Optimizing Threshold for Extreme Scale Analysis

Robert Maynard[a], Kenneth Moreland[b],

Utkarsh Ayachit[a], Berk Geveci[a],

and Kwan-Liu Ma[c]

[a]Kitware, Inc.
[b]Sandia National Labaratories.
[e]University of California at Davis.

ABSTRACT

As the HPC community starts focusing its efforts towards exascale, it becomes clear that we are looking at machines with a billion way concurrency. Although parallel computing has been at the core of the performance gains achieved until now, scaling over 1,000 times the current concurrency can be challenging. As discussed in this paper, even the smallest memory access and synchronization overheads can cause major bottlenecks at this scale. As we develop new software and adapt existing algorithms for exascale, we need to be cognizant of such pitfalls. In this paper, we document our experience with optimizing a fairly common and parallelizable visualization algorithm, threshold of cells based on scalar values, for such highly concurrent architectures. Our experiments help us identify design patterns that can be generalized for other visualization algorithms as well. We discuss our implementation within the Dax toolkit, which is a framework for data analysis and visualization at extreme scale. The Dax toolkit employs the patterns discussed here within the framework's scaffolding to make it easier for algorithm developers to write algorithms without having to worry about such scaling issues.

Keywords: Software,Programming Techniques, Concurrent Programming

1. INTRODUCTION

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. Although the concurrency is bound to increase by over 1,000 times, the memory increase is expected to be only 30 to 500 fold .[1] Additionally, many forecasts predict the use of heterogeneous architectures containing accelerators of a design similar to that in GPU processors today. This change in processor design has dramatic effects on the design of large-scale parallel programs. Based on several recent studies ,[1–3] 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.

To make it easier to write visualization and analysis algorithms for an exascale machine within the restrictions posed by the high concurrency, and possibly heterogeneous architecture, we have been developing the Dax (*D*ata *A*nalysis at e*X*treme) toolkit .[4] The Dax toolkit is designed to encapsulate the complexity of multi-threaded visualization and data analysis algorithms. Dax provides *pervasive parallelism* throughout all its visualization algorithms. It does this by providing programming constructs called *worklets* that operate on a very fine granularity of the data, thereby removing the onus from the visualization algorithm developer. In absence of an exascale machine, our initial implementation targets GPUs and multi-core CPUs using NVIDIA's CUDA programming environment[5] and OpenMP[6] respectively.

Further author information: (Send correspondence to R.M.A.)
R.M.A.: E-mail: robert.maynard@kitware.com
K.M.A.: E-mail: kmorel@sandia.gov
U.A.A.: E-mail: utkarsh.ayachit@kitware.com
B.G.A.: E-mail: berk.geveci@kitware.com K.L.A.: E-mail: ma@cs.ucdavis.edu

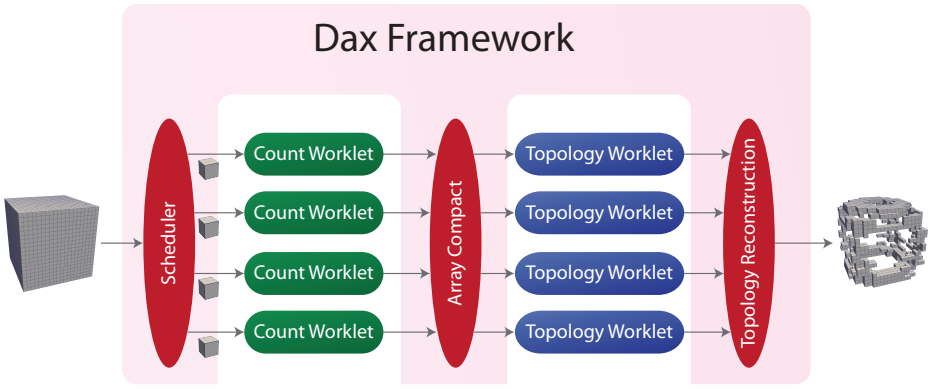


Figure 1: Overview of the various stages involved in the parallel implementation of the algorithm for thresholding cells implemented within the Dax framework.

