Lawrence Berkeley National Laboratory
LBL Publications

Title
Performance Analysis of Traditional and Data-Parallel Primitive Implementations of Visualization and Analysis Kernels

Permalink
https://escholarship.org/uc/item/3v87w9mt

Authors
Bethel, E Wes
Camp, David
Perciano, Talita
et al.

Publication Date
2020-10-05

Peer reviewed

Performance Analysis of Traditional and Data-Parallel Primitive Implementations of Visualization and Analysis Kernels

E. Wes Bethel, David Camp, and Talita Perciano
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
Email: [ewbethel,dcamp,tperciano]@lbl.gov

Colleen Heinemann
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
University of Illinois
Urbana-Champaign, IL, USA
Email: cheinemann@lbl.gov

Abstract—

Measurements of absolute runtime are useful as a summary of performance when studying visualization and analysis methods on computational platforms of increasing concurrency and complexity. We can obtain even more insights by measuring and examining more detailed measures from hardware performance counters, such as the number of instructions executed by an algorithm implemented in a particular way, the amount of data moved to/from memory, memory hierarchy utilization levels via cache hit/miss ratios, and so forth. This work focuses on performance analysis on modern multi-core platforms of three different visualization and analysis kernels that are implemented in different ways: one is "traditional", using combinations of C++ and VTK, and the other uses a data-parallel approach using VTK-m. Our performance study consists of measurement and reporting of several different hardware performance counters on two different multi-core CPU platforms. The results reveal interesting performance differences between these two different approaches for implementing these kernels, results that would not be apparent using runtime as the only metric.

I. INTRODUCTION

Runtime, the measurement of elapsed time for a given method, is a commonly used metric in performance studies. While useful as a summary of performance, runtime by itself may not be enough information to reveal why a method performs the way it does. At the other end of the spectrum, theoretical complexity analysis provides a useful understanding of the relationship between problem size N and runtime, e.g., $O(N^2)$ vs. $O(NlogN)$. In between these two, theoretical analysis and measurement of absolute runtime, is an area where we can gain more insight about the actual performance of a method. Performance analysis in this regime often consists of collecting additional performance measures, such as hardware performance counters, which reveal much more information about algorithmic performance than either runtime or complexity analysis alone.

Because computational platforms are rapidly evolving and increasing in complexity, a significant amount of research is underway to find pathways towards performance portability so that investments made in implementations perform well on a variety of different types of hardware now, and hopefully into the future. This paper studies the measured performance differences that can result when using alternative implementations of key visualization and analysis kernels. Most previous works in visualization performance analysis have focused on measuring and reporting only runtime. While runtime alone is a useful metric, deeper insights as to the nature of performance differences can result when considering additional metrics, such as hardware performance counters.

Our study focuses on three computational kernels common in visualization and analysis: isocontouring, particle advection, and stencil-based convolution. The stencil-based convolution is implemented using C++/OpenMP with VTK-m as the data-parallel alternative; the isocontouring and particle advection are written in C++ and invoke VTK methods as the traditional approach, and invoke VTK-m methods as the data-parallel alternative. This study is motivated by the desire to compare off-the-shelf implementations of staple algorithms, with an eye towards deepening understanding of performance differences.

This study is timely because there is significant interest in having methods perform well on evolving platforms where there is increasing node-level concurrency, which is the consequence of the end of Dennard scaling [1]. Some of the approaches for platform-portable parallelism include OpenMP-based loop parallelism [2], as well as VTK-m [3], where algorithms are constructed using sequences of data-parallel primitives. The visualization community has a keen interest in VTK-m, and most of the performance studies this far have used only runtime (and derivatives) as the performance metric. What our study adds is deeper insight and understanding of why these algorithms perform the way they do.

The contributions of this work include:
- A first-of-its kind analysis using hardware performance counters comparing traditional (VTK and C++) and emerging (VTK-m, data parallel primitives) implementations of staple methods: nearly all previous works focus exclusively on using runtime as the performance measure.
- Deeper insights into the types of performance impacts that can result from reformulating algorithm design using data-parallel primitives.

- The performance study methodology is straightforward and readily applicable to a much broader set of applications, and is particularly relevant given research interest in evolving methods to increasingly complex and heterogeneous hardware.
- Two new and different VTK-m implementations of a stencil-based computation, with a comparative performance analysis of both approaches that provides some insight into performance characteristics of different types of VTK-m worklets.

II. BACKGROUND AND PREVIOUS WORK

A. The Evolving Computational Landscape of Visualization and Analysis

In response to the end of Dennard scaling, system and processor architectures have evolved to use deepening memory hierarchies combined with increasing node-level parallelism [1]. Over the past decade or so, numerous programming models and environments — such as OpenMP [4], OpenCL [5], and Kokkos [6], to name a few — have emerged where the objective is to provide for platform portability as well as efficient shared-memory parallelism.

For distributed-memory parallelism, codes like VisIt [7] and ParaView [8] use the Message Passing Interface (MPI) [9] as the programming model to achieve platform portability. These tools each leverage VTK's core algorithms [10], and manage their execution in slightly different ways that are implementation specific.

In the visualization community, the VTK-m library [3] follows a similar technology trajectory, where user code can be executed with one of several different device- or platform-level backends, such as TBB [11], OpenMP [4], or CUDA [12]. VTK-m is positioned to be a follow-on to the VTK library [10], which has served as a stalwart in the visualization community for more than two decades but is limited to predominantly serial use due to a combination of factors, including close entanglement of data structures and execution models and use of static variables that maintain state and are hence not thread-safe.

Given the community interest in VTK-m, our work here seeks to provide more insight into the performance differences that result when implementing algorithms using traditional, either straight C++ or C++/VTK, with emerging approaches like VTK-m that offer platform portability using data parallel primitives (DPPs) as the algorithmic building block.

B. Comparing Traditional and VTK-m Implementations

Several recent works have focused on developing VTK-m implementations of key algorithms and comparing them to custom, traditional implementations. Larsen, et al. 2015 [13] study the performance of a raytracing renderer implemented using data-parallel primitives with two reference implementations. Their reported metric is frame/second, which is a derivative of runtime. Pugmire, et al. 2018 [14] present a VTK-m implementation of particle advection. They compare the performance of their implementation, which uses a *parallelize over seeds* approach, against two others: a hand-coded implementation, and the VisIt [7] application. All their studies rely on measurements of runtime.

