SAND2017-10565R

Milestone Completion Report

WBS 1.3.5.05 ECP/VTK-m
FY17Q4 [MS-17/03-06] Key Reduce / Spatial Division / Basic Advect / Normals, STDA05-4

Kenneth Moreland
Sandia National Laboratories

September 30, 2017

EXECUTIVE SUMMARY

The FY17Q4 milestone of the ECP/VTK-m project includes the completion of a key-reduce scheduling mechanism, a spatial division algorithm, an algorithm for basic particle advection, and the computation of smoothed surface normals. With the completion of this milestone, we are able to, respectively, more easily group like elements (a common visualization algorithm operation), provide the fundamentals for geometric search structures, provide the fundamentals for many flow visualization algorithms, and provide more realistic rendering of surfaces approximated with facets.

1. INTRODUCTION

This report documents the completion of the milestone "FY17Q4 [MS-17/03-06] Key Reduce / Spatial Division / Basic Advect / Normals" (listed as epic STDA05-4 in JIRA) for the ECP/VTK-m project. The overarching goal of the ECP/VTK-m project is to enable scientific visualization on the emerging processors required for the latest generation of petascale computers and the extreme scale computers of the future.

One of the biggest recent changes in high-performance computing is the increasing use of accelerators. Accelerators contain processing cores that independently are inferior to a core in a typical CPU, but these cores are replicated and grouped such that their aggregate execution provides a very high computation rate at a much lower power. Current and future CPU processors also require much more explicit parallelism. Each successive version of the hardware packs more cores into each processor, and technologies like hyperthreading and vector operations require even more parallel processing to leverage each core's full potential.

VTK-m is a toolkit of scientific visualization algorithms for emerging processor architectures. VTK-m supports the fine-grained concurrency for data analysis and visualization algorithms required to drive extreme scale computing by providing abstract models for data and execution that can be applied to a variety of algorithms across many different processor architectures.

Although there will be some time spent in the VTK-m project building up the infrastructure, the majority of the work is in redeveloping, implementing, and supporting necessary visualization algorithms in the new system. We plan to leverage a significant amount of visualization software for the exascale, but there is still a large base of complex, computationally intensive algorithms built over the last two decades that need to be redesigned for advanced architectures. Although VTK-m simplifies the design, development, and implementation of such algorithms, updating the many critical scientific visualization algorithms in use today requires significant investment. And, of course, all this new software needs to be hardened for production, which adds a significant overhead to development.

Our proposed effort will in turn impact key scientific visualization tools. Up until now, these tools — ParaView, VisIt, and their in situ forms — have been underpinned by the Visualization ToolKit (VTK) library. VTK-m builds on the VTK effort, with the "-m" referring to many-core capability. The VTK-m name was selected to evoke what VTK has delivered: a high-quality library with rich functionality and production software engineering practices, enabling impact for many diverse user communities. Further, VTK-m is being developed by some of the same people who built VTK, including Kitware, Inc., which is the home to VTK (and other product lines). Developers of ParaView and VisIt are in the process of integrating VTK-m, using funding coming from ASCR and ASC. However, while VTK-m has made great strides in recent years, it is missing myriad algorithms needed to be successful within the ECP. Developing those algorithms is the focus of this ECP proposal.

2. MILESTONE OVERVIEW

The "FY17Q4 [MS-17/03-06] Key Reduce / Spatial Division / Basic Advect / Normals" comprises the completion of 4 VTK-m activities, captured as stories in JIRA. These are "[MS-17/03] Key Reduce

Worklet” (STDA05-7), “[MS-17/04] Spatial Division” (STDA05-8), “[MS-17/05] Advect Steady State” (STDA05-9), and “[MS-17/06] Smooth Surface Normals” (STDA05-10).

[MS-17/03] KEY REDUCE WORKLET

Milestone Description: A reduce-by-key pattern is a helpful tool for resolving dependencies of data computed in parallel. The idea is that when a thread needs to produce results that might be affected by other threads it instead writes out an intermediate result along with a uniquely identifying key. Then a reduce-by-key operation (provided by this milestone) collects all identical keys and runs a function with all the data associated with the values associated with that key. Typically, there are many unique keys and all reductions can be run in parallel.

Milestone Execution Responsibility: Kenneth Moreland

Milestone Execution Plan: Implementing a reduce-by-key operation is well understood. The basic idea is to sort the keys so that they are adjacent in an array, which makes them easy to identify in parallel. This work codifies the implementation in a worklet. The execution goes as follows.

- Implement the classes required to manage keys and provide variable length groups of keys to worklets.
- Add the necessary transport and fetch operations for worklet execution.
- Implement worklet and dispatcher classes.
- Updated and simplify existing worklets to the new feature where relevant.

Milestone Linkage: No prior milestone completion required. Several milestones can benefit from this building block including [MS-18/05], [MS-19/03], [MS-19/05], [MS-20/03], [MS-20/06], and likely many others.

Milestone Completion Criteria:

- Implementation of the new worklet, dispatcher, and related components are merged to the master branch of the central VTK-m repository.
- Documentation of the key-reduce worklet is added to the VTK-m User’s Guide working document (including examples).

[MS-17/04] SPATIAL DIVISION

Milestone Description: Spatial division algorithms create a partitioning in space and identify the location of points in that partitioning. Examples include k-d trees and OBB (oriented bounding box) trees. Typically, the spatial partitioning is designed to evenly divide the points at each level of a recursive partitioning. Many search algorithms use spatial division to rapidly locate or intersect elements.

Milestone Execution Responsibility: David Rogers

Milestone Execution Plan:

- Investigate the effectiveness of a select group of spatial partitions in organizing and finding geometric data. Candidates include k-d trees, uniform binning, and 2-level grids.

- Implement the structures necessary to represent the best search structures and reference the data they come from.
- Implement the ability to build the best search structures. These build algorithms should use the parallel algorithm features of VTK-m for fast parallel execution.

Milestone Linkage: No prior milestone completion required. Needed for [MS-18/06], locate point, and [MS-18/07], locate cell.

Milestone Completion Criteria:
- Implementation of the algorithms to build spatial division structures are merged to the master branch of the central VTK-m repository.
- Documentation of the spatial division structures, how to build them, and their use are added to the VTK-m User's Guide working document. (Note that the implementation of this milestone is mostly in support of implementing milestones [MS-18/06] and [MS-18/07], so the interface and documentation at this point might be sparse with complete documentation for these later milestones.)

