution. If not specified, set to 10.

Unlike other methods in `dax::math`, `NewtonsMethod` is declared with `DAX_EXEC_EXPORT`, meaning it only works in the execution environment. This is so that the callbacks passed to `NewtonsMethod` need to work only in the execution environment.

### Precision and Non-Finites

Functions that deal with the precision of numbers and the handling of non-finite numbers (such as not-a-number and infinity) are located in dax/math/Precision.h. The following functions are available.

**`dax::math::Ceil`** Rounds and returns the smallest integer not less than the single argument. If given a vector, performs a component-wise operation.

**`dax::math::Epsilon`** Returns the difference between 1 and the least value greater than 1 that is representable by a `dax::Scalar`. Epsilon is useful for specifying the tolerance one should have when considering numerical error.

**`dax::math::Floor`** Rounds and returns the largest integer not greater than the single argument. If given a vector, performs a component-wise operation.

**`dax::math::FMod`** Computes the remainder on the division of 2 floating point numbers. The return value is $numerator - n \cdot denominator$, where *numerator* is the first argument, *denominator* is the second argument, and $n$ is the quotient of *numerator* divided by *denominator* rounded towards zero to an integer. For example, `FMod(6.5,2.3)` returns 1.9, which is $6.5 - 2 \cdot 4.6$. If given vectors, `FMod` performs a component-wise operation. `FMod` is similar to `Remainder` except that the quotient is rounded toward 0 instead of the nearest integer.

**`dax::math::Infinity`** Returns the `dax::Scalar` representation for infinity. The result is greater than any other number except another infinity or NaN. When comparing two infinities or infinity to NaN, neither is greater than, less than, nor equal to the other.

**`dax::math::IsFinite`** Returns true if the argument is a normal number (neither a NaN nor an infinite).

**`dax::math::IsInf`** Returns true if the argument is either positive infinity or negative infinity.

**`dax::math::IsNan`** Returns true if the argument is not a number (NaN).

`dax::math::ModF` Returns the integral and fractional parts of the first argument. The second argument is a reference in which the integral part is stored. The return value is the fractional part. If given vectors, `ModF` performs a component-wise operation.

`dax::math::Nan` Returns the `dax::Scalar` representation for not-a-number (NaN). A NaN represents an invalid value or the result of an invalid operation such as 0/0. A NaN is neither greater than nor less than nor equal to any other number including other NaNs.

`dax::math::NegativeInfinity` Returns the `dax::Scalar` representation for negative infinity. The result is less than any other number except another negative infinity or NaN. When comparing two negative infinities or negative infinity to NaN, neither is greater than, less than, nor equal to the other.

`dax::math::Remainder` Computes the remainder on the division of 2 floating point numbers. The return value is $numerator - n \cdot denominator$, where *numerator* is the first argument, *denominator* is the second argument, and $n$ is the quotient of *numerator* divided by *denominator* rounded towards the nearest integer. For example, `FMod(6.5,2.3)` returns $-0.4$, which is $6.5 - 3 \cdot 2.3$. If given vectors, `Remainder` performs a component-wise operation. `Remainder` is similar to `FMod` except that the quotient is rounded toward the nearest integer instead of toward 0.

`dax::math::RemainderQuotient` Performs an operation identical to `Reminder`. In addition, this function takes a third argument that is a reference in which the quotient is given.

`dax::math::Round` Rounds and returns the integer nearest the single argument. If given a vector, performs a component-wise operation.

### Positive or Negative Numbers

The dax/math/Sign.h header contains functions that deal with the sign (positive or negative) of numbers. It defines the following functions.

`dax::math::Abs` Returns the absolute value of the single argument. If given a vector, performs a component-wise operation.

`dax::math::CopySign` Copies the sign of the second argument onto the first argument and returns that. If the second argument is positive, returns the absolute value of the first argument. If the second argument is negative, returns the negative absolute value of the first argument.

`dax::math::IsNegative` Returns true if the single argument is less than zero, false otherwise.

`dax::math::SignBit` Returns a nonzero value if the single argument is negative.

### Trigonometry

The dax/math/Trig.h header contains the standard trigonometric functions.

`dax::math::ACos` Returns the arccosine of a ratio in radians. If given a vector, performs a component-wise operation.

`dax::math::ACosH` Returns the hyperbolic arccossine. If given a vector, performs a component-wise operation.

`dax::math::ASin` Returns the arcsine of a ratio in radians. If given a vector, performs a component-wise operation.

`dax::math::ASinH` Returns the hyperbolic arcsine. If given a vector, performs a component-wise operation.

`dax::math::ATan` Returns the arctangent of a ratio in radians. If given a vector, performs a component-wise operation.

`dax::math::ATan2` Computes the arctangent of $y/x$ where $y$ is the first argument and $x$ is the second argument. `ATan2` uses the signs of both arguments to determine the quadrant of the return value. `ATan2` is only defined for `dax::Scalar`.

`dax::math::ATanH` Returns the hyperbolic arctangent. If given a vector, performs a component-wise operation.

`dax::math::Cos` Returns the cosine of an angle given in radians. If given a vector, performs a component-wise operation.

`dax::math::CosH` Returns the hyperbolic cosine. If given a vector, performs a component-wise operation.

`dax::math::Pi` Returns the constant $\pi$ (about 3.14159).

`dax::math::Sin` Returns the sine of an angle given in radians. If given a vector, performs a component-wise operation.

`dax::math::SinH` Returns the hyperbolic sine. If given a vector, performs a component-wise operation.

`dax::math::Tan` Returns the tangent of an angle given in radians. If given a vector, performs a component-wise operation.

`dax::math::TanH` Returns the hyperbolic tangent. If given a vector, performs a component-wise operation.

### Vector Analysis

Visualization and computational geometry algorithms often perform vector analysis operations. The dax/math/VectorAnalysis.h header file provides functions that perform the basic common vector analysis operations.

`dax::math::Cross` Returns the cross product of two `dax::Vector3`.