In the area of graph algorithms, Lessley, et al., 2017 [15] describe a VTK-m implementation of a method for maximal clique enumeration and compare its performance with reference implementations on CPU and GPU platforms. Lessley, et al., 2018 [16] present a VTK-m implementation of a method for probabilistic graphical modeling optimization, a form of unsupervised learning, and compare its performance to a reference implementation on a CPU. In both these works, the basis for comparison is elapsed runtime.

Recent work by Perciano, et al., 2020 [17] delves more deeply into use of LIKWID and performance counters to gain deeper insight into performance differences between three different shared-memory parallel implementations of an unsupervised learning method using graphical model optimization: C++/pthreads, C++/OpenMP, and VTK-m. That study's finding is that the VTK-m algorithmic reformulation executes far fewer instructions than the traditional counterpart, owing to the significant algorithmic refactorization that occurs going from traditional coding style to one that uses data parallel primitives. Our work is similar in that we are diving deep into the collection and analysis of performance counters, and using them to shed insight into the performance differences between traditional and VTK-m implementations. However, we are focusing on comparing staple visualization kernels rather than graphical model optimization.

One primary difference between these previous works, except for Perciano, et al., 2020 [17], and our work here is the deeper introspection provided by using detailed hardware performance counters. These additional metrics offer the ability to better understand why a given code performs better or worse in a particular set of circumstances, and also helps to provide a more sound basis for performance analysis.

III. DESIGN AND IMPLEMENTATION

The focus of this section is on presenting concepts related to design and implementation using "traditional" and data-parallel approaches. In the case of isocontouring (§III-A) and particle advection (§III-B), we briefly describe the implementations used in our study. The core VTK and VTK-m implementations existed already as part of previous work done by others. In the case of stencil-based convolution (§III-C), we describe our traditional and VTK-m implementation in some detail, as this is new work that did not exist prior to this study.

A comment about parallelism: while VTK-m is parallel capable by its very design, VTK is not. Two of the three methods we present (isocontouring, particle advection) use VTK implementations for the "traditional" approach: we created applications that load data and invoke the appropriate VTK (or VTK-m) method. The VTK implementation of these methods themselves are not parallel, nor are they readily parallel capable: due to the existence of thread-unsafe constructs, such as static variables inside VTK that hold state, they cannot be invoked in parallel by multiple concurrent execution threads.

A. Isocontouring

Isocontouring in 2D and 3D is a staple visualization method because it is useful for showing geometric structure associated with specific data values. The seminal method for computing isosurfaces is from Lorenson and Cline, 1987 [18].

This algorithm operates on scalar fields by considering the cell formed by eight neighboring points that form a hexahedron. For each such cell, the algorithm performs a classification step on each vertex of the cell, where the classification is the union of booleans indicating if the scalar value at each node is above or below the contouring value. Then, the algorithm uses this classification as an index into a lookup table containing a specification of a triangulation of the surface passing through a cell with that given configuration: some configurations have one triangle, others may have two or more triangles. Then, the algorithm computes isosurface triangle vertex values using an interpolant, typically trilinear interpolation. The triangles from all such computations at each cell comprise the resulting isosurface.

It is well beyond the scope of this paper to enumerate the improvements and enhancements over more than 30 years since this algorithm first appeared in 1987. In brief, some of the key improvements consist of resolving some of the topological ambiguities that arose in the original formulation [19], and use of spatial data structures to accelerate finding cells that contain the isosurface (c.f., Newman and Yi, 2006 [20]).

For the purpose of our study, we are interested in using and benchmarking two reference implementations: the `vtkContourFilter` method from VTK, and the `vtkm::filter::Contour` method in VTK-m, which is a reimplementation of the `vtkContourFilter` [21]. Both the VTK and VTK-m methods perform the same fundamental algorithmic processing steps — cell classification and triangle generation — but use different mechanisms for doing so.

The VTK-m implementation uses a combinations of `worklets`, which are instantiations of data parallel primitives. For example, it uses a `WorkletVisitPointsWithCells` for the `ClassifyCell` operation. This type of worklet will "apply a function (the operator in the worklet) on all elements of a particular type (such as points or cells) and creates a new field for those elements" [22], with the added feature that nearby, incident cells are also accessible. It also uses other types of worklets, such as `ScatterPermutation` and `ScatterCounting`, which are used to create multiple outputs (each of which satisfies some condition) from an input, as well as others. Both VTK and VTK-m implementations are sophisticated works of complex design and engineering.

For this study, we created two applications, each of which loads data and writes output. One version invokes VTK's `vtkContourFilter`, and the other invokes VTK-m's `vtkm::filter::Contour` method. One primary difference between the VTK and VTK-m implementations concerns parallelism. The VTK-m implementation, through its execution environment, is capable of running worklets in shared-memory parallel fashion using a number of different device backends (e.g., OpenMP, TBB, CUDA, etc.). The VTK-m device execution environment will decompose a worklet's workload into chunks that execute in parallel. This type of decomposition and parallel operation does not exist in much of VTK, including the `vtkContourFilter`, which is limited to serial use only.

B. Particle Advection

Particle advection is a technique for calculating the trajectory a particle follows through a flow field. An integral curve — commonly referred to as a pathline — encodes the trajectory of a single massless particle, which in turn gives insight into the flow behavior in the area surrounding the particle's path.

Advection constructs integral curves, which are continuous functions tangential to the vector field. The curves are solutions to an ordinary differential equation, and, for an integral curve I and vector field V, can be represented as:

$$\frac{d}{dt}I(t) = v(I(t), t) \tag{1}$$

where $I(t_0) = x_0$, and for a seed point at location x_0 at time t_0 [23].

As with the isocontouring study, our focus is to compare performance of two existing reference implementations, one from VTK and the other from VTK-m. From VTK, we are using the `vtkStreamTracer` method, and from VTK-m, we are using the `vtkm::filter::Streamline` method. Both methods take as input a vector field and then perform integral curve calculation to compute streamlines from a set of input seed points with a user-specified integrator (e.g., Runge-Kutta4). Our application that uses the VTK implementation of the `vtkStreamTracer` method is serial in nature and cannot be trivially parallelized due to explicit limitations in the VTK implementation.

