SAND2011-2842C

A Proposed Framework for Data Analysis and Visualization at Extreme Scale

Utkarsh Ayachit* Kitware, Inc.
Kenneth Moreland† Sandia National Laboratories
Berk Geveci‡ Kitware, Inc.
Kwan-Liu Ma§ University of California at Davis

ABSTRACT

Experts agree that the exascale machine will comprise processors that contain many cores, which in turn will necessitate a much higher degree of concurrency. Software will require a minimum of a 1000 times more concurrency. Most parallel analysis and visualization algorithms today work by partitioning data and running mostly serial algorithms concurrently on each data partition. Although this approach lends itself well to the concurrency of current high performance computing, it does not exhibit the appropriate pervasive parallelism required for exascale computing. The data partitions are too small and the overhead of the threads too large to make effective use of all the cores in an extreme scale machine. This paper introduces a new visualization framework designed to exhibit the pervasive parallelism necessary for extreme scale machines. We demonstrate the use of this system on a GPU processor, which we feel is the best analog to an exascale node that we have available today.

Index Terms: D.1.3 [Software]: Programming Techniques—Concurrent Programming

1 INTRODUCTION

Most of today's visualization libraries and applications are based off of what is known as the *visualization pipeline* [19, 27]. The visualization pipeline is the key metaphor in many visualization development systems such as the Visualization Toolkit (VTK) [45], SCIRun [33], the Application Visualization System (AVS) [49], OpenDX [1], and Iris Explorer [16]. It is also the internal mechanism or external interface for many end-user visualization applications such as ParaView [46], VisIt [26], VisTrails [6], MayaVi [40], VolView [24], OsiriX [44], 3D Slicer [38], and BioImageXD [23].

In the visualization pipeline model, algorithms are encapsulated as *filter* components with inputs and outputs. These filters can be combined by connecting the outputs of one filter to the inputs of another filter. The visualization pipeline model is popular because it provides a convenient abstraction that allows users to combine algorithms in powerful ways.

Although the visualization pipeline lends itself well to the concurrency of current high performance computing [29, 35, 39, 51], its structure prohibits the necessary extreme concurrency required for exascale computers. This paper describes the design of the *Dax toolkit* to perform Data Analysis at Extreme scales. The computational unit of this framework is a *worklet*, a single operation on a small piece of data. Worklets can be combined in much the same way as filters, but their light weight, lack of state, and small data access make them more suitable for the massive concurrency required by exascale computers and associated multi- and many-core processors. Figure 1 highlights the difference between a Dax pipeline execution with a traditional visualization pipeline provided by existing visualization frameworks such as VTK.

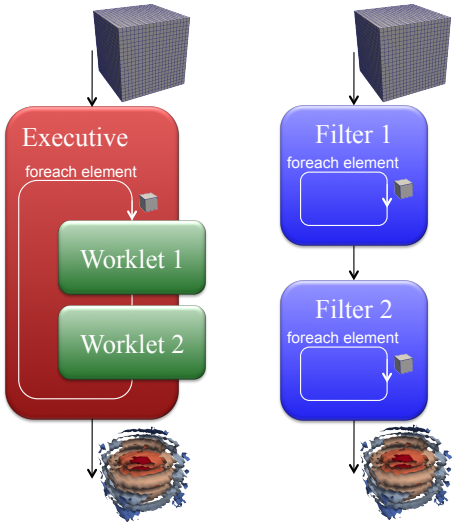

Figure 1: Comparison between Dax pipeline execution and traditional visualization pipeline execution. Dax makes it possible to achieve higher degree of concurrency by moving the iteration over mesh elements out of the algorithm and letting the framework manage the parallelism.

2 RELATED WORK

Since its inception, many improvements have been made to the visualization pipeline to allow it to function well with large data. By dividing tasks, pipelines, or data amongst processes, the serial algorithms of a visualization pipeline can work in parallel with little or no modification [2]. In particular, the data parallel mode, partitioning the data and replicating the pipeline, performs well on current high performance computers [10].

The basic function of a visualization pipeline is to process data flowing from upstream to downstream. More recent visualization pipeline implementations, such as those in VTK, ParaView, and VisIt, implement more advanced control mechanisms and pass meta-data throughout the pipeline. These control mechanisms can, for example, subset the data in space [13] or time [7] based on the needs of the individual computing units. Recent work is coupling this mechanism with query-driven visualization techniques [17] to better sift through large data sets with limited resources.