[MS-17/05] ADVECT STEADY STATE

Milestone Description: This includes the ability to advect many particles in 2D and 3D flows where the flow direction is not changing over time. We will support multiple integration methods and multiple mesh structures.

The initial implementation of steady state advection will work only on uniform grids where cells are easy to locate. Follow on milestones will generalize the ability to other grid types as cell location gets implemented.

Milestone Execution Responsibility: David Pugmire, Hank Childs

Milestone Execution Plan:
- Design a simple framework for particle advection that allows you to specify the integration method and the output method for each step.
- Implement the framework with some basic integration methods (such as Runge-Kutta) and output methods (such as trace of particles).
- Wrap the functionality in one or more filter classes.

Milestone Linkage: No prior milestone completion required. Needed for [MS-19/07], advect time varying fields. [MS-18/07] will add the ability to advect through different cell topologies (by providing the necessary search structures). [MS-19/04] might improve functionality for multi-block data sets.

Milestone Completion Criteria:
- Implementation is merged to the master branch of the central VTK-m repository.
- Documentation is added to the VTK-m User's Guide working document.

[MS-17/06] SMOOTH SURFACE NORMALS

Milestone Description: This milestone improves on the faceted surface normals by averaging the normals to each point in the mesh. These normals on points can then be interpolated by the rendering system to estimate the lighting on a smooth surface.

Milestone Execution Responsibility: Berk Geveci

Milestone Execution Plan:
- Implement a cell to point worklet that takes normals defined for each polygon and average them out. This implementation could either compute the normals on the fly (saving on memory) or use an array of normals for each cell (saving on computation). Only one such implementation needs to be created.
- Update the associated filter.

Milestone Linkage: Requires [MS-17/02], faceted surface normals. Needed for [MS-20/03], feature-sensitive surface normals.

Milestone Completion Criteria:
- Implementation is merged to the master branch of the central VTK-m repository.
- Documentation is added to the VTK-m User's Guide working document.

3. TECHNICAL WORK SCOPE, APPROACH, RESULTS

[MS-17/03] KEY REDUCE WORKLET

This activity has the following completion criteria:
- Implementation of the new worklet, dispatcher, and related components are merged to the master branch of the central VTK-m repository.
- Documentation of the key-reduce worklet is added to the VTK-m User's Guide working document (including examples).

Merge into VTK-m repository

The VTK-m project uses "merge requests" as its mechanism for adding code to the master VTK-m repository. The gitlab repository manages the review process and saves the information for it. The following completed merge requests were used to add the code for this activity.

- MR !645, Reduce by Key Worklet Type, https://gitlab.kitware.com/vtk/vtk-m/merge_requests/645
- MR !662, Make external faces more generic, https://gitlab.kitware.com/vtk/vtk-m/merge_requests/662
- MR !903, Expand usage of AverageByKey, https://gitlab.kitware.com/vtk/vtk-m/merge_requests/903

Documentation

The following excerpt from the user's guide document this new functionality.

14.5.3 Reduce by Key

A worklet deriving vtkm::worklet::WorkletReduceByKey operates on an array of keys and one or more associated arrays of values. When a reduce by key worklet is invoked, all identical keys are collected and the worklet is called once for each unique key. Each worklet invocation is given a Vec-like containing all values associated with the unique key. Reduce by key worklets are very useful for combining like items such as shared topology elements or coincident points.

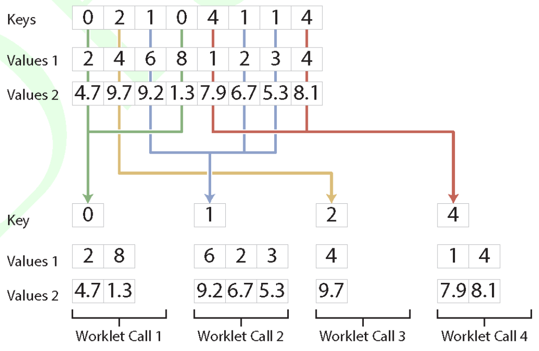


Figure 14.2: The collection of values for a reduce by key worklet.

Figure 14.2 show a pictorial representation of how VTK-m collects data for a reduce by key worklet. All calls to a reduce by key worklet has exactly one array of keys. The key array in this example has 4 unique keys: 0, 1, 2, 4. These 4 unique keys will result in 4 calls to the worklet function. This example also has 2 arrays of values associated with the keys. (A reduce by keys worklet can have any number of values arrays.)

Within the dispatch of the worklet, all these common keys will be collected with their associated values. The parenthesis operator of the worklet will be called once per each unique key. The worklet call will be given a `Vec`-like containing all values that have the key.

A `WorkletReduceByKey` subclass is invoked with a `vtkm::worklet::DispatcherReduceByKey`. 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 reduce by key worklet supports the following tags in the parameters of its `ControlSignature`.

`KeysIn` This tag represents the input keys. A `KeysIn` argument expects a `vtkm::worklet::Keys` object in the associated parameter of the dispatcher's `Invoke`. The `Keys` object, which wraps around an `ArrayHandle` containing the keys and manages the auxiliary structures for collecting like keys, is described later in this section.

  Each invocation of the worklet gets a single unique key.

  A `WorkletReduceByKey` object must have exactly one `KeysIn` parameter in its `ControlSignature`, and the `InputDomain` must point to the `KeysIn` parameter.

`ValuesIn` This tag represents a set of input values that are associated with the keys. A `ValuesIn` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The number of values in this array must be equal to the size of the array used with the `KeysIn` argument. Each invocation of the worklet gets a `Vec`-like object containing all the values associated with the unique key.

  `ValuesIn` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 143.

`ValuesInOut` This tag behaves the same as `ValuesIn` except that the worklet may write values back into the `Vec`-like object, and these values will be placed back in their original locations in the array. Use of `ValuesInOut` is rare.

`ValuesOut` This tag behaves the same as `ValuesInOut` except that the array is resized appropriately and no input values are passed to the worklet. As with `ValuesInOut`, values the worklet writes to its `Vec`-like object get placed in the location of the original arrays. Use of `ValuesOut` is rare.

`ReducedValuesOut` This tag represents the resulting reduced values. A `ReducedValuesOut` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

  `ReducedValuesOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 143.

`ReducedValuesIn` This tag represents input values that come from (typically) from a previous invocation of a reduce by key. A `ReducedValuesOut` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The number of values in the array must equal the number of *unique* keys.