In this paper, we document our experience with migrating a commonly used data processing algorithm, thresholding of cells, to the Dax framework. We dissect the thresholding algorithm into passes that can be parallelized and evaluate the benefits and drawbacks for different approaches for each of these stages. Most of these optimizations are abstracted away in the design of the Dax framework itself, in keeping with our original goal of freeing the algorithm developer of the concurrency concerns.

2. RELATED WORK

Using GPUs for general purpose computing has been extensively studied in recent years. Since several of the constraints presented to the developers on the GPUs are similar to what we expect on an exascale machine, we use GPUs as a proxy for future hardware. We leverage strategies discussed by some of this past work in our case study.

Thrust[7] provides a common interface to a collection of data parallel algorithms that operate on multiple back-ends including CUDA,[5] OpenMP,[6] and Intel Threading Building Blocks (TBB).[8] Our implementation within Dax uses several of these algorithms within the framework including copy_if, sort, upper and lower bounds, for_each, reduce, unique, and inclusive_scan.

Stream compaction is a crucial step in our threshold implementation. Horn[9] and Sengupta et al.[10] describe some of the early attempts to implement stream compaction on the GPU. Lo et al.[11] describes a GPU based thresholding implementation that uses stream compaction. Our implementation combines the ideas described by by Sengupta[10] and Lo,[11] respectively.

The design of our topology and field information was influenced by the previous work done by VTK[12] to design a simple and concise mesh representation, which was designed mainly for performance and readibility. The design is also influenced by the work done by Solano,[13] which shows the importance of how point and cell properties are associated with topology.

The development of the HistoPyramids lookup structure[14] and the research done by Lo et al.[11] are the basis of how we build the topological information needed for a system that maps each output cell index to the original input cell index.

3. BACKGROUND

This paper documents our experience with migrating the thresholding algorithm to the Dax toolkit. We begin with an overview of the Dax toolkit and the thresholding algorithm.

3.1 Dax Toolkit

As mentioned earlier, the Dax toolkit supports fine-grained concurrency for data analysis and visualization algorithms required to drive exascale computing. The basic computational unit of the Dax toolkit is a *worklet*, a function that implements the algorithm operating on a single element of the input mesh. This element can be a single point, edge, face, or cell or it could potentially be a small local neighborhood. The worklet is constrained to be serial and stateless; it can access only the

element passed to and from the invocation. With this constraint, the serial worklet function can be concurrently scheduled on an unlimited number of threads without worrying about shared resource access, race conditions, and incorrect memory access.

Dax separates the programming environment into two components: execution and control. The execution environment is the API exposed to developers that write worklets for different visualization and analysis algorithms. The control environment is the scaffolding that manages scheduling and execution of these worklets in parallel. By providing different implementations for the device dependent component of the control environment, which we call the *device adaptor*, we are able to support different architectures including GPUs using CUDA, multi-core CPUs using OpenMP and even a serial implementation for debugging. In this paper, we compare the performance results on these different devices to determine if certain optimizations favor certain architectures more than the others.

The following code segment is a simple worklet that estimates gradients using finite differences. Note some of the key features of the worklet. The code uses concepts familiar to visualization programmers like points, cells, and fields. The worklet is also devoid of any parallel or device-dependent constructs. These features make worklet easy to create and understand.

```
DAX_WORKLET void CellGradient(
    DAX_IN dax::exec::WorkMapCell& work,
    DAX_IN dax::exec::FieldCoordinates points,
    DAX_IN dax::exec::FieldPoint& point_attribute,
    DAX_OUT dax::exec::FieldCell& cell_attribute)
{
  dax::exec::Cell cell(work);
  dax::Vector3 parametric_cell_center
    = dax::make_Vector3(0.5, 0.5, 0.5);

  dax::Vector3 value = cell.Derivative(
    parametric_cell_center,
    points,
    point_attribute,
    0);
  cell_attribute.Set(work, value);
}
```

A worklet is constrained by the execution environment API so that it cannot access the mesh in its entirety but rather a small neighborhood. This allows Dax to manage scheduling of the worklets independent of the worklets themselves. Since the access to mesh elements and fields is via the execution environment API alone, we can easily enforce safe and efficient access to these data items. We discuss some of the performance benefits gained by making such choices in the API design in this paper.