These control mechanisms can also be used to stream data, in pieces, through the pipeline [3]. More recent advances allow us to prioritize the streaming, thus allowing to compensate for high latency of streaming by presenting the most relevant data first [4]. This in turn has lead to multi-resolution visualization [34,52]. Mul-

*email: utkarsh.ayachit@kitware.com
†email: kmorel@sandia.gov
‡email: berk.geveci@kitware.com
§email: ma@cs.ucdavis.edu

tiple resolutions further hide limited resource latency by first presenting low-detailed results and then iteratively refining them as needed. This assumes, of course, that a multi-resolution hierarchy of data is already built (a non-trivial task for unstructured data). This work is being implemented in a traditional visualization pipeline, but could be leveraged in many other types of frameworks, including the one proposed for this project.

Another recent research project extends the visualization-pipeline streaming mechanism by automatically orchestrating task concurrency in independent components of the pipeline [50]. The technique adapts the visualization pipeline to multi-core processors, but it has its limitations. There is a high overhead with regard to each execution thread created; they require isolated buffers of memory for input and output and independent call stacks, which typically run many calls deep. Furthermore, algorithms in the filters are optimized to iterate over sizable data chunks, which will not be the case with massive multi-threading. At some point the algorithms will have to be reengineered to process small data chunks or themselves be multithreaded. It will be necessary to leverage a threading paradigm like the one proposed for this project to engineer this kind of change on a full-featured toolkit.

An alternative data analysis and visualization architecture is implemented by the Field Encapsulation Library (FEL) [8]. FEL provides abstractions that allows programs to access the structure and fields of a mesh independently from the data storage. More importantly, FEL uses C++ template constructs to build functional definitions of fields. These fields compute values on demand when requested. These functional fields are similar in nature to the worklets defined in our work.

Although the main concerns addressed by FEL, mesh flexibility and memory overhead, is complementary to this project, FEL does not adequately manage the complexity of massive multi-threading. To support pervasive parallelism we need to hide the complexity of work distribution. Also, as the name implies, the Field Encapsulation Library is primarily concerned with defining, accessing, and operating on fields. There is no mechanism for topological operations that change or create meshes. Nor is there any explicit method for aggregation. In order to address the varied data analysis and visualization needs, this project enables these features.

Another system with constructs similar to the framework proposed for this project is provided by Intel Threading Building Blocks (TBB) [41], a popular open-source C++ library for multi-core programming. In addition to high-level parallel constructs such as parallel looping and reduction operations, TBB also provides a simple pipeline execution environment. Like Dax, TBB's pipeline mechanism partitions data based on available hardware threads, handing-off the resulting partitions to caller-supplied functions that each iterate over their assigned ranges to perform computation. Thus, TBB provides a hybrid abstraction where callers are isolated from some of the complexity of scheduling work across multiple cores, but each function is still responsible for iteration over its subset of the data.

This approach is appropriate for current architectures where an individual host has a relatively small number of hardware threads, but requires that function authors continue to deal with data organization and iteration issues on an ad-hoc basis. Further, TBB does not address issues of scheduling or communication across multiple hosts in a distributed platform. Dax envisions a stricter separation of responsibilities where worklets are responsible for computation only, leaving data retrieval and inter-processor communication to separate executive components.

The MapReduce programming model [14] is also similar in spirit to our proposed framework. Like our approach, MapReduce simplifies parallel programming by defining algorithms in terms of local, stateless operations. However MapReduce, not being designed as such, does not have all the conventions necessary for a fully featured visualization and data analysis library. For example, expressing local topological connections (e.g. cells connected to vertices, vertices connected to cells, or cells connected to cells) are difficult to express. The combining of predefined computation units is not directly supported, nor is the specification of topological, spatial, or temporal domains. In contrast, our proposed system provides the primitives with which visualization and data analysis programmers are accustomed.

3 Motivation

As the scale of supercomputers has progressed from the teraflop to the petaflop we have enjoyed a resiliency of the message passing model (embodied in the use of MPI) as an effective means of attaining scalability. However, as we consider the high performance computer of the future, the exascale machine, we discover that this concurrency model will no longer be sufficient. All industry trends infer that the exascale machine will be built using many-core processors containing hundreds to thousands of cores per chip. This change in processor design has dramatic effects on the design of large-scale parallel programs. As stated by a recent study by the DARPA Information Processing Techniques Office [42]:

> The concurrency challenge is manifest in the need for software to expose at least $1000\times$ more concurrency in applications for Extreme Scale systems, relative to current systems. It is further exacerbated by the projected memory-computation imbalances in Extreme Scale systems, with Bytes/Ops ratios that may drop to values as low as 10^{-2} where Bytes and Ops represent the main memory and computation capacities of the system respectively. These ratios will result in $100\times$ reductions in memory per core relative to Petascale systems, with accompanying reductions in memory bandwidth per core. Thus, a significant fraction of software concurrency in Extreme Scale systems must come from exploiting more parallelism within the computation performed on a single datum.