From VTK-m, the `vtkm::filter::Streamline` implementation uses a parallelize-over-seeds approach, which is described in Pugmire, et al., 2018 [14]. Their implementation uses a `Field Map` worklet, where the worklet is invoked at each input datum, which in this case, consists of a list of seed points. VTK-m's internal execution model will parallelize this operation by invoking the worklet in parallel, where each invocation will process some number of input seed locations.

C. Stencil-based Convolution

In a stencil based computation, each point of a multidimensional grid is updated with contributions from its neighbors. This form of computation lies at the heart of many different types of scientific computations, such as solving partial differential equations on a regular, structured grid (c.f., [24]). We include this computationl pattern because it is common in many types of computing applications, such as numerical simulation, image analysis/computer vision, convolutional neural networks, and more.

In our study, we focus on a particular type of stencil-based computation, namely image convolution. This form

```
1 float smoothPixel(Si, Sj, S, R, weights)
2 {
3     // compute the weight sum of pixels nearby
4     // this code doesn't handle edge conditions
5     // and assumes sum of weights[i,j] = 1.0
6     float sum = 0.0;
7     for (int j=0; j<R; j++)
8         for (int i=0; i<R; i++)
9             sum += weights[i,j]*S[Si+i,Sj+j]
10     return sum;
11 }
```
Listing 1. Stencil computation in 2D: performs sum of product of nearby pixels with weights.

of computation is a structured memory access code, where memory is accessed in a regular and predictable fashion. In particular, our computation performs Guassian smoothing, which is a spatial image processing filter. In this computation, each destination pixel $d(i)$ is a sum of nearby pixels averaged using a weighting scheme that gives more weight to pixels closer to i, and less pixels further away (Eq. 2).

$$d(i) = \sum g(i, \bar{i}) \qquad (2)$$

where the Gaussian weights are given by

$$g(i, \bar{i}) = e^{-\frac{1}{2}\left(\frac{\delta(i,\bar{i})}{\sigma_d}\right)^2} \qquad (3)$$

In Eq. 3, $\delta(i, \bar{i})$ is the distance between pixels i and $\bar{i}$. σ is a parameter that defines whether the weights are more tightly focused around the source pixel (somewhat less smoothing), or if the weights are more diffuse, and give more weight to pixels further away (somewhat more smoothing).

In practice, this type of computation is bounded such that a stencil of size R^N, in N dimensions, contains filter weights computed such that the integral of Eq. 3 over the set of R^N weights sums to 1. Then, computing the smoothed pixel value consists of performing a sum of products of the source pixel weights with the filter weights. An example for 2D image convolution appears in Listing 1, and this computation trivially generalizes to N dimensions.

An application that uses the smoothPixel method will iterate over the pixels/voxels in a source image/volume, and invoke this method at each pixel/voxel. This type of application is straightforward to parallelize in that the sense that the computation at each $d(i)$ is independent of the computation at all other locations; these computations may be performed independently and in parallel.

Listing 2 shows one such parallelization using OpenMP. In that listing, each of the OpenMP threads is assigned a row (scanline) of pixels to work on. OpenMP does assignments using one of several different strategies, such as round-robin, etc. depending on the setting of a runtime environment variable (c.f. [4]). This type of parallelization is relatively coarse-grained in the sense that each thread is assigned a significant amount of work.

One potential VTK-m implementation of this stencil operation is to use the same computational kernel shown in Listing 1, but then let VTK-m manage how this computation

```
1 #pragma omp parallel for
2 for (j=0; j<imageHeight; j++)
3     for (i=0; i<imageWidth; i++)
4         destImage[i,j] = smoothPixel(i, j, S, R,
         weights)
```
Listing 2. OpenMP parallel computation: each thread gets a scanline to process.

is invoked. The basic concepts are as follows. First, we define a Field Map worklet that has input and output parameters that are "arrays"; these are essentially std::vector objects. Then, after populating the input array with the source image, we invoke the dispatcher that invokes the ImageConvolutionWorklet, as shown in Listing 3. Then, VTK-m will invoke that worklet once per array item using one of several different potential device backends, depending upon user build configuration options and runtime choices (see the VTK-m User's Manual for more details [22]). In the case of our application, this worklet is invoked once per input pixel. The result is a much finer-grained approach to parallelism than the scanline-based parallelism shown in Listing 2. We refer to this design as the VTK-m-*FM* method, where *FM* refers to the use of the *Field Map Worklet*, and which here uses an explicit indexing computation to access specific locations in the input and output image data arrays.

```
1 template <typename InputArrayType, typename
     OutputArrayType>
2   VTKM_EXEC void operator()(const InputArrayType &
     inputImg, OutputArrayType & outputImg,
3   vtkm::Id indx) const
4   {
5       /// assume private R: stencil size
6       /// explicit indexing: compute (i,j) indices
     from 1D indx
7       int jVal = indx / nCols - R/2; // which row
8       int iVal = indx % nCols - R/2; // which column
9
10      float sum = 0.0;
11      for(int j=0; j<R; j++)
12          for(int 0; i<R; i++)
13              sum += weights[i,j]*inputImg[iVal+i,jVal
     +j];
14      outputImg.Set(indx, sum);
15  }
16 ...
17 /// then invoke it in main():
18 vtkm::worklet::DispatcherMapField<vtkm::worklet::
     ImageConvolutionWorklet> dispatcher(myWorklet);
19 dispatcher.Invoke(inputImageArray, outputImageArray)
     ;
```
Listing 3. VTKm-*FM* algorithm: In VTK-m, execution environment iterates over the field and invokes the smoothPixel worklet in parallel.

There are other potential implementations of this stencil that we could pursue in VTK-m, implementations that could potentially take greater advantage of VTK-m's data parallel primitives (DPPs). These DPPs include operations like Reduce, Sort, CopyIf, ScanExclusiveByKey, and so forth. Some interesting future work would entail exploring recasting the stencil computation in terms of using VTK-m's DPP Device Algorithms, as has been done with other types of computa-

tions, such as probabilistic graphical modeling optimization [16].