Dax's data model is loosely based on VTK's data model.[12] Unlike VTK however, Dax decouples the mesh topology and geometry from the attributes associated with the cells, points, or other primitives in the mesh. This decoupling allows worklets to operate on any field data without being concerned about the mesh type.

To define the geometry and topology for an unstructured grid, Dax uses two arrays: an array of floating-point 3-tuples for the coordinates of each vertex and an array of indices listing the vertices comprising each cell. Although this implies that we can only support a single cell type in the unstructured grid, it allows for easier lookup of the points of a given cell. The work done by Solano-Quinde[13] is the basis of our design.

The uniform grid is even simpler than the unstructured grid. Because it only supports voxel cells, it allows the entire topology to be stored by knowing the origin, spacing, and extents of the grid, similar to VTK's implementation. Using this meta-data, one can easily compute the point coordinates for any point in a cell. This allows the uniform grid to use very little memory, in exchange for computation time.

3.2 Threshold

For our exercise, we pick a well understood and fairly common data analysis task of thresholding a uniform grid of voxel cells, transforming it into a unstructured grid of hexahedron cells that match a given range on a point scalar field. We use

Figure 2: Comparison of point field access worklets between (a) which uses manual iteration and (b) which applies a function to each element.

(a)

```
template<typename CellType, typename FieldType>
DAX_WORKLET void Threshold(
    DAX_IN dax::exec::WorkCountCell<CellType> &work,
    DAX_IN FieldType thresholdMin,
    DAX_IN FieldType thresholdMax,
    DAX_IN dax::exec::FieldPoint<FieldType> &inField)
{
  char valid = 1;
  for (dax::Id vertexId = 0;
       vertexId < CellType::NUM_POINTS;
       vertexId++)
    {
    //apply the threshold function to each point field value
    FieldType inValue = work.GetFieldValue(inField, vertexId);
    valid &= ((inValue >= thresholdMin) && (inValue <= thresholdMax));
    }
  work.SetCellCount(valid);
}
```

(b)

```
template<typename CellType, typename FieldType>
DAX_WORKLET void Threshold(
    DAX_IN dax::exec::WorkCountCell<CellType> &work,
    DAX_IN FieldType thresholdMin,
    DAX_IN FieldType thresholdMax,
    DAX_IN dax::exec::FieldPoint<FieldType> &inField)
{
  dax::Tuple<FieldType,CellType::NUM_POINTS> fieldValues = work.GetFieldValues(inField);
  ThresholdFunction<FieldType> threshold(thresholdMin, thresholdMax);
  // use vector fore each to apply the threshold function
  dax::exec::VectorForEach(fieldValues,threshold);
  work.SetCellCount(thresholdFunc.valid);
}
```

thresholding on a point field as it a more complicated task for parallelization as we need to access point field values, which are shared among cells, to properly classify each cell.

The threshold algorithm can be summarized as follows: For each cell in the dataset, 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 dataset. 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 require more space than needed and that the resultant dataset is suitable for further analysis, if needed.

4. PARALLELIZING THRESHOLD

In this section we describe our experience with parallelizing the algorithm of thresholding cells based on a point field criteria. We describe the different stages in a fine-grained parallel implementation of the algorithm and the approaches we experimented with at each stage and resulting performance changes.