Put simply, efficient concurrency on exascale machines requires a massive amount of concurrent threads, each performing many operations on a small and localized piece of data.

Other studies concur. The Workshop on Visual Analysis and Data Exploration at Extreme Scale [22] corroborates the need for "pervasive parallelism" throughout visual analysis tools and that data access is a prime consideration for future tools. The International Exascale Software Project's recent road map [15] also states a required thousand fold increase in concurrency and that applications may require ten billion threads. The road map also notes a change in I/O and Memory that will "affect programming models and optimization." Careful consideration of memory access is also expected to have a dramatic effect on energy consumption as much of the power of an exascale system will be expended moving data.

Will visualization systems need to run on these exascale systems? They undoubtedly will. Although it has been a common practice to use specialty high performance platforms for visualization and graphics [53], this trend is coming to an end. The cost of creating specialty visualization computers that are capable of analyzing data generated from large supercomputer runs is becoming prohibitive [12]. Consequently, researchers are beginning to leverage the same supercomputers used for creating the data [36, 37, 54]. This, coupled with a renewed interest in running visualization *in-situ* with simulations to overcome file I/O constraints [43, 47, 48], ensures that high performance visualization code will run on the same technology as the simulation code for the foreseeable future.

Visualization pipelines fit poorly into this massive concurrency model; the granularity of the pipeline computational unit, the filter, is too large. Each filter must ingest, process, and produce an entire

data set when invoked. Large scale concurrency today is achieved by replicating the pipeline and partitioning the input data amongst processes [2]. However, extreme scale computers would require the data to be broken into billions of partitions. The overhead of capturing the connectivity information between these partitions as well as the overhead of executing these large computation units on such small partitions of data is too great to make such an approach practical.

To understand why, consider the sobering comparison between the Jaguar XT5 partition, a current petascale machine, and the projections for an exascale machine of by the International Exascale Software Project RoadMap [15] given in Table 1. Because processor clock rates are not increasing, an exascale computer requires a thousand-fold increase in the number of cores. Furthermore, trends in processor design suggest that these cores must be hyper-threaded in order to keep them executing at full efficiency. In all, to drive a complete exascale machine will require between one and ten billion concurrently running threads.

Table 1: Comparison of characteristics between petascale and projected exascale machines.

	Jaguar – XT5	Exascale	Increase
Cores	224,256	100 million – 1 billion	$\sim 1,000\times$
Threads	224,256 way	1 – 10 billion way	$\sim 50,000\times$
Memory	300 Terabytes	128 Petabytes	$\sim 500\times$

Most of our current tools rely on MPI for concurrency. An MPI process has the overhead of a running program with its own memory space. A common process has an overhead of about twenty megabytes. Running on the entirety of Jaguar yields an overhead of about 4 terabytes, less than two percent of the overall available memory. In contrast, the overhead for using MPI processes for all the concurrency on an exascale machine requires up to 200 petabytes, possibly exceeding the total memory on the system in overhead alone.

Even getting around problems with the overhead of MPI, the visualization pipeline still has inherent problems at this level of concurrency. Consider using Jaguar to process a one trillion cell mesh. If we partition these cells evenly amongst all the cores where replicated pipelines will process each partition, that yields roughly 5 million cells per pipeline. General rules of thumb indicate this ratio is optimal for structured grids when running parallel VTK pipelines [28]. Scaling to an exascale machine, we can project to processing 500 trillion cells (considering this is the expected growth in system memory). If we partition these cells evenly amongst all the necessary cores where replicated pipelines will process each partition, that yields as few as 50 thousand cells per pipeline. Here we are starving our pipeline.

Even if we somehow avoid the problem of running on the largest exascale machines, the problem of a fundamental change in processor architecture persists. The parallel visualization pipeline simply does not conform well to multi-core processors and many-core accelerators. In response several researchers are pursing the idea of a *hybrid parallel pipeline* [9, 11, 20]. The hybrid parallel pipeline breaks the problem into two hierarchical levels. The first level partitions the data amongst distributed memory nodes in the same way as the current parallel pipeline. In the second level we run a threaded shared memory algorithm to take advantage of a multi- or many-core processor.

Although the current visualization pipeline does a good job in providing this first level of distributed memory concurrency, it provides no facilities whatsoever for this second layer of multi-threaded concurrency. This places the onus on each visualization pipeline filter developer. That is, each filter must be independently and painstakingly designed to exploit concurrency and optimized for whatever architecture is used. Even if this undertaking were to be performed, the concurrency is ultimately undermined at the connections of filters where execution threads must be synchronized and data combined.

Our Dax toolkit is designed to encapsulate the complexity of multi-threaded visualization and data analysis algorithms. Our initial implementation targets GPU architectures. We feel that the idiosyncrasies of these accelerators, many threads with explicit memory locality, are representative of all future architectures.