  `ReducedValuesIn` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 143.

  A `ReducedValuesIn` argument is usually used to pass reduced values from one invocation of a reduce by key worklet to another invocation of a reduced by key worklet such as in an algorithm that requires iterative steps.

A reduce by key worklet supports the following tags in the parameters of its ExecutionSignature.

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

ValueCount This tag produces a vtkm::IdComponent that is equal to the number of times the key associated with this call to the worklet occurs in the input. This is the same size as the Vec-like objects provided by ValuesIn arguments.

WorkIndex This tag produces a vtkm::Id that uniquely identifies the invocation of the worklet.

VisitIndex This tag produces a vtkm::IdComponent that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 14.10).

InputIndex This tag produces a vtkm::Id that identifies the index of the input element, which can differ from the WorkIndex in a worklet with a scatter (as described in Section 14.10).

OutputIndex This tag produces a vtkm::Id that identifies the index of the output element. (This is generally the same as WorkIndex.)

ThreadIndices This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects are described in Section 24.2, but most users can get the information they need through other signature tags.

As stated earlier, the reduce by key worklet is useful for collected like values. To demonstrate the reduce by key worklet, we will create a simple mechanism to generate a histogram in parallel. (VTK-m comes with its own histogram implementation, but we create our own version here for a simple example.) The way we can use the reduce by key worklet to compute a histogram is to first identify which bin of the histogram each value is in, and then use the bin identifiers as the keys to collect the information. To help with this example, we will first create a helper class named BinScalars that helps us manage the bins.

Example 14.10: A helper class to manage histogram bins.

```
class BinScalars
{
public:
  VTKM_EXEC_CONT
  BinScalars(const vtkm::Range& range, vtkm::Id numBins)
    : Range(range), NumBins(numBins)
  {  }

  VTKM_EXEC_CONT
  BinScalars(const vtkm::Range& range, vtkm::Float64 tolerance)
    : Range(range)
  {
    this->NumBins = vtkm::Id(this->Range.Length()/tolerance) + 1;
  }

  VTKM_EXEC_CONT
  vtkm::Id GetBin(vtkm::Float64 value) const
  {
    vtkm::Float64 ratio = (value - this->Range.Min)/this->Range.Length();
    vtkm::Id bin = vtkm::Id(ratio * this->NumBins);
    bin = vtkm::Max(bin, vtkm::Id(0));
    bin = vtkm::Min(bin, this->NumBins-1);
    return bin;
  }

private:
  vtkm::Range Range;
  vtkm::Id NumBins;
};
```

Using this helper class, we can easily create a simple map worklet that takes values, identifies a bin, and writes that result out to an array that can be used as keys.

Example 14.11: A simple map worklet to identify histogram bins, which will be used as keys.

```
struct IdentifyBins : vtkm::worklet::WorkletMapField
{
  typedef void ControlSignature(FieldIn<Scalar> data,
                                FieldOut<IdType> bins);
  typedef _2 ExecutionSignature(_1);
  using InputDomain = _1;

  BinScalars Bins;

  VTKM_CONT
  IdentifyBins(const BinScalars& bins)
    : Bins(bins)
  { }

  VTKM_EXEC
  vtkm::Id operator()(vtkm::Float64 value) const
  {
    return Bins.GetBin(value);
  }
};
```

Once you generate an array to be used as keys, you need to make a vtkm::worklet::Keys object. The Keys object is what will be passed to the invoke of the reduce by keys dispatcher. This of course happens in the control environment after calling the dispatcher for our worklet for generating the keys.

Example 14.12: Creating a vtkm::worklet::Keys object.

```
  vtkm::cont::ArrayHandle<vtkm::Id> binIds;
  vtkm::worklet::DispatcherMapField<IdentifyBins, DeviceAdapterTag>
      identifyDispatcher(bins);
  identifyDispatcher.Invoke(valuesArray, binIds);

  vtkm::worklet::Keys<vtkm::Id> keys(binIds, DeviceAdapterTag());
```

Now that we have our keys, we are finally ready for our reduce by key worklet. A histogram is simply a count of the number of elements in a bin. In this case, we do not really need any values for the keys. We just need the size of the bin, which can be identified with the internally calculated ValueCount.

A complication we run into with this histogram filter is that it is possible for a bin to be empty. If a bin is empty, there will be no key associated with that bin, and the reduce by key dispatcher will not call the worklet for that bin/key. To manage this case, we have to initialize an array with 0's and then fill in the non-zero entities with our reduce by key worklet. We can find the appropriate entry into the array by using the key, which is actually the bin identifier, which doubles as an index into the histogram. The following example gives the implementation for the reduce by key worklet that fills in positive values of the histogram.

Example 14.13: A reduce by key worklet to write histogram bin counts.

```
struct CountBins : vtkm::worklet::WorkletReduceByKey
{
  typedef void ControlSignature(KeysIn keys, WholeArrayOut<> binCounts);
  typedef void ExecutionSignature(_1, ValueCount, _2);
  using InputDomain = _1;

  template <typename BinCountsPortalType>
  VTKM_EXEC
  void operator()(vtkm::Id binId,
                  vtkm::IdComponent numValuesInBin,
                  BinCountsPortalType& binCounts) const
  {
    binCounts.Set(binId, numValuesInBin);
  }
};
```

The previous example demonstrates the basic usage of the reduce by key worklet to count common keys. A more common use case is to collect values associated with those keys, do an operation on those values, and provide a "reduced" value for each unique key. The following example demonstrates such an operation by providing a worklet that finds the average of all values in a particular bin rather than counting them.

Example 14.14: A worklet that averages all values with a common key.

```
struct BinAverage : vtkm::worklet::WorkletReduceByKey
{
  typedef void ControlSignature(KeysIn keys,
                                ValuesIn<> originalValues,
                                ReducedValuesOut<> averages);
  typedef _3 ExecutionSignature(_2);
  using InputDomain = _1;

  template <typename OriginalValuesVecType>
  VTKM_EXEC
  typename OriginalValuesVecType::ComponentType
  operator()(const OriginalValuesVecType& originalValues) const
  {
    typename OriginalValuesVecType::ComponentType sum = 0;
    for (vtkm::IdComponent index = 0;
         index < originalValues.GetNumberOfComponents();
         index++)
    {
      sum = sum + originalValues[index];
    }
    return sum/originalValues.GetNumberOfComponents();
  }
};
```

To complete the code required to average all values that fall into the same bin, the following example shows the full code required to invoke such a worklet. Note that this example repeats much of the previous examples, but shows it in a more complete context.