`dax::math::Magnitude` Returns the magnitude of a vector. This function works on scalars as well as vectors, in which case it just returns the scalar. It is usually much faster to compute `MagnitudeSquared`, so that should be substituted when possible (unless you are just going to take the square root, which would be besides the point). On some hardware it is also faster to find the reciprocal magnitude, so `RMagnitude` should be used if you actually plan to divide by the magnitude.

`dax::math::MagnitudeSquared` Returns the square of the magnitude of a vector. It is usually much faster to compute the square of the magnitude than the length, so you should use this function in place of `Magnitude` or `RMagnitude` when needing the square of the magnitude or any monotonically increasing function of a magnitude or distance. This function works on scalars as well as vectors, in which case it just returns the square of the scalar.

`dax::math::Lerp` Given two values $x$ and $y$ in the first two parameters and a `dax::Scalar` weight $w$ as the third parameter, interpolates between $x$ and $y$. Specifically, the linear interpolation is $(y - x)w + x$ although `Lerp` might compute the interpolation faster than using the independent arithmetic operations. The two values may be scalars or equal sized vectors.

`dax::math::Normal` Returns a normalized version of the given vector. The resulting vector points in the same direction as the argument but has unit length.

`dax::math::Normalize` Takes a reference to a vector and modifies it to be of unit length. `Normalize(v)` is functionally equivalent to `v *= RMagnitude(v)`.

`dax::math::RMagnitude` Returns the reciprocal magnitude of a vector. On some hardware `RMagnitude` is faster than `Magnitude`, but neither is as fast as `MagnitudeSquared`. This function works on scalars as well as vectors, in which case it just returns the reciprocal of the scalar.

`dax::math::TriangleNormal` Given three points in space (contained in `dax::Vector3`s) that compose a triangle return a vector that is perpendicular to the triangle. The magnitude of the result is equal to twice the area of the triangle. The result points away from the "front" of the triangle as defined by the standard counter-clockwise ordering of the points.

### 3.5.4 Cells and Operations

In the control environment, data is defined in grid structures that comprise a set of finite cells. When worklets that operate on cells are scheduled, these grid structures are broken into their independent cells and that data is handed to the worklet.

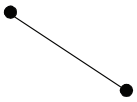
Unlike some other libraries such as VTK, the Dax toolkit does not have a cell class that holds all the information pertaining to a cell of a particular type. Instead, the Dax toolkit provides tags defining the cell type, independent tuples of information containing topological, geometric, and field information, and a collection of functions that perform typical operations. This structure is designed so that a worklet may specify exactly what information it needs, and only that information will be loaded.

#### Tags

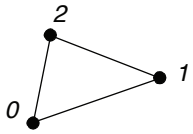
Tags for cells, like all other tags in the Dax toolkit, are empty structures that are used as parameters so that templates may be specialized or functions overloaded based on a cell type. The type of the cell (and by implication the cell tag) is derived from the type of grid structure passed to a `Topology` parameter.

The following table lists all of the cells defined by the Dax toolkit along with the structure and node ordering of each.

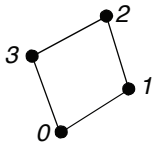
`dax::CellTagVertex`



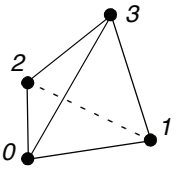
`dax::CellTagLine`



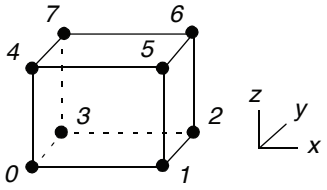

`dax::CellTagTriangle`



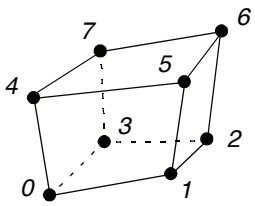

`dax::CellTagQuadrilateral`



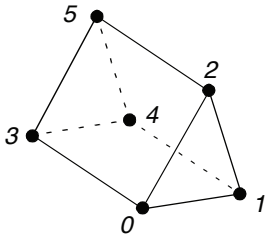

`dax::CellTagTetrahedron`




`dax::CellTagVoxel`




`dax::CellTagHexahedron`




`dax::CellTagWedge`

### Traits

The Dax toolkit has a dax::CellTraits templated class that provides general information about each of the cell types. CellTraits can be used in either the control or execution environment, but are more commonly used in the execution environment.

Like other traits classes, CellTraits contains static information that can be used to specialize templates or overload functions based on general traits. A CellTraits class contains the following items for all known cell tags.

NUM_VERTICES A static constant number set to the number of vertices in the cell.

TOPOLOGICAL_DIMENSIONS A static constant number set to the topological dimensions of the cell type. It is 3 for polyhedra, 2 for polygons, 1 for lines, and 0 for points.

TopologicalDimensionsTag Always typedefed to dax::CellTopologicalDimensionsTag<TOPOLOGICAL_DIMENSIONS>. This tag provides a convenient way to overload a function based on topological dimensions (which is often more efficient than conditionals).

GridTag A typedef to a tag specifying the type of grid that holds this type of cell. Can be set to dax::GridTagUniform or dax::GridTagUnstructured. These correspond to the control environment structures described in Section 3.4.3.

CanonicalCellTag A typedef to a tag for a cell type that can be held in an unstructured grid that has an equivalent topology as this cell. For example, dax::CellTraits<dax::CellTagVoxel>::CanonicalCellTag is typedefed to dax::CellTagHexahedron. This trait is useful for specializing templates and functions that operate on all cell types with equivalent topology. For example, it can be used to create a function that operates on cells from an unstructured grid of hexahedra or a uniform grid.

### Vertex and Field Information

When a worklet scheduled on cells is operating on a point field (generally identified with a Field(Point) tag in the ControlSignature as described in Section 3.5.1), then the Dax dispatcher pulls all the relevant field values and passes them to the worklet in a dax::exec::CellField object. CellField is templated first on the field data type and second on the cell tag. The CellField class has a NUM_VERTICES constant set to the number of field values in the cell. The CellField also has its bracket operator overloaded to access each field value. CellField also has the methods GetAsTuple and SetFromTuple to convert to and from a dax::Tuple structure.