4 System Overview

According to the ExaScale Software Study performed by DARPA IPTO [42], "it is important to ensure that the intrinsic parallelism in a program can be expressed at the finest level possible *e.g.*, at the statement or expression level, and that the compiler and runtime system can then exploit the subset of parallelism that is useful for a given target machine." Taking this advice to heart, we propose building a visualization framework using the worklet as the basic computational unit. A worklet is an algorithm broken down to its finest level. It operates on one datum and, when possible, generates one datum. A worklet has no state; it operates only on the data it is given.

Reducing visualization algorithms to this fine of a computational unit is feasible because of the embarrassingly parallel nature of most visualization algorithms. We are exploiting the same algorithm properties that make the streaming and data parallelism approaches feasible [2, 3]. In essence, the operations of most visualization algorithms involve data at a single location in the mesh and its immediate neighborhood. Hence, we can break the data down to the elemental pieces of the mesh.

In this section we describe different components of the Dax framework. In our current implementation we use the GPU processor as an analog for the exascale node. We use OpenCL [30] to compile and execute worklets on the GPU, however it must be noted that all the user-developed worklet code is a subset of Standard C [5] and is independent of any GPGPU/OpenCL constructs, thus making it possible to port the worklets to different computing languages based on Standard C including CUDA [31].

With the analysis algorithms implemented as worklets, the framework provides mechanisms to connect these worklets to form visualization pipelines. Since we decided to use GPUs as the analog to an exascale node available to us today, the framework also manages data movement and scheduling of the worklets for executing on the GPU. Also, as is the case with any full-fledged visualization framework, we provide a data model. The data model essentially helps us define the data-structures used to store data in memory for the system and semantics associated with them.

Dax toolkit provides two programming environments: one to develop new worklets and one to use the Dax system.

Execution Environment This is the API exposed to developers that write worklets for different visualization algorithms. This API provides work for one datum with convenient access to information such as connectivity and neighborhood needed by typical visualization algorithms. This is a Standard C API that makes it possible to compile the worklets for existing GPU devices using OpenCL.

Control Environment This is an API that is used on a node in an exascale machine to build the visualization pipeline, transfer data to and from IO devices, and schedule the parallel execution on the worklets. It is a C++ API that is designed for users that want to use the Dax toolkit to analyze and visualize their data using provided or supplied worklets.

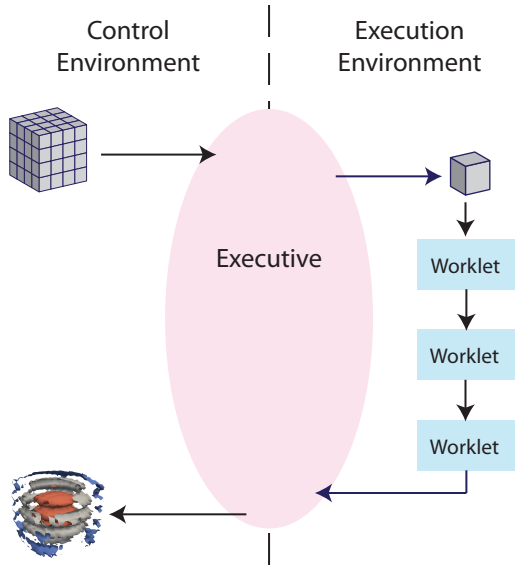

Figure 2: Layout of the Dax system. Applications using Dax have a different programming and execution environment than the worklets. The executive organizes the interaction between these two environments.

The dual programming environments is partially a convenience to isolate the application from the execution of the worklets and is partially a necessity to support GPU languages with host and device environments. The Dax toolkit provides an object called an *Executive* that acts as an interface between the control and execution environment. The executive accepts mesh data and execution requests from the application running in the control environment. Based on these requests, the executive builds worklet pipelines, manages memory, and schedules threads in the execution environment. The relationship between the control and execution environments and the executive's role in managing them is demonstrated in Figure 2.

4.1 Data Model

The data model used by the Dax framework is loosely based on the OpenDX Data Model [21], which itself is based on the data model proposed by Haber et al. [18]. This data model makes it possible to describe data attributes defined on different kinds of grids including uniform rectilinear grids, curvilinear grids, and unstructured grids made up of triangles, quads, tetrahedra, etc. Similar to OpenDX, the bulk of the data is encapsulated in array objects called *daxArray*. Also, there are different types of arrays such as regular array, constant array, and irregular array that make it possible to compactly represent values in the array. A dataset comprises a named collection of arrays with relationships between the arrays. There are two possible relationships between arrays:

dep relationship records a dependency. Array A *dep*ends on array B implies that A has exactly as many items as B and every item in A corresponds to an item in B.

ref relationship records a reference or indirection. Array A *ref*erences array B implies that values in A are references to values in B e.g. an array describing the cell connections between points references the "positions" array that defines the point coordinates.