Example 14.15: Using a reduce by key worklet to average values falling into the same bin.

```
struct CombineSimilarValues
{
  struct IdentifyBins : vtkm::worklet::WorkletMapField
  {
    typedef void ControlSignature(FieldIn<Scalar> data,
                                  FieldOut<IdType> bins);
    typedef _2 ExecutionSignature(_1);
    using InputDomain = _1;

    BinScalars Bins;

    VTKM_CONT
    IdentifyBins(const BinScalars& bins)
      : Bins(bins)
    {  }

    VTKM_EXEC
    vtkm::Id operator()(vtkm::Float64 value) const
    {
      return Bins.GetBin(value);
    }
  };

  struct BinAverage : vtkm::worklet::WorkletReduceByKey
  {
    typedef void ControlSignature(KeysIn keys,
                                  ValuesIn<> originalValues,
                                  ReducedValuesOut<> averages);
    typedef _3 ExecutionSignature(_2);
    using InputDomain = _1;

    template <typename OriginalValuesVecType>
    VTKM_EXEC
    typename OriginalValuesVecType::ComponentType
    operator()(const OriginalValuesVecType& originalValues) const
    {
      typename OriginalValuesVecType::ComponentType sum = 0;
      for (vtkm::IdComponent index = 0;
           index < originalValues.GetNumberOfComponents();
           index++)
      {
        sum = sum + originalValues[index];
      }
      return sum/originalValues.GetNumberOfComponents();
    }
  };

  template <typename InArrayHandleType, typename DeviceAdapterTag>
  VTKM_CONT
  static
  vtkm::cont::ArrayHandle<typename InArrayHandleType::ValueType>
  Run(const InArrayHandleType &valuesArray, vtkm::Id numBins, DeviceAdapterTag)
  {
    VTKM_IS_ARRAY_HANDLE(InArrayHandleType);

    using ValueType = typename InArrayHandleType::ValueType;

    vtkm::Range range =
        vtkm::cont::ArrayRangeCompute(valuesArray).GetPortalConstControl().Get(0);
    BinScalars bins(range, numBins);

    vtkm::cont::ArrayHandle<vtkm::Id> binIds;
    vtkm::worklet::DispatcherMapField<IdentifyBins, DeviceAdapterTag>
        identifyDispatcher(bins);
    identifyDispatcher.Invoke(valuesArray, binIds);

    vtkm::worklet::Keys<vtkm::Id> keys(binIds, DeviceAdapterTag());

    vtkm::cont::ArrayHandle<ValueType> combinedValues;

    vtkm::worklet::DispatcherReduceByKey<BinAverage, DeviceAdapterTag>
        averageDispatcher;
    averageDispatcher.Invoke(keys, valuesArray, combinedValues);

    return combinedValues;
  }
};
```

17.2 Combining Like Elements

Our motivating example in Section 17.1 created a cell set with a line element representing each edge in some input data set. However, on close inspection there is a problem with our algorithm: it is generating a lot of duplicate elements. The cells in a typical mesh are connected to each other. As such, they share edges with each other. That is, the edge of one cell is likely to also be part of one or more other cells. When multiple cells contain the same edge, the algorithm we created in Section 17.1 will create multiple overlapping lines, one for each cell using the edge, as demonstrated in Figure 17.1. What we really want is to have one line for every edge in the mesh rather than many overlapping lines.

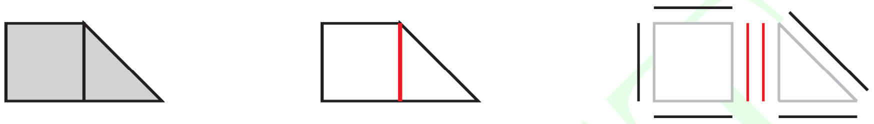


Figure 17.1: Duplicate lines from extracted edges. Consider the small mesh at the left comprising a square and a triangle. If we count the edges in this mesh, we would expect to get 6. However, our naïve implementation in Section 17.1 generates 7 because the shared edge (highlighted in red in the wireframe in the middle) is duplicated. As seen in the exploded view at right, one line is created for the square and one for the triangle.

In this section we will re-implement the algorithm to generate a wireframe by creating a line for each edge, but this time we will merge duplicate edges together. Our first step is the same as before. We need to count the number of edges in each input cell and use those counts to create a `vtkm::worklet::ScatterCounting` for subsequent worklets. Counting the edges is a simple worklet.

Example 17.5: A simple worklet to count the number of edges on each cell.

```
struct CountEdges : vtkm::worklet::WorkletMapPointToCell
{
  typedef void ControlSignature(CellSetIn cellSet, FieldOut<> numEdges);
  typedef _2 ExecutionSignature(CellShape, PointCount);
  using InputDomain = _1;

  template<typename CellShapeTag>
  VTKM_EXEC_CONT
  vtkm::IdComponent operator()(CellShapeTag shape,
                               vtkm::IdComponent numPoints) const
  {
    return vtkm::exec::CellEdgeNumberOfEdges(numPoints, shape, *this);
  }
};
```

In our previous version, we used the count to directly write out the lines. However, before we do that, we want to identify all the unique edges and identify which cells share this edge. This grouping is exactly the function that the reduce by key worklet type (described in Section 14.5.3 is designed to accomplish. The principal idea is to write a "key" that uniquely identifies the edge. The reduce by key worklet can then group the edges by the key and allow you to combine the data for the edge.

Thus, our goal of finding duplicate edges hinges on producing a key where two keys are identical if and only if the edges are the same. One straightforward key is to use the coordinates in 3D space by, say, computing the midpoint of the edge. The main problem with using point coordinates approach is that a computer can hold a point coordinate only with floating point numbers of limited precision. Computer floating point computations are notorious for providing slightly different answers when the results should be the same. For example, if an edge as endpoints at p_1 and p_2 and two different cells compute the midpoint as $(p_1+p_2)/2$ and $(p_2+p_1)/2$, respectively, the answer is likely to be slightly different. When this happens, the keys will not be the same and we will still produce 2 edges in the output.