To implement the algorithm within the Dax framework, we need to map the algorithm to a worklet. As described in Section 3.1, worklet is a function that implements the algorithm on a single element of the input mesh. Based on the implementation in ,[11] the thresholding operation can be characterized as worklets as follows:

1. For every cell in the input dataset, we need to first determine if it passes the threshold criteria. This can be implemented as a *count worklet* whose output is both the number of cells it will create and the which existing points for the cell. In the case of threshold, this will either be 0 or 1 cell, and none or all of the original points. We schedule the count worklet over the entire input domain to generate two mask arrays with cell and point counts, respectively.

2. Once we have generated the cell count array, we need to identify the valid cells and and purge the others. This is a fairly common task in parallel programming, known as stream compaction. This stage provide us with information about how many hexahedrons will be in the final output.

3. Likewise, we need to identify the valid point coordinates and purge the others using stream compaction. This stage provides us with the valid coordinates that will be used in the final output

4. For every valid cell, we now need to generate the hexahedron. We launch a *topology worklet* that constructs the geometry and topology for the resulting unstructured grid.

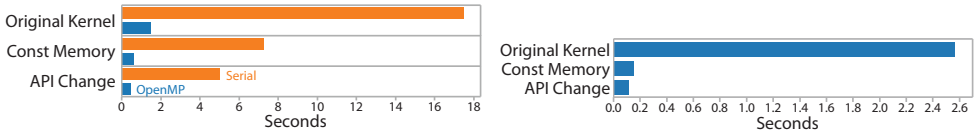
Figure 1 provides an overview of the implementation in Dax.

4.1 Efficient Cell Classification

The first step in the thresholding algorithm is to classify the cells using the count worklet. The count worklet relies on the ability to query the points for each cell to determine the scalar values for a given field. The concept is fairly simple; for each point in the cell look up the field value at that point. Since our input dataset is a uniform grid, we can implicitly compute the ids for the points that form a cell given the cell id. In our first approach, we designed the Dax API such that the user code is expected to call the *GetFieldValue(...)* function for every point in succession, Figure $2(a)$. On profiling this worklet on the GPU, we observed that the worklet was executing at 0.45 instructions per cycle, with only 38% of the GPU's warps active throughout the entire run. This implies that the worklet is not using the full potential of the GPU.

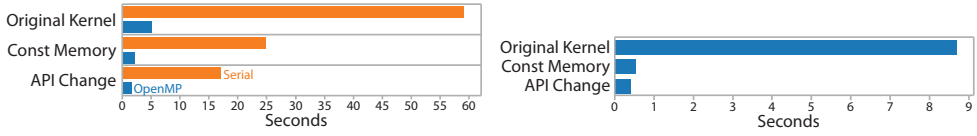
The second implementation of the worklet, Figure $2(b)$, is designed to remove manual iteration over the returned point field values. We achieve this by constraining the Dax toolkit API to return all point field values for a particular cell en masse instead of independently per point, using *GetFieldValues(...)*. We then provide functions that can be used to execute operations over the values returned, such *dax::exec::VectorForEach(...)*. On profiling this new worklet on the GPU we find it to execute at 1.45 instructions per cycle with 42% of the GPU's warps active over the entire run.

Figure 3 and Figure 4 compare the performance of the two versions on a uniform grid of size 512^3 and 768^3, respectively. We see that the version using GetFieldValues() shows over an order of magnitude speed increase using the new API over the version using GetFieldValue().



(a) Serial and OpenMP (b) CUDA

Figure 3: Results of the count worklet when executing on a grid size of 512^3. Figure 3a shows the results of the serial CPU implementation using Intel Xeon X5680 and parallel CPU implementation using OpenMP on dual Intel Xeon X5680 with a total of 12 cores. Figure 3b shows the results of the parallel GPU implementation using CUDA on Tesla C2050.