Figure 3 shows a representation for a dataset with cell and point attributes.

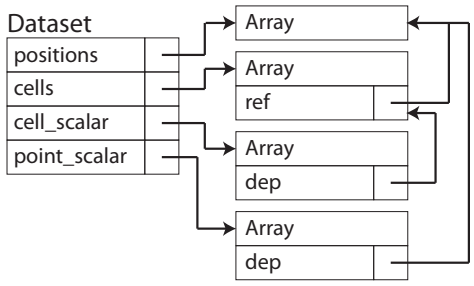

Figure 3: Example of a dataset with a point and cell scalars.

4.2 Execution Model

The execution model is based on the data-flow paradigm. In a data-flow implementation, all nodes in the data-flow pipeline are pure stateless functions through which the data flows. The Dax execution model is similar. It comprises Modules (daxModule) that are connected together to form pipelines. The crux of the module (i.e. the algorithm or the processing logic) is the worklet. A worklet is C-function that takes processes input array(s) to generate output array(s). The module can be thought of as the wrapper around the worklet that facilitates hooking up of these worklets to form pipelines as well as provide support for type-checking and kernel generation. The Executive is an object that builds the data-flow pipeline and schedules its execution on the device. The complexity in the Executive stems from ensuring that the worklets are executed in correct orders on the device.

For our implementation, we are focusing on OpenCL. Thus the worklet is written as an OpenCL function using dAPI (Device API) for accessing data (described later). The executive, accessed via OpenCL's host API, generates an OpenCL kernel to schedules the kernel for execution to produce the requested result.

4.3 Execution Environment

The worklet code uses this API to access the data, compute values, and then produce the result. A worklet is a simply a C-function that takes in input array(s) and produces an output array. The execution environment adds annotations to the arguments making it possible to deduce relationships between the arguments. For example, the following code shows the prototype for a worklet that takes in a position coordinate and point scalar to produce a new point scalar.

```
__worklet__ void PointWorklet(daxWork work,
  const daxArray* __positions__ in_positions,
  const daxArray* __dep__(__positions__) in_point_scalars,
  daxArray* __dep__(__positions__) out_scalars)
  {
  daxFloat3 point_cordinate =
    daxGetArrayValue3(work, in_positions);
  daxFloat in_value =
    daxGetArrayValue(work, in_point_scalar);
  daxFloat out_value = ...;
  daxSetArrayValue(work, out_scalars, out_value);
  }
```

Figure 4: Pseudo-code for a worklet operating on input point scalars and point coordinates to generate point scalars.

As is clear from the above code snippet, the worklet code never directly access any memory locations. It uses the API to get and set values from opaque types. Also, the datum the operations are being performed on are identified by an opaque handle called daxWork. daxWork makes it possible for the framework to optimize reads and writes to and from global memory.

The following worklet code computes cell-gradients. It demonstrates iterating over all the points in a cell and computing a derived quantity, which is typical of many visualization algorithms.

```
__worklet__ void CellGradient(const daxWork work,
  const daxArray* __positions__ in_positions,
  const daxArray*
    __and__(__connections__, __ref__(in_positions))
    in_connections,
  const daxArray* __dep__(in_positions) inputArray,
  daxArray* __dep__(in_connections) outputArray)
{
  daxConnectedComponent cell;
  daxGetConnectedComponent(work, in_connections, &cell);

  daxFloat scalars[MAX_CELL_POINTS];
  uint num_elements = daxGetNumberOfElements(&cell);
  daxWork point_work;
  for (uint cc=0; cc < num_elements; cc++)
    {
    point_work = daxGetWorkForElement(&cell, cc);
    scalars[cc] = daxGetArrayValue(point_work, inputArray);
    }

  daxFloat3 parametric_cell_center =
    (daxFloat3)(0.5, 0.5, 0.5);
  daxFloat3 gradient = daxGetCellDerivative(&cell,
    0, parametric_cell_center, scalars);
  daxSetArrayValue3(work, outputArray, gradient);
}
```

Figure 5: Worklet for computing cell gradients in Dax Execution environment.

Since the worklet code never directly accesses memory, the framework is free to optimize the fetches and write-backs to global memory under the covers, avoiding global memory writes all together for intermediate results in the pipeline. Furthermore, the design isolates the worklet developers from changes to the underlying platform. For example, by simply providing CUDA based implementations for all execution environment and updating the executive to use CUDA runtime components, we can port the entire framework to a CUDA platform instead of OpenCL.