It is also possible to get the point indices for each vertex of the cell. This is done by using the Vertices tag in the ExecutionSignature as described in Section 3.5.1. In this

case, the point indices are placed in a dax::exec::CellVertices object. A CellVertices object behaves the same as a CellField with the field value type set as dax::Id.

The following artificial example uses dax::exec::CellField and dax::exec::CellVertices to make a worklet that finds for each cell the incident point that has the maximum field value.

Example 3.75: Using CellField and CellVertices.

```
#include <dax/exec/CellField.h>
#include <dax/exec/CellVertices.h>
#include <dax/exec/WorkletMapCell.h>

class FindMaxPointIds : public dax::exec::WorkletMapCell
{
public:
  typedef void ControlSignature(Topology, Field(Point), Field(Out));
  typedef _3 ExecutionSignature(Vertices(_1), _2);

  template<typename CellType, typename FieldType>
  DAX_EXEC_EXPORT
  dax::Id operator()(const dax::exec::CellVertices<CellType> &vertices,
                     const dax::exec::CellField<FieldType,CellType> &fieldValues) const
  {
    int maxVertexIndex = 0;
    FieldType maxFieldValue = fieldValues[0];
    for (int vertexIndex = 1; vertexIndex < fieldValues.NUM_VERTICES; vertexIndex++)
      {
      FieldType fieldValue = fieldValues[vertexIndex];
      if (!(fieldValue < maxFieldValue))
        {
        maxVertexIndex = vertexIndex;
        maxFieldValue = fieldValue;
        }
      }
    return vertices[maxVertexIndex];
  }
};
```

### Operations

The Dax execution environment API comes with several functions and classes that perform operations on cells. These facilities are templated on cell tags and operate on CellField containers.

**Parametric Coordinates** Each cell type supports a one-to-one mapping between a set of parametric coordinates in the unit cube (or some subset of it) and the points in 3D space that are the locus contained in the cell. Parametric coordinates are useful because certain features of the cell such as vertex location and center, are at a consistent location in parametric space irrespective of the location and distortion of the cell in world space. Also, many field operations are much easier with parametric coordinates.

The dax/exec/ParametricCoordinates.h header file contains the following functions for working with parametric coordinates.

**dax::cont::ParametricCoordinatesToWorldCoordinates** Given a CellField of point coordinates and a dax::Vector3 containing parametric coordinates, returns the world coordinates.

**dax::cont::WorldCoordinatesToParametricCoordinates** Given a CellField of point coordinates and a dax::Vector3 containing world coordinates, returns the parametric coordinates. This function can be slow for cell types with nonlinear interpolation (which is anything that is not a simplex).

The dax/exec/ParametricCoordinates.h header additionally provides a templated class named dax::exec::ParametricCoordinates. The single template argument is a cell tag. This class holds static methods that return parametric coordinates for special locations. The ParametricCoordinates class contains the following static functions.

**Center** Returns a dax::Vector3 containing the parametric coordinates for the center of a cell.

**Vertex** Returns a dax::exec::CellField of dax::Vector3s containing the parametric coordinates for each vertex of a cell.

**Interpolations** The shape of every cell is defined by the connections of some finite set of vertices. Field values defined on those vertices can be interpolated to any point within the cell to estimate a continuous field.

The dax/exec/Interpolate.h header contains the function dax::exec::CellInterpolate that takes a dax::exec::CellField of field values and a dax::Vector3 containing parametric coordinates. It returns the interpolated value of the field. The dax::worklet::PointDataToCellData worklet provides a simple example of interpolating fields.

Example 3.76: Interpolating a field to the center of a cell.

```
class PointDataToCellData : public dax::exec::WorkletMapCell
{
public:
  typedef void ControlSignature(Topology,Field(Point), Field(Out));
  typedef _3 ExecutionSignature(_2);

  template<class CellTag>
  DAX_EXEC_EXPORT
  dax::Scalar operator()(const dax::exec::CellField<dax::Scalar,CellTag> &pointField) const
  {
    dax::Vector3 center =  dax::exec::ParametricCoordinates<CellTag>::Center();
    return dax::exec::CellInterpolate(pointField,center,CellTag());
  }
};
```

**Derivatives** Since interpolations provide a continuous field function over a cell, it is reasonable to consider the derivative of this function. The dax/exec/Derivative.h/header file provides the overloaded function `dax::exec::CellDerivative` that computes the partial derivative of a field with respect to each axis in 3D space (also known as the *gradient*). `CellDerivative` takes 4 arguments: a `dax::Vector3` containing parametric coordinates of where the gradient should be found, a `dax::exec::CellField` of `dax::Vector3` containing the point coordinates at each vertex, a `dax::exec::CellField` containing the field values at each vertex, and the tag of the cell type.

The `dax::worklet::CellGradient` worklet provides a simple example of finding field derivatives.

Example 3.77: Finding derivatives of a field at the center of a cell.

```
class CellGradient : public dax::exec::WorkletMapCell
{
public:
  typedef void ControlSignature(Topology, Field(Point), Field(Point), Field(Out));
  typedef _4 ExecutionSignature(_2,_3);

  template<class CellTag>
  DAX_EXEC_EXPORT
  dax::Vector3 operator()(const dax::exec::CellField<dax::Vector3,CellTag> &coords,
                          const dax::exec::CellField<dax::Scalar,CellTag> &pointField) const
  {
    dax::Vector3 parametricCellCenter = dax::exec::ParametricCoordinates<CellTag>::Center();
    return dax::exec::CellDerivative(parametricCellCenter, coords, pointField, CellTag());
  }
};
```

## 3.6 OpenGL Interoperability

Although it is designed to run on GPUs, the Dax toolkit is not a rendering library. However, as a toolkit for visualization, it is often desirable to directly render the results from computations using a rendering library like OpenGL. In such a circumstance, it is desirable to transfer data in Dax arrays directly into OpenGL buffers rather than pull back to the CPU and push again to the in a different context.