Fortunately, there is a better choice for keys based on the observation that in the original cell set each edge is specified by endpoints that each have unique indices. We can combine these 2 point indices to form a "canonical" descriptor of an edge (correcting for order).[1] VTK-m comes with a helper function, vtkm::exec::CellEdgeCanonicalId, defined in vtkm/exec/CellEdge.h to produce these unique edge keys as vtkm::Id2s. Our second worklet produces these canonical edge identifiers.

Example 17.6: Worklet generating canonical edge identifiers.

```
class EdgeIds : public vtkm::worklet::WorkletMapPointToCell
{
public:
  typedef void ControlSignature(CellSetIn cellSet,
                                FieldOut<> canonicalIds);
  typedef void ExecutionSignature(CellShape shape,
                                  PointIndices globalPointIndices,
                                  VisitIndex localEdgeIndex,
                                  _2 canonicalIdOut);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  VTKM_CONT
  ScatterType GetScatter() const { return this->Scatter; }

  VTKM_CONT
  explicit EdgeIds(const ScatterType& scatter)
    : Scatter(scatter)
  {  }

  template <typename CellShapeTag, typename PointIndexVecType>
  VTKM_EXEC
  void operator()(CellShapeTag shape,
                  const PointIndexVecType& pointIndices,
                  vtkm::IdComponent localEdgeIndex,
                  vtkm::Id2& canonicalIdOut) const
  {
    canonicalIdOut =
        vtkm::exec::CellEdgeCanonicalId(pointIndices.GetNumberOfComponents(),
                                        localEdgeIndex,
                                        shape,
                                        pointIndices,
                                        *this);
  }

private:
  ScatterType Scatter;
};
```

Our third and final worklet generates the line cells by outputting the indices of each edge. As hinted at earlier, this worklet is a reduce by key worklet (inheriting from vtkm::worklet::WorkletReduceByKey). The reduce by key dispatcher will collect the unique keys and call the worklet once for each unique edge. Because there is no longer a consistent mapping from the generated lines to the elements of the input cell set, we need pairs of indices identifying the cells/edges from which the edge information comes. We use these indices along with a connectivity structure produced by a WholeCellSetIn to find the information about the edge. As shown later, these indices of cells and edges can be extracted from the ScatterCounting used to executed the worklet back in Example 17.6.

As we did in Section 17.1, this worklet writes out the edge information in a vtkm::Vec<vtkm::Id,2> (which in some following code will be created with an ArrayHandleGroupVec).

[1]Using indices to find common mesh elements is described by Miller et al. in "Finely-Threaded History-Based Topology Computation" (in *Eurographics Symposium on Parallel Graphics and Visualization*, June 2014).

Example 17.7: A worklet to generate indices for line cells from combined edges.

```
class EdgeIndices : public vtkm::worklet::WorkletReduceByKey
{
public:
  typedef void ControlSignature(KeysIn keys,
                                WholeCellSetIn<> inputCells,
                                ValuesIn<> originCells,
                                ValuesIn<> originEdges,
                                ReducedValuesOut<> connectivityOut);
  typedef void ExecutionSignature(_2 inputCells,
                                  _3 originCell,
                                  _4 originEdge,
                                  _5 connectivityOut);
  using InputDomain = _1;

  template <typename CellSetType,
            typename OriginCellsType,
            typename OriginEdgesType>
  VTKM_EXEC
  void operator()(const CellSetType& cellSet,
                  const OriginCellsType& originCells,
                  const OriginEdgesType& originEdges,
                  vtkm::Id2& connectivityOut) const
  {
    // Regardless of how many cells/edges are in the input, we know they are
    // all the same, so just pick the first one.
    vtkm::Vec<vtkm::IdComponent, 2> localEdgeIndices =
        vtkm::exec::CellEdgeLocalIndices(
          cellSet.GetNumberOfIndices(originCells[0]),
          originEdges[0],
          cellSet.GetCellShape(originCells[0]),
          *this);
    auto pointIndices = cellSet.GetIndices(originCells[0]);
    connectivityOut[0] = pointIndices[localEdgeIndices[0]];
    connectivityOut[1] = pointIndices[localEdgeIndices[1]];
  }
};
```

Did you know?

It so happens that the vtkm::Id2s generated by CellEdgeCanonicalId contain the point indices of the two endpoints, which is enough information to create the edge. Thus, in this example it would be possible to forgo the steps of looking up indices through the cell set. That said, this is more often not the case, so for the purposes of this example we show how to construct cells without depending on the structure of the keys.

With these 3 worklets, it is now possible to generate all the information we need to fill a vtkm::cont::-CellSetSingleType object. A CellSetSingleType requires 4 items: the number of points, the constant cell shape, the constant number of points in each cell, and an array of connection indices. The first 3 items are trivial. The number of points can be taken from the input cell set as they are the same. The cell shape and number of points are predetermined to be line and 2, respectively.

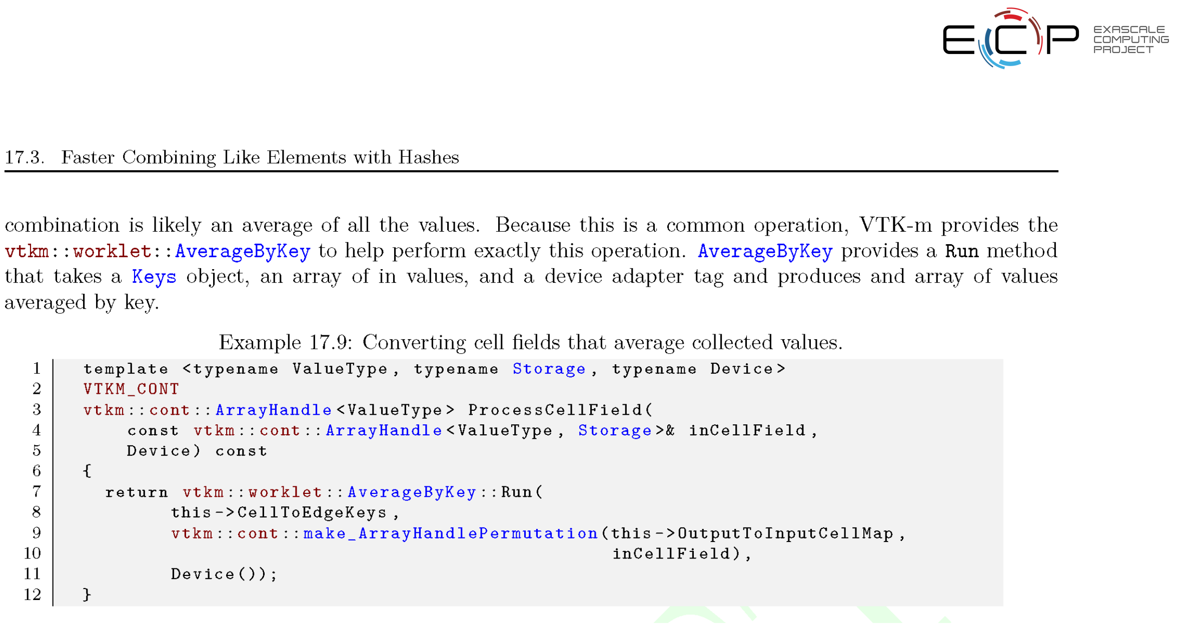
The last item, the array of connection indices, is what we are creating with the worklet in Example 17.7. The connectivity array for CellSetSingleType is expected to be a flat array of vtkm::Id indices, but the worklet needs to provide groups of indices for each cell (in this case as a Vec object). To reconcile what the worklet provides and what the connectivity array must look like, we use the vtkm::cont::ArrayHandleGroupVec fancy array handle (described in Section 10.2.11) to make a flat array of indices look like an array of Vec objects. The following example shows a **Run** method in a worklet helper class. Note the use of make_ArrayHandleGroupVec when calling the **Invoke** method to make this conversion.