The annotations used in the arguments to the worklets serve two purposes. First, it enables the worklet developer to make certain assumptions about the arrays. For example, if an argument is marked as *__positions__*, the array represents point coordinates with exactly 3 components. Second, it enables the control environment (as explained the section 4.4) to validate the pipeline, catching any invalid connections before the job is dispatched to the parallel platform.

4.4 Control Environment

The control environment can be considered as the scaffolding interface that helps set up the visualization pipeline. Each worklet gets wrapped into a module (daxModule), with each array argument to the worklet becoming an input or an output port for the module. A pipeline is constructed by connecting the output ports to input ports on different modules. To execute the pipeline, one calls Executive::Execute(). That results in first validating the pipeline to ensure that input port requirements are met. Second, an OpenCL kernel is generated that executes the entire pipeline on a datum. Finally, the data arrays are uploaded to the device memory and the OpenCL kernel is triggered.

The execution of the worklets in the pipeline is demand driven. That is, it is triggered by calling the last worklet in the pipeline. The framework maintains the information of how each intermediate array is to be generated. When a worklet makes a daxGetArrayValue$*$ function call, it checks if the value has been computed or available in global memory. If not, it executes the source-worklet to produce the value implicitly. Likewise, daxSetArrayValue$*$ function calls don't necessarily result in a global memory write. If the array being written to an intermediate result i.e. result being passed from one worklet to another, then the value gets written in local memory and returned by the daxGetArrayValue$*$ call that triggered the execution of the worklet. All this is happening under the covers without the worklet having to worry about any of these optimizations.

Our current implementations don't store any intermediate values. Consequently if a worklet access the same input array location twice, then the producer worklet that generates that value is also executed twice. However, based on the platform, we can easily support sharing of intermediate results between worklet groups (called workgroups in OpenCL).

5 Results

Table 2 compares the execution times for a simple pipeline Elevation → Cell Gradient applied to 256x256x256 block of 3D uniform rectilinear grid using NVIDIA GeForce 8800 GTX graphics card against a serial VTK-based implementation of the same pipeline on a Intel Xeon 3.00 GHz CPU.

Table 2: Performance comparison between Dax toolkit and VTK

Computation	VTK	Dax	Speedup
Elevation → Gradient, 256^3	15.61 s	0.72 s	21.7

We see that even with a simple pipeline involving just two worklets, we can get a decent speed up.

The code that developer writes in the worklet is comparable to the code one would write to do the something similar in existing visualization frameworks such as VTK. Figure 6 shows a code snippet for computing cell gradients in VTK's vtkCellDerivatives filter.

```
int vtkCellDerivatives::RequestData(...)
{
  ...[allocate output arrays]...
  ...[validate inputs]...
  for (cellId=0; cellId < numCells; cellId++)
    {
    ...
    input->GetCell(cellId, cell);
    subId = cell->GetParametricCenter(pcoords);
    inScalars->GetTuples(cell->PointIds, cellScalars);
    scalars = cellScalars->GetPointer(0);
    cell->Derivatives(subId, pcoords, scalars, 1, derivs);
    outGradients->SetTuple(cellId, derivs);
    }
  ...[cleanup]...
}
```

Figure 6: Code to compute cell gradients in VTK (vtkCellDerivatives). Compare this with code for the same in Dax Toolkit (Figure 5).

6 Challenges Ahead

In this paper we outlined our proposed framework and demonstrated the feasibility using worklets that compute point scalars or generate derived cell quantities using cell geometry and topology. These cover a wide range of visualization and analysis algorithms however there still remain a set of algorithms such as clipping cells, iso-surfacing that remain to be addressed. In general, algorithms that change the topology or connectivity have not been discussed. The unique characteristic of such algorithms is that they cannot determine the number of elements a priori. Unlike traditional filters,

a worklet has no explicit memory management or control capabilities. Using the information provided by the annotations, the executive deduces the memory requirements and allocates appropriate buffers. Since that's no longer possible for topology changing algorithms, it becomes essential that such worklets are split into at least two components: one computing the number of elements being generated and the second doing the actual work to generate the new elements. The executive will then have to execute the two components in separate passes. This multipass approach is employed by existing implementations of marching cubes algorithm on the GPU using CUDA [32].

Another challenge is interworklet communication. Certain visualization algorithms are not amenable to being parallelized without extensive communication e.g. streamline generation. Although theoretically it is possible for worklets to communicate with each other using explicit synchronization mechanisms it can affect the performance drastically. As framework designers, we either have to bite the bullet and support such algorithms or provide alternatives. For example, although streamline tracing is not easily parallelizable, line integral convolution [25] can be suggested as parallel alternative for analyzing vector fields.