To facilitate this direct transfer, the Dax toolkit comes with an OpenGL interoperability feature. To transfer an array used in the Dax toolkit to an OpenGL context, use the `dax::opengl::TransferToOpenGL` function. The function takes two arguments. The first argument is a `dax::cont::ArrayHandle` to transfer. The second argument is a reference to a `GLuint`. If this argument contains a valid handle to an OpenGL buffer, then that buffer will point to the array being transferred. Otherwise, a new buffer handle will be generated and returned in this second argument. The function returns a `GLenum` containing the identifier for the OpenGL type in the OpenGL buffer. An overloaded version version of `TransferToOpenGL` takes a third argument that specifies the OpenGL type to use in a `GLenum`.

Example 3.78: Using OpenGL Interoperability

```
DAX_CONT_EXPORT
void BindPointCoordinates(dax::cont::ArrayHandle<dax::Vector3> pointArray)
{
  GLuint oglPointBuffer;
  glGenBuffers(1, &oglPointBuffer);

  dax::opengl::TransferToOpenGL(pointArray, oglPointBuffer);

  glEnableClientState(GL_VERTEX_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER, oglPointBuffer);
  glVertexPointer(3, GL_FLOAT, 0, NULL);
}
```

Although the Dax toolkit supports CUDA computations, it also supports several other types of architectures. Thus, the process for interoperability changes depending on the device adapter being used. The Dax interoperability takes this into account and will provide a custom overload of `TransferToOpenGL` depending on the abilities of the device adapter. Thus, code that uses the Dax toolkit with OpenGL does not need to provide conditionals to manage the different types of devices.

## 3.7 Coding Conventions

Several developers contribute to the Dax toolkit and we welcome others who are interested to also contribute to the project. To ensure readability and consistency in the code, we have adopted the following coding conventions. Many of these conventions are adapted from the coding conventions of the VTK project. This is because many of the developers are familiar with VTK coding and because we expect the Dax toolkit to have continual interaction with VTK.

- All code contributed to Dax must be compatible with Dax's BSD license.
- Copyright notices should appear at the top of all source, configuration, and text files. The statement should have the following form:

```
//============================================================================
//
//  Copyright (c) Kitware, Inc.
//  All rights reserved.
//  See LICENSE.txt for details.
//
//  This software is distributed WITHOUT ANY WARRANTY; without even
//  the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
//  PURPOSE.  See the above copyright notice for more information.
//
//  Copyright 2013 Sandia Corporation.
//  Under the terms of Contract DE-AC04-94AL85000 with Sandia Corporation,
//  the U.S. Government retains certain rights in this software.
```

```
//
//============================================================================
```

The CopyrightStatement checks all files for a similar statement. The test will print out a suggested text that can be copied and pasted to any file that has a missing copyright statement (with appropriate replacement of comment prefix). Exceptions to this copyright statement (for example, third-party files with different but compatible statements) can be added to LICENSE.txt.

- All include files should use include guards. starting right after the copyright statement. The naming convention of the include guard macro is that it should start with two underscores and be followed with the path name, starting from the top-level source code directory, with non alphanumeric characters, such as `/` and `.` replaced with underscores. The `#endif` part of the guard at the bottom of the file should include the guard name in a comment. For example, the dax/cont/ArrayHandle.h header contains the guard

```
#ifndef __dax_cont_ArrayHandle_h
#define __dax_cont_ArrayHandle_h
```

at the top and

```
#endif //__dax_cont_ArrayHandle_h
```

at the bottom.

- The Dax toolkit has several nested namespaces. The declaration of each namespace should be on its own line, and the code inside the namespace bracket should not be indented. The closing brace at the bottom of the namespace should be documented with a comment identifying the namespace. Namespaces can be grouped as desired. The following is a valid use of namespaces.

```
namespace dax {
namespace cont {

namespace detail {

class InternalClass;

} // namespace detail

class ExposedClass;

}
} // namespace dax::cont
```

- Multiple inheritance is not allowed in Dax classes.

- Any functional public class should be in its own header file with the same name as the class. The file should be in a directory that corresponds to the namespace the class is in. There are several exceptions to this rule.

  - Templated classes and template specialization often require the implementation of the class to be broken into pieces. Sometimes a specialization is placed in a header with a different name.
  - Many Dax toolkit features are not encapsulated in classes. Functions may be collected by purpose or co-located with associated class.
  - Although tags are technically classes, they do not behave as an enumeration for the compiler. Multiple tags that make up this enumeration are collected together.
  - Some classes, such as `dax::Tuple` are meant to behave as basic types. These are sometimes collected together as if they were related `typedef`s. The dax/Types.h header is a good example of this.

- The indentation style can be characterized as the "indented brace" (a modified whitesmith) style. Indentations are two spaces, and the curly brace (scope delimiter) is placed on the following line and indented along with the code (i.e. the curly brace lines up with the code).

- Conditional clauses (including loop conditionals such as `for` and `while`) must be in braces below the conditional. That is, instead of

  ```
  if (test) { clause; }
  ```

  use

  ```
  if (test)
    {
    clause;
    }
  ```

  The rational for this requirement is to make it obvious whether the clause is executed when stepping through the code with the debugger. The one exception to this rule is when the clause contains a control-flow statement with obvious side effects such as `return` or `break`. However, even if the clause contains a single statement and is on the same line, the clause should be surrounded by braces.

- Use two space indentation.

- Tabs are not allowed. Only use spaces for indentation. No one can agree on what the size of a tab stop is, so it is better to not use them at all.

- There should be no trailing whitespace in any line.

- Use only alphanumeric characters in names. Use capitalization to demarcate words within a name (camel case). The exception is preprocessor macros and constant numbers that are, by convention, represented in all caps and a single underscore to demarcate words.