We did explore a second VTK-m implementation, namely one that uses a `Point Neighborhood Worklet`, which we refer to as the VTK-m-*PN* algorithm. Like the VTK-m-*FM* algorithm, VTK-m invokes the worklet once per input grid location. One significant difference is that the VTK-m-*PN* algorithm may access field values of nearby points within a neighborhood of a given size, as opposed to having access to the entire mesh. In other words, the VTK-m-*FM* method needs to do its own indexing: it is handed a 1D input index, and then needs to use this input to produce an index into a multidimensional array. In contrast, methods that use the `Point Neighborhood Worklet` will use an `inputData.Get()` method to access data, rather than using an index computation. Our VTK-m-*PN* algorithm is shown in Listing 4.

```
1  template < typename InputFieldPortalType >
2    VTKM_EXEC typename InputFieldPortalType ::
     ValueType operator ()(
3    const vtkm :: exec :: FieldNeighborhood <
     InputFieldPortalType >& inputField ,
4    const vtkm :: exec :: BoundaryState & boundary )
      const
5    {
6        auto minIndices = boundary.
     MinNeighborIndices (this -> stencilRadius );
7        auto maxIndices = boundary.
     MaxNeighborIndices (this -> stencilRadius );

9        float sum=0.0;
10     for(vtkm::IdComponent j=minIndices [1]; j<=
     maxIndices[1]; ++j)
11         for (vtkm::IdComponent i=minIndices[0]; i
     <=maxIndices[]0]; ++i)
12             sum += inputField.Get(i, j) * this->
     filterWeights[f(i,j)];

14         return static_cast <T> (sum);
15   }
16 ...
17 /// then in main():
18 // set the image dimension sizes
19 vtkm::cont::CellSetStructured<2> myGrid2D
20 myGrid2D.SetPointDimensions(vtkm::Id2(nCols, nRows);

22 // create the dispatcher and invoke it
23 vtkm::worklet::DispatcherPointNeightborhood<vtkm::
     worklet::ImageConvolutionWorklet> dispatcher(
     myWorklet);
24 dispatcher.Invoke(myGrid2, inputImageArray,
     outputImageArray);
```
Listing 4. VTK-m-*PN* algorithm: similar to the VTK-m-*FM* algorithm, but without the global indexing computation as VTKm provides a view only to the local mesh/image neighborhood

IV. RESULTS

A. Research Questions

One of the primary areas of study is to better understand the key performance characteristics of a "traditional" implementation with an implementation that uses VTK-ms data-parallel primitives (DPPs). While we do report runtime, we collect and study hardware performance counter data to gain deeper insight into the potential factors contributing to runtime differences.

Given that the VTK implementations of isocontouring and particle advection are serial, we focus our tests on comparing serial VTK and VTK-m implementations. While these VTK-m implementations do execute in parallel, we are not performing any scaling comparisons here between the VTK and VTK-m implementations of isocontouring and particle advection. In contrast, the stencil computation runs in parallel in all configurations, and so we include a scaling study for that method.

Given these limitations, our primary objective is to gather hardware performance counters and analyze the results to better understand the reasons for runtime performance differences between VTK and VTK-m implementations.

B. Methodology

1) Computational Platforms: Intel Xeon Phi and Intel Haswell. `Cori.nersc.gov` is a Cray XC40 system comprised of 2,388 nodes containing two 2.3 Ghz 16-core Intel Haswell processors and 128 GB DDR4 2133 MHz memory, and 9,688 nodes containing a single 68-core 1.4 GHz Intel Xeon Phi 7250 (Knights Landing) processor and 96 GB DDR4 2400 GHz memory. For our experiments, we use the KNL processor, where each core has a 32 KiB L1 cache, and each pair of cores share a 1 MiB L2 cache (L2 is the last level cache on this platform). On this platform we are using Intel's C/C++ compiler, icc (ICC) 19.0.3.199 20190206. Compiler flags for both VTK and VTK-m include the following optimization options to enable vectorization: `-O3 -march=knl -mtune=knl -DNDEBUG -funroll-loops`.

IvyBridge. `Allen.lbl.gov` is an Intel(R) Xeon(R) CPU E5-2609 v2 containing 2 2.5 GHz 4-core Intel Xeon IvyBridge EN/EP/EX processors and 32 GB of memory. Each core has a 32 KiB L1 cache, a 256 KiB L2 cache, and all cores share a 10 MiB L3 cache (L3 is the last level cache on this platform). On this platform, we are using Intel's C/C++ compiler, icc (ICC) 19.1.0.166 20191121. Compiler flags for both VTK and VTK-m include the following optimization options to enable vectorization: `-O3 -march=ivybridge -mtune=ivybridge -funroll-loops`.

2) Software Environment: VTK and VTK-m. We are using VTK version 8.2.0, and VTK-m version 1.5.0. For all tests, we use VTK-m's OpenMP backend, which generates OpenMP shared-memory parallel code that runs on multi-core CPU platforms. While VTK-m is capable of emitting CUDA-based code, and while modern versions of OpenMP are also capable of emitting CUDA-based device code, we did not include GPU as one of the platforms in this study; this will make for interesting future work.

3) Data Sets and Algorithmic Parameters: For the stencil-based smoothing and isocontouring studies, we are using a scientific dataset that was obtained by the Lawrence Berkeley National Laboratory Advanced Light Source X-ray beamline 8.3.2 [25]. This dataset contains cross-sections of a geological

sample and conveys information regarding the x-ray attenuation and density of the scanned material as a gray scale value. The original data consists of a stack of 500 images at resolution 1290×1305.

For the stencil-based smoothing study, we are using an augmented version of the dataset where we use one 2D slice, and then replicate it twice in X and Y to produce an image of resolution 5160×5220. The size of the convolution kernel is 19×19 pixels, and $\sigma = 0.33$.

For the isocontouring study, we use a 400^3 subset of a processed version of the original $1290 \times 1305 \times 500$ volume. The processing consists of multiple stages in an image analysis pipeline, where we use a custom, high-quality image segmentation algorithm based on an unsupervised learning method [16], and then run the segmented image through a 3D version of the stencil-based smoothing code presented in this study. We performed contouring with an isocontour level of 15, which results in the surface that appears in Fig. 1. Note that since we are focusing on results from serial runs only, parallel performance variations that might result from various conditions such as data-dependent factors or domain decomposition effects are not of concern in these studies.

For the particle advection study, we are using a vector field data set produced by the NIMROD code, which is performing numerical modeling of a magnetically confined plasma in a tokamak device [26]. The magnetic field lines, through which we are doing particle advection, travel around and around the toroidal-shaped domain in a helical fashion. As a result, most particles will not exit the domain, and the amount of work for each seed point will be about the same over the lifetime of the integration. For both VTK and VTK-m runs, we use 500 seed points placed equidistantly along a diagonal running through the domain, and compute particle positions for 1000 steps using the Runge-Kutte4 integrator. Note that since we are showing only serial VTK-m results, parallel performance variations as a function of seed point placement are not of concern in these studies.