ACKNOWLEDGEMENTS

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] G. Abram and L. A. Treinish. An extended data-flow architecture for data analysis and visualization. Technical Report RC 20001 (88338), IBM Thomas J. Watson Research Center, 1995.

[2] J. Ahrens, C. Law, W. Schroeder, K. Martin, and M. Papka. A parallel approach for efficiently visualizing extremely large, time-varying datasets. Technical Report #LAUR-00-1620, Los Alamos National Laboratory, 2000.

[3] J. P. Ahrens, K. Brislawn, K. Martin, B. Geveci, C. C. Law, and M. E. Papka. Large-scale data visualization using parallel data streaming. *IEEE Computer Graphics and Applications*, 21(4):34–41, July/August 2001.

[4] J. P. Ahrens, N. Desai, P. S. McCormic, K. Martin, and J. Woodring. A modular, extensible visualization system architecture for culled, prioritized data streaming. In *Visualization and Data Analysis 2007*, pages 64950I:1–12, 2007.

[5] American National Standard for Information Systems. Programming Language C ANSI X3.159–1989. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, 1990.

[6] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo. VisTrails: Enabling interactive multiple-view visualizations. In *Proceedings of IEEE Visualization*, October 2005.

[7] J. Biddiscombe, B. Geveci, K. Martin, K. Moreland, and D. Thompson. Time dependent processing in a parallel pipeline architecture. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1376–1383, November/December 2007. DOI=10.1109/TVCG.2007.70600.

[8] S. Bryson, D. Kenwright, and M. Gerald-Yamasaki. FEL: The field encapsulation library. In *Proceedings Visualization '96*, pages 241–247, October 1996.

[9] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. Joy. Streamline integration using mpi-hybrid parallelism on large multi-core architecture. *IEEE Transactions on Visualization and Computer Graphics*, December 2010. DOI=10.1109/TVCG.2010.259.

[10] A. Cedilnik, B. Geveci, K. Moreland, J. Ahrens, and J. Favre. Remote large data visualization in the paraview framework. In *Eurographics Parallel Graphics and Visualization 2006*, pages 163–170, May 2006.

[11] L. Chen and I. Fujishiro. Optimization strategies using hybrid MPI+OpenMP parallelization for large-scale data visualization on earth simulator. In *A Practical Programming Model for the Multi-Core Era*, volume 4935, pages 112–124. 2008. DOI=10.1007/978-3-540-69303-1_10.

[12] H. Childs. Architectural challenges and solutions for petascale post-processing. *Journal of Physics: Conference Series*, 78(012012), 2007. DOI=10.1088/1742-6596/78/1/012012.

[13] H. Childs, E. Brugger, K. Bonnell, J. Meredith, M. Miller, B. Whitlock, and N. Max. A contract based system for large data visualization. In *IEEE Visualization 2005*, pages 191–198, 2005.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[15] J. Dongarra, P. Beechman, et al. The international exascale software project roadmap. Technical Report ut-cs-10-652, University of Tennessee, January 2010.

[16] D. Foulser. IRIS Explorer: A framework for investigation. *ACM SIGGRAPH Computer Graphics*, 29(2):13–16, May 1995.

[17] L. J. Gosink, J. C. Anderson, E. W. Bethel, and K. I. Joy. Query-driven visualization of time-varying adaptive mesh refinement data. *IEEE Transactions of Visualization and Computer Graphics*, 14(6):1715–1722, November/December 2008.

[18] R. B. Haber, B. Lucas, and N. Collins. A data model for scientific visualization with provisions for regular and irregular grids. In *Proceedings of the 2nd conference on Visualization '91*, pages 298–305, 1991.

[19] P. E. Haeberli. ConMan: A visual programming language for interactive graphics. *ACM SIGGRAPH Computer Graphics*, 22(4):103–111, August 1988.

[20] M. Howison, E. W. Bethel, and H. Childs. Hybrid parallelism for volume rendering on large, multi- and many-core systems. *IEEE Transactions on Visualization and Computer Graphics*, January 2011. DOI=10.1109/TVCG.2011.24.

[21] IBM Corporation. *IBM Visualization Data Explorer User's Guide*, seventh edition, May 1997.

[22] C. Johnson and R. Ross. Visualization and knowledge discovery: Report from the DOE/ASCR workshop on visual analysis and data exploration at extreme scale. Technical report, October 2007.

[23] P. Kankaanpää, K. Pahajoki, V. Marjomäki, D. White, and J. Heino. BioImageXD - free microscopy image processing software. *Microscopy and Microanalysis*, 14(Supplement 2):724–725, August 2008.

[24] Kitware, Inc. VolView 3.2 user manual, June 2009.