- Namespace names are in all lowercase. They should be a single word that designates its meaning.
- All class, method, member variable, and functions should start with a capital letter. Local variables should start in lower case and then use camel case. Exceptions can be made when such naming would conflict with previously established conventions in other library. (For example, `make_Vector2` corresponds to `make_pair` in the standard template library.)
- Always spell out words in names; do not use abbreviations except in cases where the shortened form is widely understood and a name in its own right (e.g. OpenMP).
- Always use descriptive names in all identifiers, including local variable names. Particularly avoid meaningless names of a few characters (e.g. `x`, `foo`, or `tmp`) or numbered names with no meaning to the number or order (e.g. `value1`, `value2`,...). Also avoid the meaningless for loop variable names `i`, `j`, `k`, etc. Instead, use a name that identifies what type of index is being referenced such as `pointIndex`, `vertexIndex`, `componentIndex`, etc.
- Classes are documented with Doxygen-style comments before classes, methods, and functions.
- Exposed classes should not have public instance variables outside of exceptional situations. Access is given by convention through methods with names starting with `Set` and `Get` or through overloaded operators.
- References to classes and functions should be fully qualified with the namespace. This makes it easier to establish classes and functions from different packages and to find source and documentation for the referenced class. As an exception, if one class references an internal or detail class clearly associated with it, the reference can be shortened to `internal::` or `detail::`.
- use `this->` inside of methods when accessing class methods and instance variables to distinguish between local variables and instance variables.
- Include statements should generally be in alphabetical order. They can be grouped by package and type.
- Namespaces should not be brought into global scope or the scope of any Dax package namespace with the "using" keyword. It should also be avoided in class, method, and function scopes (fully qualified namespace references are preferred).
- All code must be valid by the C++03 and C++11 specifications. It must also compile on older compilers that support C++98. Code that uses language features not available in C++98 must have a second implementation that works around the limitations of C++98. The DAX_FORCE_ANSI turns on a compiler check for ANSI compatibility in gcc and clang compilers.
- Limit all lines to 80 characters whenever possible.
- New code must include regression tests that will run on the dashboards. Generally a new class will have an associated "UnitTest" that will test the operation of the test directly. There may be other tests necessary that exercise the operation with different components or on different architectures.

- All code must compile and run without error or warning messages on the nightly dashboards, which should include Windows, Mac, and Linux.
- Use `dax::Scalar` in lieu of `float` or `double` to represent data for computation and use `dax::Id` in lieu of `int` or `long` for data structure indices. This future-proofs code against changes in precision of the architecture. The indices of `dax::Tuple` are an exception. Using `int` to reference `dax::Tuple` (and other related classes like `dax::exec::CellField` and `dax::math::Matrix`) indices are acceptable as it is unreasonable to make these vectors longer than the precision of `int`.
- All functions and methods defined within the Dax toolkit should be declared with `DAX_CONT_EXPORT`, `DAX_EXEC_EXPORT`, or `DAX_EXEC_CONT_EXPORT`.

We should note that although these conventions impose a strict statute on Dax coding, these rules (other than those involving licensing and copyright) are not meant to be dogmatic. Examples can be found in the existing code that break these conventions, particularly when the conventions stand in the way of readability (which is the point in having them in the first place). For example, it is often the case that it is more readable for a complicated `typedef` to stretch a few characters past 80 even if it pushes past the end of a display.

# Chapter 4

# Progress Report

In this chapter we capture the information acquired during the project that is not directly related to the implementation of the Dax toolkit as recorded in Chapters 2 and 3.

## 4.1 Lessons Learned

As with any research project, over the course of the work we made discovers that taught us what does and does not work well. The publications listed in the executive summary of this document report the major findings, particularly the larger ones. Here we capture some smaller discoveries that, although themselves not warranting their own publication, have contributed to our understanding of developing a general purpose toolkit for massively threaded visualization algorithms.

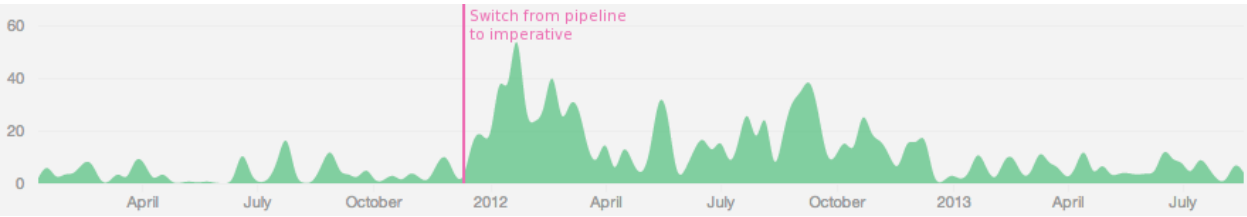
### 4.1.1 Abandonment of Kernel Fusion

In the original proposal for this project, we described a system that builds a pipeline of operations, much like the VTK pipeline but with much lighter-weight operations. The system would find chains of operations and then group their execution together so that all operations would happen on a single data element together. The main intention of this specialized form of kernel fusion is to maximize the amount of execution happening per load of the data.

As we attempted to implement such a system, we found the implementation was more technically difficult than expected. Worse, the API for managing these connections was extremely awkward in the sense that the it was difficult to understand and difficult to use even if it was understood. Both the underlying system and the API underwent several unsatisfactory redesigns.

Eventually, we decided that this extra complication in both interface and implementation was not providing much. The Dax toolkit is already structured in such a way as to encourage adding many instructions to a minimal amount of data by coding through the worklet paradigm. Ultimately, the optimization was doing nothing that a Dax user was not likely to be doing herself.

This distraction stalled the toolkit development for several months. Once we changed to a more imperative interface, development took off as is evident with the plot of repository commits over time.




### 4.1.2 Explicit Memory Hierarchy

The NVIDIA CUDA architecture has a very unique memory hierarchy. Cores are grouped into streaming multiprocessor units that together access a local "shared memory" that is much faster than the global shared memory. Our initial expectation was that we would gain significant performance enhancement by explicitly managing these memory hierarchies.

However, early in the projects we made comparisons of the effect of memory access when either explicitly loading data into these shared memory banks or by simply allowing the GPU's caching mechanism to load the shared memory. What we found was that the automatic caching mechanism generally worked as well as our hand-tuned memory loading using knowledge about the structure of the data and our access to it.

So, our conclusion is that allowing the caching mechanism is a good way to populate the memory hierarchy while also realizing code that is simpler and easier to maintain.

### 4.1.3 Alternate Topology Data Structures

Before designing algorithms within the Dax toolkit, we had to agree on a data representation for our grid structures. Meshes like a uniform grid are straightforward because their topology is implicit and there is nothing much to capture. However, for more general unstructured combinations there must be an explicit way to establish the shape of each cell and the connections between them.