Hardware Performance Counters. For this study, we are leveraging the LIKWID software infrastructure v4.3.4 [27], [28]. LIKWID is a set of lightweight, command-line tools that are useful for obtaining measurements of hardware performance counters on Linux platforms in user space. While LIKWID is capable of collecting performance counters for an unmodified application, we make use of LIKWID's Marker API to collect performance data only in the section of code containing the computational algorithm; we do not include other operations, like data I/O, in the performance counter data. Table I describes the performance counters we gather for analysis in all the experiments.

4) Testing Procedure: For each of the different kernels, we execute both VTK and traditional on both the Ivy Bridge and KNL platforms using different datasets (§IV-B3). We are collecting and analyzing the performance counters shown in Table I. Of that set, we report a subset in the individual subsections below: total number of instructions executed, number of scalar and vector floating point instructions executed, cycles-

TABLE I
HARDWARE PERFORMANCE COUNTERS AND OTHER MEASURES WE USE IN OUR EXPERIMENTS.

| Performance counter, measure	(Source) Description
`INSTR_RETIRED_ANY`	(LIKWID) Shows how many instructions were completely executed, and does not include speculative instruction loads [29].
`FLOPS`, `FLOPS_DP`, `FLOPS_SP`	(LIKWID) Count of the number of single- and double-precision floating point operations that were executed. On the KNL platform, there is no way to differentiate between single- and double-precision operations.
Cycles Per Instruction (CPI)	(LIKWID) A derived metric computed as the quotient of CPU_CLK_UNHALTED_CORE / INSTR_RETIRED_ANY to give an estimate of the number of clock cycles per instruction (c.f. [30]).
Vectorization ratio	(LIKWID) The ratio of "packed" FLOPS to the sum of all FLOPS. On the KNL platform, these counters may also include integer arithemtic instructions [28].
`L3CACHE`	(LIKWID) Measures the locality of your data accesses with regard to the L3 cache. It reports the L3 request rate, L3 miss rate, and L3 miss ratio [28]. (Ivy Bridge only)
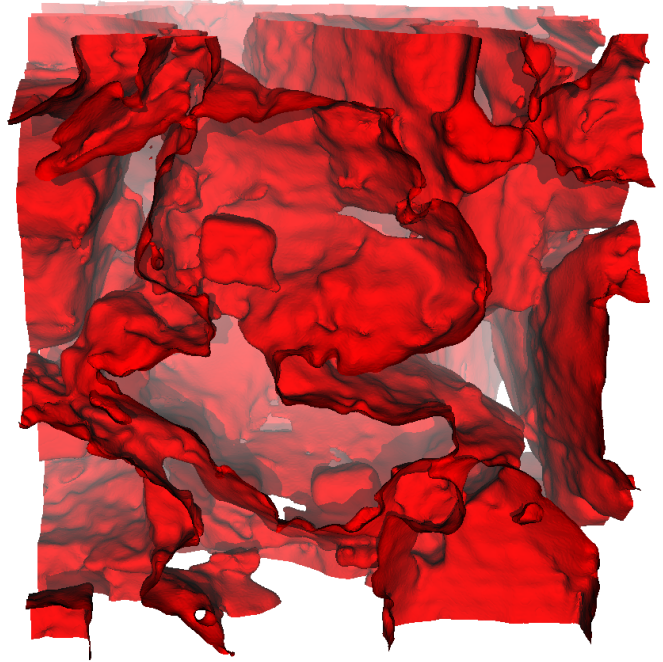
Fig. 1. Isocontouring of a 400^3 subset of smoothed, segmented Sandstone data results in 3,038,366 tris rendered using VisIt.

per-instruction, vectorization ratio, and L3 cache miss ratio (on the Ivy Bridge) platform.

Counter/Measure	Code version	Platform	
		Ivy Bridge	KNL
Runtime (secs)	VTK	3.97	5.92
	VTK-m	5.48	16.74
INSTR_RETIRED_ANY 10^9	VTK	21.96	11.10
	VTK-m	22.01	21.10
FLOPS scalar 10^9	VTK	0.08	0.50
	VTK-m	0.36	0.54
FLOPS non-scalar 10^9	VTK	≈ 0.00	0.51
	VTK-m	0.01	0.91
Vectorization %	VTK	≈ 0.00	50.50
	VTK-m	3.35	62.77
CPI	VTK	0.44	0.76
	VTK-m	0.55	1.11
L3 Miss Ratio %	VTK	23.37	N/A
	VTK-m	79.03	N/A

C. Isocontouring Study

Visual results from this study are shown in Fig. 1, and the performance counter and other data appear in Table II. One observation we draw from the performance data is that for this problem configuration, on the Ivy Bridge platform, the serial VTK version runs in about 72% of the time of the serial VTK-m version. On the KNL, this difference is more pronounced, with the VTK version running in about 35% of the time of the VTK-m implementation. This significant difference in runtime is something of a surprise given that both versions are executing the same type of Marching Cubes algorithm, and neither makes use of any special data structures to accelerate the search for cells that containing the isolevel: each implementation must examine each and ever cell in the entire domain, test its nodes for above/below the isolevel, and then generate triangles accordingly.

To better understand why there is such a performance discrepancy, we turn to the hardware performance counter data shown in Table II. On the Ivy Bridge, the number of overall number of instructions executed (INSTR_RETIRED_ANY) is about the same. However, the mix of instructions is different between the two implementations, with the VTK code executing about 22% the number of FLOPS as the VTK-m version.

For additional clues, we look to the L3 Miss Ratio. The VTK-m L3 Miss Ratio is more than $3\times$ that of the VTK code. This combination, of increased L3 cache misses, which will introduce stalls into the VTK-m code, which may or may not be the source of higher CPI. We know that the VTK-m code consists of sequences of data parallel primitives, where the output of one stage goes on to be the input of the next stage.

We can see the effects of this additional memory handling overhead quite clearly in terms of lower cache utilization, which contribute to the slower runtime and higher CPI. The runtimes shown here reflect the product of CPI and number of instructions executed [30]. And the source of higher CPI for the VTK-m code is most likely the execution of memory access instructions that are more time consuming.