[25] R. S. Laramee, B. Jobard, and H. Hauser. Image space based visualization of unsteady flow on surfaces. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 18–, Washington, DC, USA, 2003. IEEE Computer Society.

[26] Lawrence Livermore National Laboratory. *VisIt User's Manual*, October 2005. Technical Report UCRL-SM-220449.

[27] B. Lucas, G. D. Abram, N. S. Collins, D. A. Epstein, D. L. Gresh, and K. P. McAuliffe. An architecture for scientific visualization system. In *IEEE Visualization*, pages 107–114, 1992.

[28] K. Moreland. The paraview tutorial, version 3.8. Technical Report SAND 2009-6039 P, Sandia National Laboratories, 2010.

[29] K. Moreland, D. Rogers, J. Greenfield, B. Geveci, P. Marion, A. Neundorf, and K. Eschenberg. Large scale visualization on the Cray XT3 using paraview. In *Cray User Group*, 2008.

[30] A. Munshi. *The OpenCL Specification*. Khronos OpenCL Working Group, September 2010. Version 1.1, Revision 36.

[31] NVIDIA. *NVIDIA CUDA Programming Guide, Version 2.3.1*, August 2009.

[32] NVIDIA Corporation. Marching Cubes Isosurfaces. `http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html#marchingCubes`.

[33] S. G. Parker and C. R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Proceedings ACM/IEEE*

Conference on Supercomputing, 1995.

[34] V. Pascucci and R. J. Frank. Global static indexing for real-time exploration of very large regular grids. In *Proceedings of ACM/IEEE Conference on Supercomputing*, November 2001.

[35] J. Patchett, J. Ahrens, S. Ahern, and D. Pugmire. Parallel visualization and analysis with ParaView on a Cray Xt4. In *Cray User Group*, 2009.

[36] T. Peterka, D. Goodell, R. B. Ross, H.-W. Shen, and R. Thakur. A configurable algorithm for parallel image-compositing applications. In *Proceedings of ACM/IEEE Conference on Supercomputing*, November 2009.

[37] T. Peterka, R. B. Ross, H.-W. Shen, K.-L. Ma, W. Kendall, and H. Yu. Parallel visualization on leadership computing resources. In *Journal of Physics: Conference Series SciDAC 2009*, June 2009.

[38] S. Pieper, M. Halle, and R. Kikinis. 3D slicer. In *Proceedings of the 1st IEEE International Symposium on Biomedical Imaging: From Nano to Macro 2004*, pages 632–635, April 2004.

[39] D. Pugmire, H. Childs, and S. Ahern. Parallel analysis and visualization on cray compute node linux. In *Cray User Group*, 2008.

[40] P. Ramachandran. MayaVi: A free tool for CFD data visualization. In *4th Annual CFD Symposium*, August 2001.

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

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

[43] R. B. Ross, T. Peterka, H.-W. Shen, Y. Hong, K.-L. Ma, H. Yu, and K. Moreland. Visualization and parallel I/O at extreme scale. *Journal of Physics: Conference Series*, 125(012099), 2008. DOI=10.1088/1742-6596/125/1/012099.

[44] A. Rosset, L. Spadola, and O. Ratib. OsiriX: An open-source software for navigating in multidimensional dicom images. *Journal of Digital Imaging*, 17(3):205–216, September 2004.

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

[46] A. H. Squillacote. *The ParaView Guide: A Parallel Visualization Application*. Kitware Inc., 2007. ISBN 1-930934-21-1.

[47] D. Thompson, N. D. Fabian, K. D. Moreland, and L. G. Ice. Design issues for performing *in situ* analysis of simulation data. Technical Report SAND2009-2014, Sandia National Laboratories, 2009.

[48] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K.-L. Ma, and D. R. O'Hallaron. From mesh generation to scientific visualization: An end-to-end approach to parallel supercomputing. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[49] C. Upson, T. F. Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, July 1989.

[50] H. T. Vo and C. T. Silva. Multi-threaded streaming pipeline for VTK. Technical Report UUSCI-2009-005, SCI Institute, University of Utah, 2009.

[51] D. White. Red storm capability visualization level II ASC milestone #1313 final report. Technical Report SAND2005-5989P, Sandia National Laboratories, 2005.

[52] J. Woodring. Interactive remote large-scale data visualization via prioritized multi-resolution streaming. In *2009 Ultrascale Visualization Workshop*, November 2009.

[53] B. Wylie, C. Pavlakos, and K. Moreland. Scalable rendering on PC clusters. *IEEE Computer Graphics and Applications*, 21(4):62–70, July/August 2001.

[54] H. Yu, C. Wang, and K.-L. Ma. Parallel volume rendering using 2-3 swap image compositing for an arbitrary number of processors. In *Proceedings of ACM/IEEE Conference on Supercomputing*, November 2008.