The data model used by VTK and many other software packages and applications uses a vertex-cell model where an index array captures for each cell the points that comprise the cell's vertices. The vertices for a particular cell type follow a conventional order such as the CGNS convention [15].

We also considered several alternate methods for representing meshes of polyhedra. These included half-edge and winged-edge structures [7], cellular data structures [1], and circular incident edge lists [8]. All of these representations use a linked list of indices that trace around the constituent elements of a cell.

The advantage of these structures is that the can capture some of the incidence relationships that the vertex-cell model does not directly capture. However, the disadvantage is that to build these structures, it means finding these incidence relationships whether they are directly used or not. Furthermore, when one mesh is derived from another mesh, these relationships again have to be derived one from another. Another problem with these mesh types is that they often require tracing links to find information about the data structure, and following these linked lists is often an inefficient operation on massively threaded processors.

Consequently, the Dax toolkit uses the traditional vertex-cell model to represent its unstructured grids, as described in Section 3.4.3. We have discovered a variety of sorting techniques to find incidence relationships that are not explicitly captured with the cell links.

### 4.1.4 Data Transfer Time

It is common knowledge that when using an accelerator like a GPU that you should avoid transferring data to and from the accelerator like the plague. This is because the speed of the bus between the host computer and the accelerator device is orders of magnitude slower than accessing the memory locally.

This advice is certainly true for something like rendering where a repetitive operation must cycle at 20Hz or faster. It is also true for small operations. For example, you cannot transfer data back and forth every time a simple vector operation is performed. However, this advice can lead you astray if taken too far.

We performed an informal investigation on the transfer time with respect to the operating time of an algorithm within the Dax framework. What we find is that for an operation that performs more than a trivial amount of computation, the transfer time is insignificant. For example, the Marching Cubes algorithm, which itself is not a huge amount of computation, still spends two orders of magnitude more time in execution than in the total time of transferring memory to and from the GPU as demonstrated in Figure 4.1.

The importance of this finding is the simplification of integrating algorithms in the Dax toolkit with other visualization systems like VTK and ParaView. These are based on the visualization pipeline [12], which splits algorithms into individual components. Although it would be possible to leave data on the GPU when passing from one component to the next, it creates great difficulty in managing the limited amount of memory available because there is no good way to know what, if anything, will be required from one component to the next.

A more tractable approach is to pull all data back to the host memory before passing from one algorithm to the next. Compared to an algorithm like Marching Cubes, the overhead is negligible. Even for very minor operations like finding normals, the overhead does not get above a few hundredths of a second. Since visualization pipelines typically do not get very deep and are executed infrequently compared to other operations like rendering, adding such an overhead is irrelevant in practice.

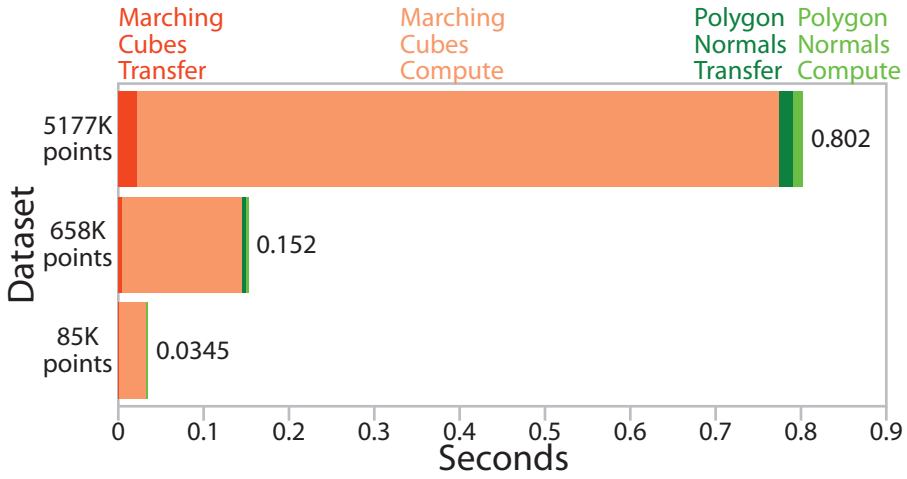



Figure 4.1: Comparison of execution time vs. transfer time.

## 4.2 Results

The intention of the Dax project is to provide the generic infrastructure to build visualization algorithms on. To demonstrate the infrastructure currently implemented, we present to exemplary algorithms: threshold and Marching Cubes.

### 4.2.1 Threshold

The threshold algorithm can be summarized as follows: For each cell in the data set, find the points that form the cell and the corresponding scalar field values for each of those points. If the scalar field values for all the points are within the threshold range, then pass the cell and the corresponding points to the output data set. Since points are often shared between cells, we also avoid passing duplicate points in the output. This ensures both that the representation of the output does not require more space than needed and that the resultant data set is suitable for further analysis if needed.

To implement the algorithm within the Dax framework, we need to map the algorithm to multiple worklets. Based on the implementation by Lo et al. [9], the thresholding operation can be characterized as the following steps:

1. For every cell in the input data set, we need to first determine if it passes the threshold criteria. This can be implemented with a cell map worklet whose output is the number of cells it will create. In the case of threshold, this will be either 0 or 1 cell.

2. Once we have generated the count array, we need to determine how much space to allocate in the output and build indices from either input to output or output to input. This is a fairly common task in parallel programming, known as stream compaction. The Dax dispatcher performs this all internally before performing the following step.

3. For every cell that passes the threshold criteria, we need to generate a duplicate cell for the output. This operation is implemented inside of a generate topology worklet.

4. It is possible (and for threshold likely) that not all points in the output are referenced by a cell. The Dax dispatcher can remove unused points and compact the array; however, this step is optional.

We demonstrate the threshold algorithm by running it on example supernova simulation results on a $432^3$ uniform grid. In our first set of experiments, we run a version of the algorithm that is isomorphic to what VTK produces. The results of these experiments are summarized in Figure 4.2.