On the KNL, we see the VTK-m version is executing about $2\times$ the number of instructions, and has a CPI value that is almost $1.5\times$ larger than the VTK version. In terms of FLOPS being executed, the number of scalar FLOPS is about the same, but the VTK-m code executes almost $2\times$ the number of vector FLOPS. Due to limitations of the KNL, we're unable to discern between single-, double-precision and integer vector operations [28]. We see the effects of the VTK-m increased memory operations directly through the elevated CPI and runtime.

One observation we draw is that use of VTK-m and recasting the algorithm to use DPPs comes at a cost, namely less effective cache utilization and a serial runtime that exceeds that of the reference VTK implementation serial runtime. This observation is not universally true: for example, Perciano et al., 2020 [31] show that recasting a graphical model optimization problem into VTK-m using DPPs results in an implementation that executes far fewer instructions than its C++ counterpart. Further studies would include adding finer-grained instrumentation with the LIWKID Marker API into VTK-m internals to isolate and study these buffer effects, including factors such as the impact of blocking and chunking the VTK-m performs in its back-ends for each device.

Another observation is that the Intel compiler is able to achieve high levels of code vectorization on the KNL for both VTK and VTK-m implementations, but does not do as well on the Ivy Bridge platform. More investigation is needed to determine why the compiler is having trouble on the Ivy Bridge with these codes. Regardless, the vectorization level does not seem to offer any significant insights into the relative performance differences.

D. Particle Advection Study

Visual results from this study are shown in Fig. 2, and the performance counter and other data appear in Table III. From this data, we see that the VTK-m implementation runs faster on both platforms; on the Ivy Bridge, it runs in about 89% of the VTK method, and on the KNL, the difference is more pronounced, where it runs in about 26% of the VTK method.

On the Ivy Bridge, we see the VTK-m implementation executes far fewer total instructions (INSTR_RETIRED_ANY), only about 23% as many as the VTK method. Proportionally speaking, the VTK code is executing a much greater percentage of non-floating point instructions as evidenced by comparing the ratio of FLOPS/INSTR_RETIRED_ANY between the two. These extra non-floating point instructions are most likely due to extensive bounds and indexing calculations: as a particle is advected, the next step of the algorithm is to determine in which mesh cell the particle is contained.
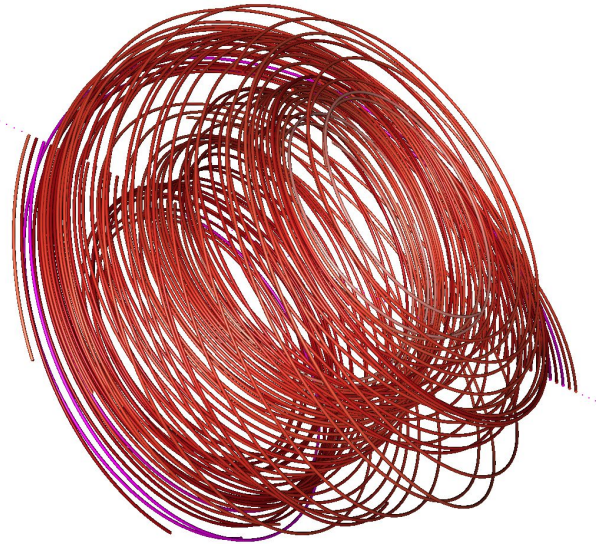
Fig. 2. Streamline results from the particle tracing method run using the NIMROD dataset rendered using VisIt.

Counter/Measure	Code version	Platform	
		Ivy Bridge	KNL
Runtime (secs)	VTK	7.68	7.72
	VTK-m	6.81	2.03
INSTR_RETIRED_ANY 10^9	VTK	25.56	5.92
	VTK-m	5.91	2.34
FLOPS scalar 10^9	VTK	0.27	0.85
	VTK-m	0.11	0.26
FLOPS non-scalar 10^9	VTK	0.01	0.68
	VTK-m	$= 0.00$	0.22
Vectorization %	VTK	2.73	44.52
	VTK-m	$= 0.00$	45.75
CPI	VTK	0.74	1.94
	VTK-m	2.78	1.28
L3 Miss Ratio %	VTK	33.53	N/A
	VTK-m	57.09	N/A

We know something about the VTK-m implementation: it is a recasting of the particle advection into a series of data parallel primitives, where output from one stage is input to the next stage. And as we saw with the isocontouring study, this type of design pattern can be result in less efficient use of the memory hierarchy. We see evidence of this effect when looking at the L3 Miss Ratio in Table III, as well as at the significantly higher CPI, which reflects the impact of costly non-cache memory accesses.

On the KNL platform, the runtime difference between the two is even more pronounced. Again, we see the VTK implementation executes more overall (about $2.5\times$) and numerical instructions (between about $3.1\times$ and $3.3\times$). We believe the increased CPI on the KNL for the VTK implementation to be a result of less efficient cache utilization: particle advection memory accesses are unstructured, so there is little opportunity for speculative memory fetches. The VTK-m code reorders memory accesses to fit its DPP execution pattern, and in this case, appears to have better cache utilization as shown by a lower CPI. For the VTK-m implementation, the combination of fewer instructions and lower CPI combine to produce a dramatically lower runtime.

E. Stencil-based Smoothing Study

The test matrix in this portion of the study consists of three different implementations of the stencil-based smoothing (traditional C++/OpenMP, VTK-m-*FM*/OpenMP, and VTK-m-*PN*/OpenMP) run at varying concurrency on two different platforms: Ivy Bridge: $P \in (1, 2, 4, 8)$; KNL: $P \in (1, 2, 4, 8, 16, 32, 64)$. For each test run, we use LIKWID to obtain the hardware performance counters and report them in Table IV.

Since we are doing a strong-scaling configuration in these tests, we expect the runtime to drop with increasing concurrency. We see such behavior in Fig. 3, where absolute runtime increases at varying concurrency, as well as speedup compared to serial. In this case, all algorithms exhibit near-perfect scaling characteristics. This result is expected since this algorithm is "embarrassingly parallel": the computation of each output pixel is completely independent of the computation at all other output pixels, and these computations may be performed in parallel.

While it is the case all methods exhibit near-perfect scaling, we see that the VTK-m-*PN*/OpenMP method has runtime far greater than the other two methods. For evidence of why this is the case, we turn to hardware performance counter data, which we present in Table IV. The performance data in this table is from a serial run of each method. What is not shown in the table is data from varying concurrency runs. It turns out that these values do not change in any significant way with increasing concurrency on this problem with these codes.