Example 17.8: Invoking worklets to extract unique edges from a cell set.

```
template<typename CellSetType, typename Device>
VTKM_CONT
vtkm::cont::CellSetSingleType<> Run(const CellSetType& inCellSet, Device)
{
  VTKM_IS_DYNAMIC_OR_STATIC_CELL_SET(CellSetType);

  // First, count the edges in each cell.
  vtkm::cont::ArrayHandle<vtkm::IdComponent> edgeCounts;
  vtkm::worklet::DispatcherMapTopology<CountEdges, Device> countEdgeDispatcher;
  countEdgeDispatcher.Invoke(inCellSet, edgeCounts);

  vtkm::worklet::ScatterCounting scatter(edgeCounts, Device());
  this->OutputToInputCellMap =
      scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells());
  vtkm::worklet::ScatterCounting::VisitArrayType outputToInputEdgeMap =
      scatter.GetVisitArray(inCellSet.GetNumberOfCells());

  // Second, for each edge, extract a canonical id.
  vtkm::cont::ArrayHandle<vtkm::Id2> canonicalIds;

  vtkm::worklet::DispatcherMapTopology<EdgeIds, Device>
      edgeIdsDispatcher((EdgeIds(scatter)));
  edgeIdsDispatcher.Invoke(inCellSet, canonicalIds);

  // Third, use a Keys object to combine all like edge ids.
  this->CellToEdgeKeys = vtkm::worklet::Keys<vtkm::Id2>(canonicalIds, Device());

  // Fourth, use a reduce-by-key to extract indices for each unique edge.
  vtkm::cont::ArrayHandle<vtkm::Id> connectivityArray;
  vtkm::worklet::DispatcherReduceByKey<EdgeIndices, Device> edgeIndicesDispatcher;
  edgeIndicesDispatcher.Invoke(
        this->CellToEdgeKeys,
        inCellSet,
        this->OutputToInputCellMap,
        outputToInputEdgeMap,
        vtkm::cont::make_ArrayHandleGroupVec<2>(connectivityArray));

  // Fifth, use the created connectivity array to build a cell set.
  vtkm::cont::CellSetSingleType<> outCellSet(inCellSet.GetName());
  outCellSet.Fill(inCellSet.GetNumberOfPoints(),
                  vtkm::CELL_SHAPE_LINE,
                  2,
                  connectivityArray);

  return outCellSet;
}
```

Another feature to note in Example 17.8 is that the method calls **GetOutputToInputMap** on the Scatter object it creates and squirrels it away for later use. It also saves the vtkm::worklet::Keys object created for later user. The reason for this behavior is to implement mapping fields that are attached to the input cells to the indices of the output. In practice, these worklet are going to be called on DataSet objects to create new DataSet objects. The method in Example 17.8 creates a new CellSet, but we also need a method to transform the Fields on the data set. The saved **OutputToInputCellMap** array and Keys object allow us to transform input fields to output fields.

The following example shows another convenience method that takes these saved objects and converts an array from an input cell field to an output cell field array. Because in general there are several cells that contribute to each edge/line in the output, we need a method to combine all these cell values to one. The most appropriate



combination is likely an average of all the values. Because this is a common operation, VTK-m provides the `vtkm::worklet::AverageByKey` to help perform exactly this operation. `AverageByKey` provides a `Run` method that takes a `Keys` object, an array of in values, and a device adapter tag and produces and array of values averaged by key.

Example 17.9: Converting cell fields that average collected values.

```
template <typename ValueType, typename Storage, typename Device>
VTKM_CONT
vtkm::cont::ArrayHandle<ValueType> ProcessCellField(
    const vtkm::cont::ArrayHandle<ValueType, Storage>& inCellField,
    Device) const
{
  return vtkm::worklet::AverageByKey::Run(
        this->CellToEdgeKeys,
        vtkm::cont::make_ArrayHandlePermutation(this->OutputToInputCellMap,
                                                inCellField),
        Device());
}
```

[MS-17/04] SPATIAL DIVISION

This activity has the following completion criteria:

- Implementation of the algorithms to build spatial division structures are merged to the master branch of the central VTK-m repository.
- Documentation of the spatial division structures, how to build them, and their use are added to the VTK-m User's Guide working document. (Note that the implementation of this milestone is mostly in support of implementing milestones [MS-18/06] and [MS-18/07], so the interface and documentation at this point might be sparse with complete documentation for these later milestones.)

Merge into VTK-m repository

The VTK-m project uses "merge requests" as its mechanism for adding code to the master VTK-m repository. The gitlab repository manages the review process and saves the information for it. The following completed merge requests were used to add the code for this activity.