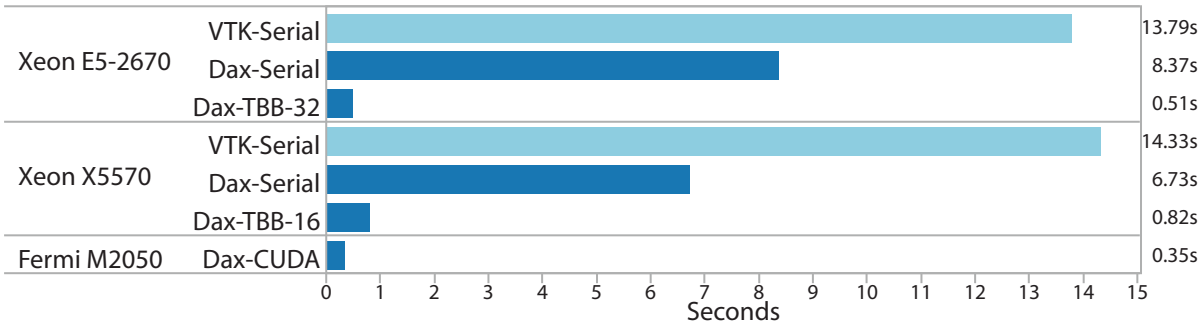



Figure 4.2: Timing of threshold with output point masking.

We see that even when running Dax in serial, its threshold algorithm is roughly twice as fast as the equivalent VTK algorithm. We attribute this mostly to more efficient data manipulations in Dax. Furthermore, the Dax parallel algorithm makes good use of multiple cores.

As previously stated, the final step of the generate topology method, where unused points are removed, is optional. In our second set of experiments, we run the algorithm on the same data set as before, but skip the point-merging step. We compare the algorithm in Dax with an equivalent algorithm from the PISTON project. Note, however, that we modified the algorithm in PISTON to output cells equivalent to what Dax produces. (Specifically, the original PISTON algorithm produces quadrilaterals of the passed faces, and that was changed to produce the hexahedra themselves. Our modified algorithm runs faster than the original because it produces smaller arrays.) The results of these experiments are summarized in Figure 4.3.

Even though Dax adds abstractions that are not used in the PISTON algorithm, the C++ templates remove most of the overhead making the Dax algorithm very efficient. In some cases we are even faster than the PISTON algorithm. This is attributed to a 3D block scheduler that is not available in the Thrust library used by PISTON.

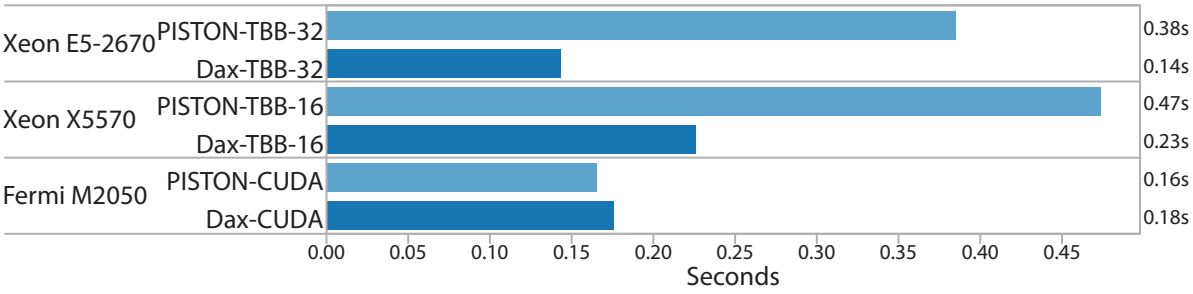



Figure 4.3: Timing of threshold without output point masking.

### 4.2.2 Marching Cubes

The Marching Cubes algorithm [10] is a classic scientific visualization algorithm used to extract the contour surface from a volume where a field is of a particular specified value. In this algorithm each cell of the volume is analyzed, and using a table of cases and interpolations a set of polygons representing the contour in that cell are produced. Polygons produced in adjacent cells will have coincident points, and managing these connections is parallel processing is challenging. The operation of Marching Cubes is similar to that of the threshold algorithm.

- For every cell in the input data set, we need to first determine how many polygons and points will be produced. In the case of Marching Cubes, we look in our case table and determine the size of the output.
- Once we have generated the count array, we need to determine how much space to allocate in the output and build indices from either input to output or output to input. This is a fairly common task in parallel programming, known as stream compaction. The Dax dispatcher performs this all internally before performing the following step.
- Armed with a mapping between input and output, a second parallel operation generates the points and cell connections that make up the contour. This operation is implemented inside of an interpolated cell worklet.
- We know that polygons generated by independent threads will have coincident points. The Dax dispatcher can find these coincident points by comparing a topological key (currently the edge it is created on and the interpolated distance across it). However, this step is optional.

We demonstrate the Marching Cubes algorithm by running it on the example supernova simulation results on a $432^3$ uniform grid. In our first set of experiments, we run a version of the algorithm that is isomorphic to what VTK produces. The results of these experiments are summarized in Figure 4.4.

We note that the serial version of the Dax algorithm is slower than that in VTK, but this is because the VTK contour algorithm is technically not Marching Cubes. It uses a

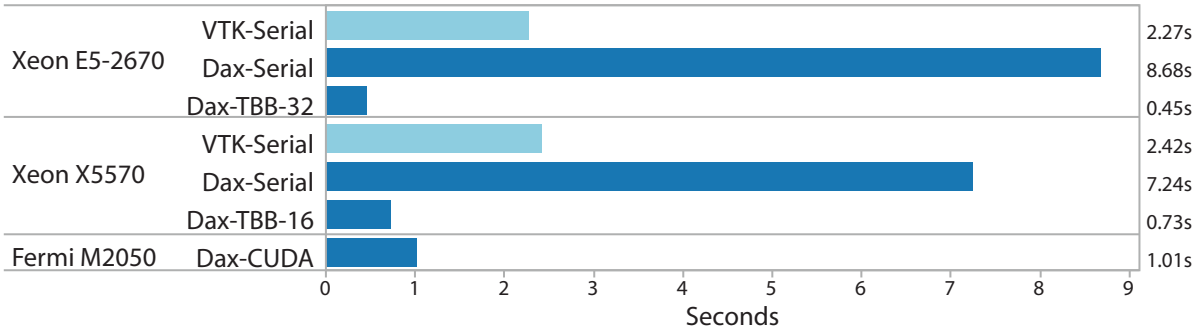



Figure 4.4: Timing of Marching Cubes when outputting a manifold surface.

different algorithm called Synchronized Templates, which uses the nature of the uniform grid structure to remove redundant computation and share coincident vertices. However, Synchronized Templates only works on this type of uniform data, and there is no known way to parallelize the algorithm. Thus, when Dax is run in parallel it can outperform the VTK version.