The first, and most significant, observation is that VTK-m-*PN* executes about 5 times as many instructions as the other implementation, which would account for its 5-fold increase in runtime compared to the other methods. This result is most likely due to the effort on the part of VTK-m to prepare point neighborhood collections of data from the original input data. In contrast, the other two methods do their own indexing into the original data array, an approach that appears to result in far fewer instructions being executed.

The second observation is that the VTK-m-*PN* method executes far fewer "packed", or vector, floating point instructions than the other two methods. This result is most likely due to the fact that the VTK-m-*PN* worklet is accessing data through a method, e.g., `getData(i, j, k)`, rather
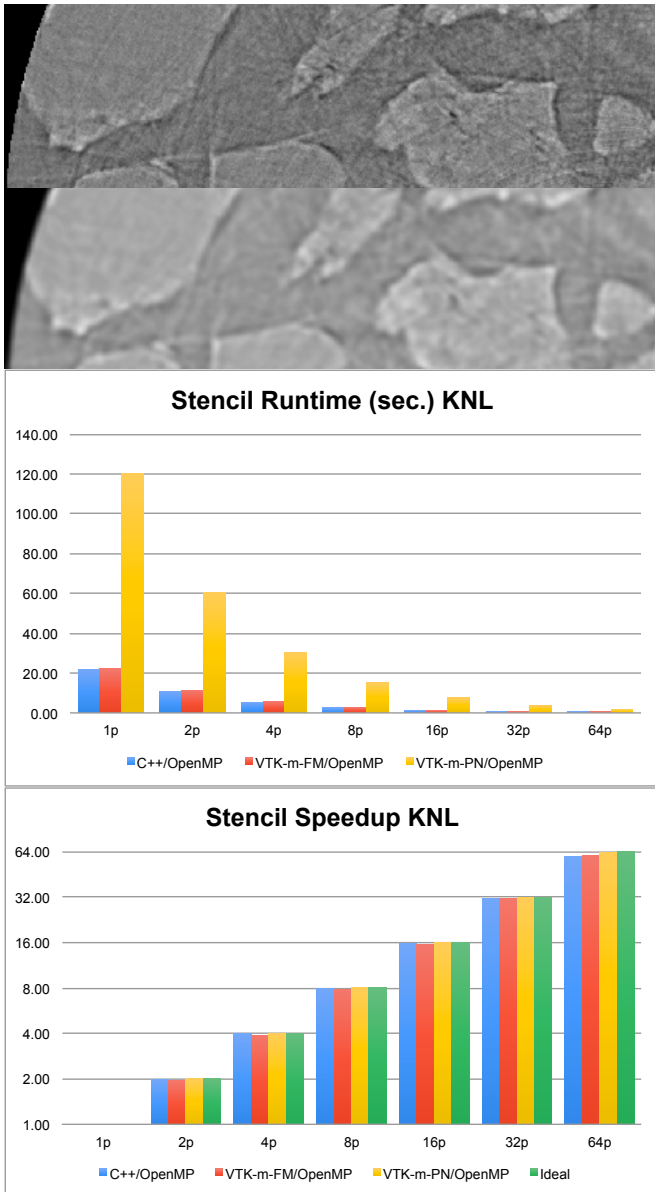
method, VTK-m parallelizes the call to the worklet, which is invoked once per output pixel. Therefore, VTK-m-*FM* is operating at a finer-grained level of parallelism, compared to the C++/OpenMP. The resulting performance data suggest that the C++/OpenMP approach, with its coarser-grained approach to parallelism, runs a bit faster most likely due to its better use of temporal cache locality, as evidenced by the L3 cache data on the IvyBridge platform.

Fig. 3. (Top) Original image data of a geological sample, obtained by micro computed tomography at the Advanced Light Source. (Middle-top) Results of Guassian smoothing using the C++/OpenMP implementation with $5x5$ stencil. (Bottom 2 images) Absolute runtime and speedup of three stencil implementations on the KNL platform at varying concurrency with a $19x19$ stencil on a high-resolution ($5K \times 5K$) input image.

TABLE IV
STENCIL CONVOLUTION PLATFORM AND HARDWARE PERFORMANCE
COUNTERS FOR THE SANDSTONE5K DATASET, 19×19 STENCIL SIZE,
IVY BRIDGE AND KNL PLATFORMS.

Counter/Measure	Code version	Platform	
		Ivy Bridge	KNL
Runtime (secs)	C++	7.59	21.65
	VTK-m-*FM*	8.33	22.09
	VTK-m-*PN*	37.35	120.49
INSTR_RETIRED_ANY 10^9	C++	57.60	41.51
	VTK-m-*FM*	59.60	42.59
	VTK-m-*PN*	249.37	215.59
FLOPS scalar 10^9	C++	5.76	4.47
	VTK-m-*FM*	5.77	4.47
	VTK-m-*PN*	30.24	20.81
FLOPS non-scalar 10^9	C++	8.68	14.93
	VTK-m-*FM*	8.67	14.99
	VTK-m-*PN*	$= 0.00$	1.16
Vectorization %	C++	60.11	76.97
	VTK-m-*FM*	60.05	77.04
	VTK-m-*PN*	$= 0.00$	5.27
CPI	C++	0.32	0.76
	VTK-m-*FM*	0.34	0.77
	VTK-m-*PN*	0.37	0.83
L3 Miss Ratio %	C++	15.85	N/A
	VTK-m-*FM*	37.14	N/A
	VTK-m-*PN*	47.84	N/A

F. Discussion of Results

One observation from these studies is that the hardware performance counters are useful for understanding more about why two different methods have different runtimes. In some cases, they are executing different absolute numbers of instructions. In other cases, the type of instruction being executed can take more time: instructions that load/store memory can take significantly longer than simple arithmetic instructions. In other cases, demographics of compiler-generated scalar or vector arithmetic instructions can impact overall runtime.

The three cases we present all exhibit different aspects of why a method might have better or worse runtime than another. In some cases, the way an algorithm is implemented, such as VTK vs. VTK-m, can have a dramatic impact on overall number of instructions, a fact that is corroborated by other recent studies (c.f., [17]). In other cases, the buffer management needed to implement a complex, multi-stage processing pipeline may trigger more memory movement instructions,

than via an index that is a function of a loop variable, e.g., `data[i+offset]`.