(a) Serial and OpenMP (b) CUDA

Figure 4: Results of the count worklet when executing on a grid size of 768^3. Figure 4a shows the results of the serial CPU implementation using Intel Xeon X5680 and the parallel CPU implementation using OpenMP on dual Intel Xeon X5680 with a total of 12 cores. Figure 4b shows the results of the parallel GPU implementation using CUDA on Tesla C2050.

We know that both worklets access the same global memory locations, i.e. the field values for all the points. The key is the rest of scaffolding code within GetFieldValue() and GetFieldValues(). For uniform grid, to compute the point id for a particular vertex for a cell, we use a lookup table to map the linear vertex id to i,j,k and then convert that to the offset to use to lookup the point field. The difference between the two versions is that this lookup table is initialized every time GetFieldValue() is called, hence for all the 8 points in the voxel, while in GetFieldValues() it is built just once. Thus, despite the fact that we were not accessing global memory differently, just initializing local variables can cause dramatic performance differences. We verified this conclusion by making the lookup table a global constant array, so that it is not initialized on every call to GetFieldValue(). In this case, we observed that the performance was indeed very similar to what we get from GetFieldValues(). This global constant array approach, however, is not practical since we cannot provide such array for every cell type.

The Dax toolkit only provides GetFieldValues() API since if a worklet is interested in one cell point, more likely than not, the worklet will also be interested in the other cell points. Thus users are forced to use a more performance friendly approach.

4.2 Stream Compaction

Stream compaction is a parallel primitive algorithm that is used to filter unwanted elements from an array based on if they pass a boolean predicate.[9] The stream compaction algorithm is implemented using a two step process. The first step counts the number of valid elements, and the second step moves the valid elements to the output array.

The result of the count worklet is the generation of two mask arrays, one that corresponds to points and one to the cells. The Dax toolkit provides a stream compaction algorithm that we use on these arrays to filter out invalid points and cells.

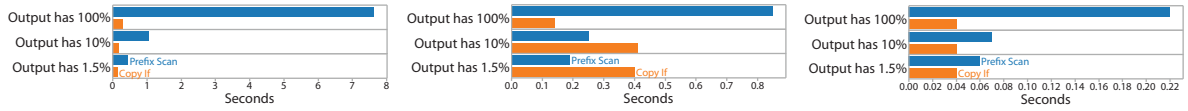
The classic approach to stream compaction is to use a prefix scan to determine the number of valid elements and the index that each element has in the output data.[15] Instead we use an accumulator to determine the number of valid cells and use a simpler conditional copy operation based on the binary flag known as copy_if,[7] for that cell. This implementation also helps minimize the memory footprint.

Figure 5 compares the performance of the two approaches on a mask array with 512^3 elements with 1.5%, 10%, and 100%, respectively, of the elements marked as valid and will be passing in the resulting compacted array. We note that copy_if is faster on the GPU and serial CPU for all cases, however with OpenMP, copy_if tends to be perform worse when the number of elements being passed in the output are less.

To ensure that the stream compaction implementation within the Dax framework performs well across all devices with a arbitrary number of elements in the output array we use the copy_if implementation as the default in Dax, while providing the ability for the user to choose to use the prefix scan implementation if needed. We are investigating using some heuristic based approach to automatically determine the optimal implementation to use for stream compaction.

5. CONCLUSION

Exascale computing requires that we migrate current serial and parallel analysis algorithms to work at a much finer granularity. Dax toolkit is designed to encapsulate the complexity of fine grained parallelism within the framework, making it easier for developers to write algorithms that are inherently parallel. In this paper, we describe our experience with implementing an algorithm for thresholding of cells in Dax. By incorporating subtle changes in the API and scaffolding, we



(a) Serial (b) OpenMP (c) CUDA

Figure 5: Comparison of the *prefix scan* and *copy_if* based approaches for stream compaction on a 512^3 array with 1.5%, 10% and 100% of the elements marked as valid and thus expected in the resulting array as see on various device implementations. Figure 5a shows the results of the serial CPU implementation using Intel Xeon X5680. Figure 5b shows the results of the parallel CPU implementation using OpenMP on dual Intel Xeon X5680 with a total of 12 cores. Figure 5c shows the results of the parallel GPU implementation using CUDA on Tesla C2050.