As previously stated, the final step of the interpolated cell method, where coincident points are merged, is optional. In our second set of experiments, we run the algorithm on the same data set as before, but skip the coincident-point-merging step to produce a triangle soup instead of a manifold surface. We compare the algorithm in Dax with an equivalent algorithm from the PISTON project. The results of these experiments are summarized in Figure 4.5.

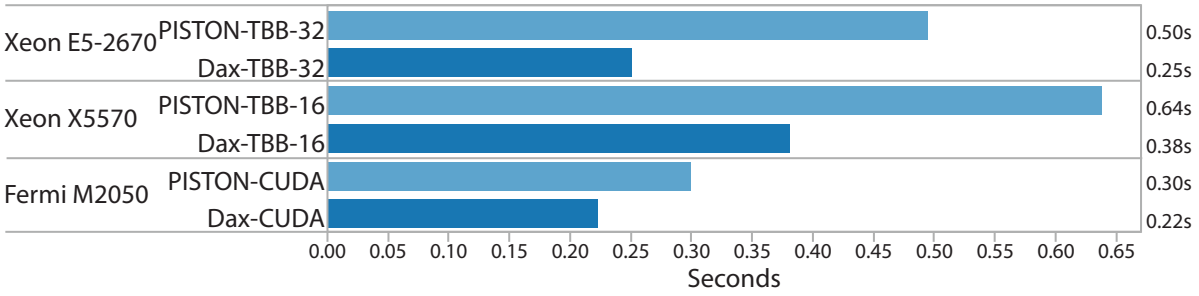



Figure 4.5: Timing of Marching Cubes when outputting a triangle soup.

Even though Dax adds abstractions that are not used in the PISTON algorithm, the C++ templates remove most of the overhead making the Dax algorithm very efficient.

## 4.3 Future Plans

We hope to soon start new projects that continue to develop the Dax framework. There are many avenues of future development we wish to pursue.

- Integrate more tightly into existing and emerging visualization libraries and applications such as VTK, ParaView, VisIt, PISTON, and EAVL.
- Expand the framework to support additional data structures and cell types.
- With the goal to provide a rich collection of commonly used analysis algorithms within the toolkit, we will continue work on developing worklets (and corresponding worktypes) for further algorithms.
- Begin to use the Dax toolkit on real scientific application problems within DOE Office of Science and elsewhere.
- Facilitate the integration of analysis with simulation by using Dax for in situ analysis.
- Provide a more formal categorization of visualization algorithms and behavior in massive parallelism.

# References


[1] Tyler J Alumbaugh and Xiangmin Jiao. Compact array-based mesh data structures. In *Proceedings, 14th International Meshing Roundtable*, pages 485–504, September 2005.

[2] Christopher G. Baker, Michael A. Heroux, H. Carter Edwards, and Alan B. Williams. A light-weight API for portable multicore programming. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 601–606, February 2010. DOI 10.1109/PDP.2010.49.

[3] Nathan Bell and Jared Hoberock. *GPU Computing Gems, Jade Edition*, chapter Thrust: A Productivity-Oriented Library for CUDA, pages 359–371. Morgan Kaufmann, October 2011.

[4] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990. ISBN 0-262-02313-X.

[5] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP*. MIT Press, 2007. ISBN 978-0-262-53302-7.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[7] Lutz Kettner. Designing a data structure for polyhedral surfaces. In *Proceedings of the Fourteenth ACM Symposium on Computational Geometry*, pages 146–154, 1998. DOI 10.1145/276884.276901.

[8] Bruno Lévy, Guillaume Caumon, Stéphane Conreaux, and Xavier Cavin. Circular incident edge lists: a data structure for rendering complex unstructured grids. In *Proceedings of IEEE Visualization*, pages 191–198, October 2001.

[9] Li-Ta Lo, Chris Sewell, and James Ahrens. PISTON: A portable cross-platform framework for data-parallel visualization operators. Technical Report LA-UR-12-10227, Los Alamos National Laboratory, 2012.

[10] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics (Proceedings of SIGGRAPH 87)*, 21(4):163–169, July 1987.

[11] Scott Mayers. *Effective C++*. Addison Wesley, third edition, August 2009. ISBN 0-321-33487-6.

[12] Kenneth Moreland. A survey of visualization pipelines. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):367–378, March 2013. DOI 10.1109/TVCG.2012.133.

[13] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, 2002. ISBN 0-521-75033-4.

[14] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, July 2007. ISBN 978-0-596-51480-8.

[15] Christopher L. Rumsey, Diane M. A. Poirier, Robert H. Bush, and Charles E. Towne. A user's guide to cgns. Technical Report TM-2001-211236, NASA, October 2001.

[16] Jason Sanders and Edward Kandrot. *CUDA by Example*. Addison Wesley, 2011. ISBN 978-0-13-138768-3.

# Index

# DISTRIBUTION:

1 Lucy Nowell
U.S. Department of Energy
SC-21
19901 Germantown Road
Germantown, MD 20874-1290

1 Teresa Beachley
U.S. Department of Energy
SC-21
19901 Germantown Road
Germantown, MD 20874-1290

1 Berk Geveci
Kitware, Inc.
28 Corporate Drive
Clifton Park, NY 12065

1 Robert Maynard
Kitware, Inc.
28 Corporate Drive
Clifton Park, NY 12065

1 Kwan-Liu Ma
Department of Computer Science
2063 Kemper Hall
University of California-Davis
One Shields Avenue
Davis, CA 95616-8562

12 MS 1326 Kenneth Moreland, 1461

1 MS 1327 Ron Oldfield, 01461

1 MS 0899 Technical Library, 9536 (electronic copy)

Sandia National Laboratories