- Scan by Key: The following merge requests was based on the work of Reduce by Key and extended VTK-m to perform segmented scan operations, both inclusive and exclusive scan are available.
  - MR !746, ScanbyKey, https://gitlab.kitware.com/vtk/vtk-m/merge_requests/746
  - MR !754, Documentation on ScanByKey, https://gitlab.kitware.com/vtk/vtk-m/merge_requests/754
- ArrayHandleReverse: The following merge requests added a new kind of Fancy Array Handle that reverses the order of traversal of the underlying Array Handle.
  - MR !763, Added ArrayHandleReverse, https://gitlab.kitware.com/vtk/vtk-m/merge_requests/763
  - MR !764, Documentation on ArrayHandleReverse, https://gitlab.kitware.com/vtk/vtk-m/merge_requests/764

    - MR !783, Enable writing to ArrayHandleReverse, https://gitlab.kitware.com/vtk/vtk-m/merge_requests/783
- KD-Tree: The following merge request added implementation of the construction of KD-Tree and nearest neighbor search, based on Chris Sewell et.al. LDAV 13 paper.
    - MR !797, Add 3d Kdtree and nearest neighbor search. https://gitlab.kitware.com/vtk/vtk-m/merge_requests/797

Documentation

Currently documentation of the implemented features are in the form of Doxygen comments and unit tests. The following is an excerpt of the Doxygen created for the KdTree3D class.

vtkm::worklet::KdTree3D Class Reference

```
#include <KdTree3D.h>
```

Public Member Functions

KdTree3D ()=default

template<typename CoordType , typename CoordStorageTag , typename DeviceAdapter >
void **Build** (const **vtkm::cont::ArrayHandle**< **vtkm::Vec**< CoordType, 3 >, CoordStorageTag > &coords, DeviceAdapter device)
Construct a 3D KD-tree for 3D point positions. More...

template<typename CoordType , typename CoordStorageTag1 , typename CoordStorageTag2 , typename DeviceAdapter >
void **Run** (const **vtkm::cont::ArrayHandle**< **vtkm::Vec**< CoordType, 3 >, CoordStorageTag1 > &coords, const **vtkm::cont::ArrayHandle**< **vtkm::Vec**< CoordType, 3 >, CoordStorageTag2 > &queryPoints, **vtkm::cont::ArrayHandle**< vtkm::Id > &nearestNeighborIds, **vtkm::cont::ArrayHandle**< CoordType > &distances, DeviceAdapter device)
Nearest neighbor search using KD-Tree. More...

Private Attributes

vtkm::cont::ArrayHandle< vtkm::Id > **PointIds**
vtkm::cont::ArrayHandle< vtkm::Id > **SplitIds**

Constructor & Destructor Documentation

◆ KdTree3D()

vtkm::worklet::KdTree3D::KdTree3D ( ) default

Member Function Documentation

◆ Build()

template<typename CoordType , typename CoordStorageTag , typename DeviceAdapter >
void vtkm::worklet::KdTree3D::Build (const **vtkm::cont::ArrayHandle**< **vtkm::Vec**< CoordType, 3 >, CoordStorageTag > & coords,
DeviceAdapter device
) inline

Construct a 3D KD-tree for 3D point positions.

Template Parameters

CoordType	type of the x, y, z component of the point coordinates.
CoordStorageTag	
DeviceAdapter	

Parameters

coords	An ArrayHandle of x, y, z coordinates of input points.
device	Tag for selecting device adapter.

◆ Run()

```
template<typename CoordType , typename CoordStorageTag1 , typename CoordStorageTag2 , typename DeviceAdapter >
void
vtkm::worklet::KdTree3D::Run    ( const vtkm::cont::ArrayHandle< vtkm::Vec< CoordType, 3 >, CoordStorageTag1 > &  coords,
                                  const vtkm::cont::ArrayHandle< vtkm::Vec< CoordType, 3 >, CoordStorageTag2 > &  queryPoints,
                                  vtkm::cont::ArrayHandle< vtkm::Id > &                                           nearestNeighborIds,
                                  vtkm::cont::ArrayHandle< CoordType > &                                          distances,
                                  DeviceAdapter                                                                   device
                                )                                                                                                  inline
```

Nearest neighbor search using KD-Tree.

Parallel search of nearest neighbor for each point in the `queryPoints` in the the set of `coords`. Returns nearest neighbor in `nearestNeighborId` and distance to nearest neighbor in `distances`.

Template Parameters

- CoordType
- CoordStorageTag1
- CoordStorageTag2
- DeviceAdapter

Parameters

coords	Point coordinates for training data set (haystack)
queryPoints	Point coordinates to query for nearest neighbor (needles).
nearestNeighborIds	Nearest neighbor in the traning data set for each points in the testing set
distances	Distances between query points and their nearest neighbors.
device	Tag for selecting device adapter.

Member Data Documentation

◆ PointIds

vtkm::cont::ArrayHandle<vtkm::Id> vtkm::worklet::KdTree3D::PointIds private

◆ SplitIds

vtkm::cont::ArrayHandle<vtkm::Id> vtkm::worklet::KdTree3D::SplitIds private

The documentation for this class was generated from the following file:

- **KdTree3D.h**

Generated by doxygen 1.8.13

[MS-17/05] ADVECT STEADY STATE

This activity has the following completion criteria:

- Implementation is merged to the master branch of the central VTK-m repository.
- Documentation is added to the VTK-m User's Guide working document.

Merge into VTK-m repository

The VTK-m project uses "merge requests" as its mechanism for adding code to the master VTK-m repository. The gitlab repository manages the review process and saves the information for it. The following completed merge requests were used to add the code for this activity.

- https://gitlab.kitware.com/vtk/vtk-m/merge_requests/930
- https://gitlab.kitware.com/vtk/vtk-m/merge_requests/922
- https://gitlab.kitware.com/vtk/vtk-m/merge_requests/877
- https://gitlab.kitware.com/vtk/vtk-m/merge_requests/852
- https://gitlab.kitware.com/vtk/vtk-m/merge_requests/845
- https://gitlab.kitware.com/vtk/vtk-m/merge_requests/843
- https://gitlab.kitware.com/vtk/vtk-m/merge_requests/828
- https://gitlab.kitware.com/vtk/vtk-m/merge_requests/949

Documentation

The following excerpt from the user's guide document this new functionality.

4.4.3 Streamlines

Streamlines are a powerful technique for the visualization of flow fields. A streamline is a curve that is parallel to the velocity vector of the flow field. Individual streamlines are computed from an initial point location (seed) using a numerical method to integrate the point through the flow field.

In addition to the standard `Execute` method, `Streamline` provides the following methods.

`SetSeeds` Set the seed locations for the streamlines.

`SetStepSize` Set the step size used for the numerical integrator (4^{th} order Runge-Kutta method) to integrate the seed locations through the flow field.