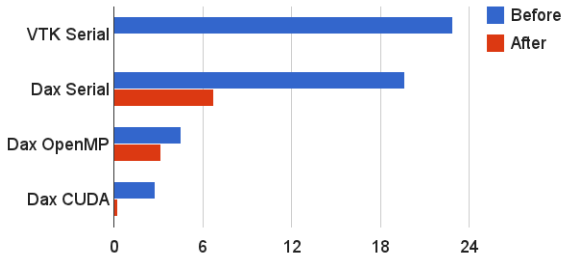


Figure 6: Comparison of performance of threshold as implemented in Dax before and after the improvements discussed on dual Intel Xeon X5680 with 12 cores and NVIDIA Tesla C2050.

were able to achieve better performance as shown in Figure 6. Since most of these design changes are incorporated within the framework itself, rather than optimizing the individual algorithm, we can leverage these performance improvements for other topology modifying algorithms such as tetrahedralizing voxels, and extracting surfaces, features, and edges.

ACKNOWLEDGMENTS

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

Part of this work was performed by Sandia National Laboratories. 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.

REFERENCES

[1] Dongarra, J., Beechman, P., et al., "The international exascale software project roadmap," Tech. Rep. ut-cs-10-652, University of Tennessee (January 2010).

[2] Richards, M. et al., "Exascale software study: Software challenges in extreme scale systems," tech. rep., DARPA Information Processing Techniques Office (IPTO) (September 2009).

[3] Johnson, C. and Ross, R., "Visualization and knowledge discovery: Report from the DOE/ASCR workshop on visual analysis and data exploration at extreme scale," tech. rep. (October 2007).

[4] Moreland, K., Ayachit, U., Geveci, B., and Ma, K.-L., "Dax toolkit: A proposed framework for data analysis and visualization at extreme scale," *LDAV 2011 IEEE Symposium on Large-Scale Data Analysis and Visualization* (November 2011).

[5] NVIDIA, *NVIDIA CUDA Programming Guide, Version 2.3.1* (August 2009).

[6] Chapman, B., Jost, G., and van der Pas, R., [*Using OpenMP*], MIT Press (2007). ISBN-13 978-0-262-53302-7.

[7] Hoberock, J. and Bell, N., "Thrust: A parallel template library," (2010). Version 1.4.0.

[8] Reinders, J., [*Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*], O'Reilly (2007). ISBN-13 978-0-596-51480-8.

[9] Horn, D., "Stream reduction operations for gpgpu applications," in [*GPU Gems 2*], Pharr, M., ed., 573–589, Addison-Wesley (2005).

[10] Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D., "Scan primitives for gpu computing," in [*Graphics Hardware 2007*], 97–106, ACM (Aug. 2007).

[11] Lo, L.-T., Sewell, C., and Ahrens, J., "Piston: A portable cross-platform framework for data-parallel visualization operators," (2012).

[12] Schroeder, W., Martin, K., and Lorensen, B., [*The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*], Kitware Inc., fourth ed. (2004). ISBN 1-930934-19-X.

[13] Solano-Quinde, L., Wang, Z. J., Bode, B., and Somani, A. K., "Unstructured grid applications on gpu: performance analysis and improvement," in [*Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*], *GPGPU-4*, 13:1–13:8, ACM, New York, NY, USA (2011).

[14] Dyken, C., Ziegler, G., Theobalt, C., and Seidel, H.-P., "High-speed marching cubes using histopyramids," *Computer Graphics Forum* **27**, 2028–2039 (September 2008).

[15] Harris, M., Sengupta, S., and Owens, J. D., "Parallel prefix sum (scan) with CUDA," in [*GPU Gems 3*], Nguyen, H., ed., ch. 39, 851–876, Addison Wesley (August 2007).