`SetNumberOfSteps` Set the number of integration steps to be performed on each streamline.

Example 4.4: Using `Streamlines`, which is a data set with field filter.

```
vtkm::filter::Streamlines streamlines;

// Specify the seeds.
std::vector<vtkm::Vec<vtkm::FloatDefault,3>> seeds(2);
seeds[0] = vtkm::Vec<vtkm::FloatDefault,3>(0,0,0);
seeds[1] = vtkm::Vec<vtkm::FloatDefault,3>(1,1,1);

vtkm::cont::ArrayHandle<vtkm::Vec<vtkm::FloatDefault,3>> seedArray;
seedArray = vtkm::cont::make_ArrayHandle(seeds);

streamlines.SetStepSize(0.1);
streamlines.SetNumberOfSteps(100);
streamlines.SetSeeds(seedArray);

vtkm::filter::Result result = streamlines.Execute(inData, "vectorvar");

if (!result.IsDataSetValid())
{
  throw vtkm::cont::ErrorBadValue("Failed to run Streamlines.");
}

vtkm::cont::DataSet streamlineCurves = result.GetDataSet();
```

[MS-17/06] SMOOTH SURFACE NORMALS

This activity has the following completion criteria:

- Implementation is merged to the master branch of the central VTK-m repository.
- Documentation is added to the VTK-m User's Guide working document.

Merge into VTK-m repository

The VTK-m project uses "merge requests" as its mechanism for adding code to the master VTK-m repository. The gitlab repository manages the review process and saves the information for it. The following completed merge requests were used to add the code for this activity.

- MR !798, Add SmoothSurfaceNormals worklet, https://gitlab.kitware.com/vtk/vtk-m/merge_requests/798

Documentation

The following excerpt from the user's guide document this new functionality.

4.2.4 Surface Normals

vtkm::filter::SurfaceNormals computes the surface normals of a polygonal data set at its points and/or cells. The filter takes a data set as input and by default, uses the active coordinate system to compute the normals. Optionally, a coordinate system or a point field of 3d vectors can be explicitly provided to the `Execute` method. The cell normals are computed based on each cell's winding order using vector cross-product. For non-polygonal cells, a zeroed vector is assigned. The point normals are computed by averaging the cell normals of the incident cells of each point.

The default name for the output fields is "Normals", but that can be overriden using the `SetCellNormalsName` and `SetPointNormalsName` methods.

In addition to the standard field filters methods, `SurfaceNormals` provides the following methods.

`SetGenerateCellNormals/GetGenerateCellNormals` These methods can be used to set/get the flag to specify if the cell normals should be generated.

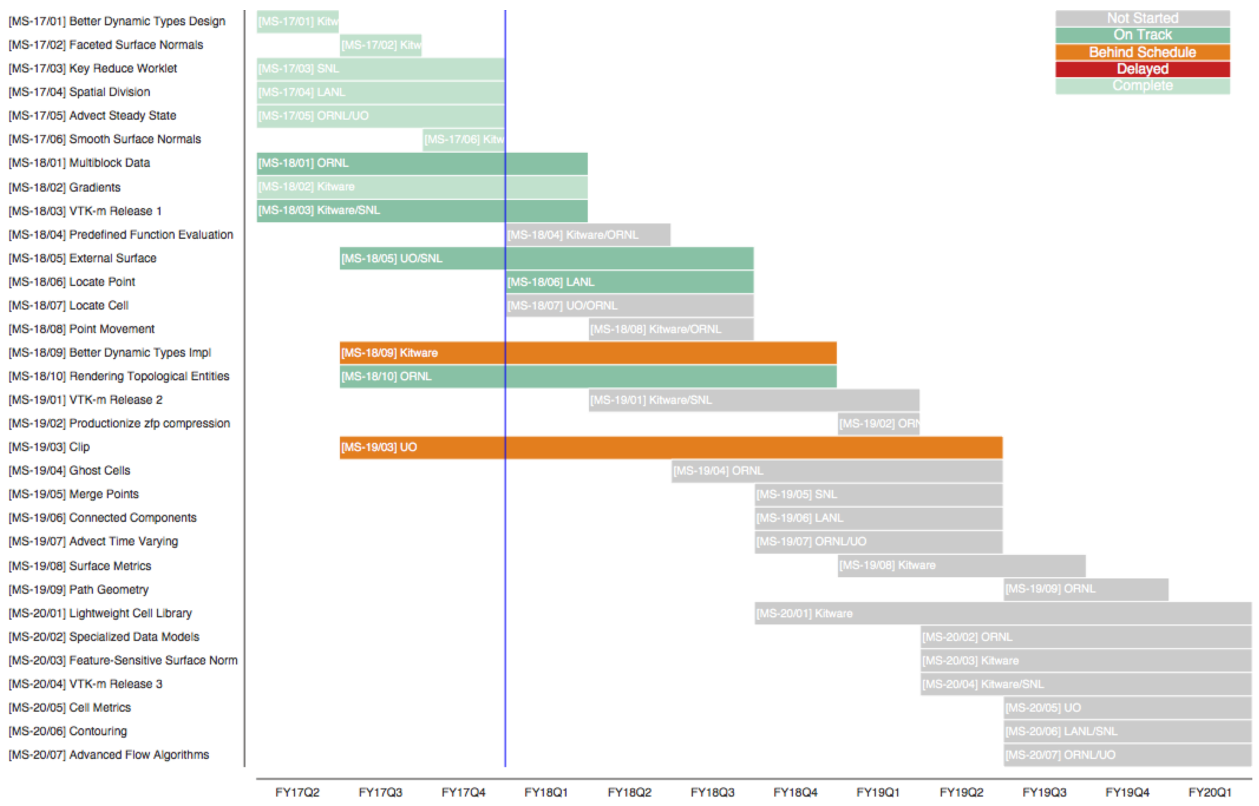
`SetGeneratePointNormals/GetGeneratePointNormals` These methods can be used to set/get the flag to specify if the point normals should be generated.

`SetCellNormalsName/GetCellNormalsName` These methods can be used to set/get the output cell normals field name.

`SetPointNormalsName/GetPointNormalsName` These methods can be used to set/get the output point normals field name.

4. CONCLUSIONS AND FUTURE WORK

The 4 activities associated with this milestone were all completed on time according to their completion criteria. The following diagram shows our current progress with the overall project.



ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.