A third observation is that the C++/OpenMP implementation appears to have a very slight performance edge compared to the VTK-m-*FM* approach in terms of runtime, number of instructions executed, and L3 cache utilization. This result is due to the differences in the way these two methods are parallelized. In the case of C++/OpenMP, each OpenMP "thread" processes an entire row of pixels; OpenMP parallelism happens at the outer loop level, resulting in a somewhat coarse level of parallelism. In the case of the VTK-m-*FM*

which may be more expensive and result in higher CPI values, and we see evidence of this in two of the examples.

V. CONCLUSIONS AND FUTURE WORK

This study demonstrates the value of using a methodology to collect and analyze hardware performance counter data to better understand the reasons behind performance differences in multiple implementations of three kernels common in visualization and analysis processing: isocontouring, particle advection, and stencil-based convolution. While runtime alone is useful, having performance counter data enables analysis at a deeper level. Although this approach of using hardware counters to do performance analysis is not new, it is relatively new to the visualization community, where previous performance studies have focused primarily on using runtime and runtime derivatives as the performance measure.

While the stencil-based computation work includes a modest strong scaling study, our results focus primarily on comparing serial implementations of methods implemented in VTK and VTK-m. While VTK-m is intrinsically (shared memory) parallel, the VTK-based implementations are inherently serial due to limitations inside VTK itself. Future work may include finding workarounds to VTK limitation so that it is possible to do more widespread performance comparison studies across a larger set of key VTK and VTK-m algorithms.

While our work focuses on a multi-core CPU platform, VTK-m is capable of emitting code that can run on a GPU. According to the OpenMP standard [32], OpenMP is capable of emitting GPU device code (called "device offload"), however there are significant constraints on exactly what may be successfully offloaded to the GPU. While it is likely possible to do a direct custom C++/OpenMP comparison with VTK-m implementation, doing so with an C++/OpenMP application that is engineered to invoke VTK methods may not be feasible, due to OpenMP device offload limitations.

Additional future work will include doing finer grained studies that isolate and quantify the cost of buffer and memory management needed for VTK-m and DPP-based patterns. One dimension of that future work would include studies that examine the performance impact of changing the blocking factor used by the VTK-m back-end as it divides up data for worklets to execute in parallel. Similarly, while this study asks some preliminary questions for each of these three methods and their multiple implementations, a significant amount of in-depth analysis work remains to fully characterize performance, such as analyzing memory system utilization and its relationship to data management strategies inside the code.

The methodology we present in our studies is useful for other applications, particularly those where you might not have the source code. LIKWID can be used in a fashion that does not require any coding instrumentation at all. In that case, the counters are collected over the entire application run, including I/O and other operations that may not or may not be of interest.

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 365–376.

[2] "The OpenMP API Specification for Parallel Programming," https://www.openmp.org/resources/openmp-compilers-tools/, Jun. 2018.

[3] K. Moreland, C. Sewell, W. Usher, L. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, M. Larsen, C.-M. Chen, R. Maynard, and B. Geveci, "VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures," *IEEE Computer Graphics and Applications (CG&A)*, vol. 36, no. 3, pp. 48–58, May/Jun. 2016.

[4] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*. Elsevier Science, 2000. [Online]. Available: https://books.google.com/books?id=pv-jXf7oGtAC

[5] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[6] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[7] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. Favre, and P. Navratil, "VisIt: An End-User Tool for Visualizing and Analyzing Very Large Data," in *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, ser. Chapman & Hall, CRC Computational Science, E. W. Bethel, H. Childs, and C. Hansen, Eds. Boca Raton, FL, USA: CRC Press/Francis–Taylor Group, Nov. 2012, pp. 357–372, lBNL-6320E. [Online]. Available: http://www.crcpress.com/product/isbn/9781439875728

[8] U. Ayachit, *The ParaView Guide*, 4th ed. Kitware, Inc., 2015, iSBN 978-1-930934-30-6.

[9] D. W. Walker, D. W. Walker, J. J. Dongarra, and J. J. Dongarra, "Mpi: A standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[10] W. Schroeder, K. M. Martin, and W. E. Lorensen, *The Visualization Toolkit (2nd Ed.): An Object-Oriented Approach to 3D Graphics*. USA: Prentice-Hall, Inc., 1998.

[11] J. Reinders, *Intel Threading Building Blocks*, 1st ed. USA: O'Reilly & Associates, Inc., 2007.

[12] Nvidia Corporation, "CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Jun. 2018.

[13] M. Larsen, J. Meredith, P. Navratil, and H. Childs, "Ray tracing within a data parallel framework," in *2015 IEEE Pacific Visualization Symposium, PacificVis 2015 - Proceedings*, ser. IEEE Pacific Visualization Symposium, S. Takahashi, S. Liu, and G. Scheuermann, Eds. United States: IEEE Computer Society, 7 2015, pp. 279–286.

[14] D. Pugmire, A. Yenpure, M. Kim, J. Kress, R. Maynard, H. Childs, and B. Hentschel, "Performance-Portable Particle Advection with VTK-m," in *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, Brno, Czech Republic, Jun. 2018, pp. 45–55.

[15] B. Lessley, T. Perciano, M. Mathai, H. Childs, and E. W. Bethel, "Maximal Clique Enumeration with Data Parallel Primitives," in *7th IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, Phoenix, AZ, USA, Oct. 2017.

[16] B. Lessley, T. Perciano, C. Heinemann, D. Camp, , H. Childs, and E. W. Bethel, "DPP-PMRF: Rethinking Optimization for a Probabilistic Graphical Model Using Data-Parallel Primitives," in *8th IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, Berlin, Germany, Oct. 2018.

[17] T. Perciano, C. Heinemann, D. Camp, B. Lessley, and E. W. Bethel, "Shared-memory parallel probabilistic graphical modeling optimization: Comparison of threads, openmp, and data-parallel primitives," in *ISC 2020, Springer LNCS*, ser. ISC 2020, Jun. 2020, (To appear).

[18] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *COMPUTER GRAPHICS*, vol. 21, no. 4, pp. 163–169, 1987.

[19] G. M. Nielson and B. Hamann, "The asymptotic decider: Resolving the ambiguity in marching cubes," in *Proceedings of the 2nd Conference on Visualization '91*, ser. VIS '91. Washington, DC, USA: IEEE Computer Society Press, 1991, p. 83–91.

[20] T. S. Newman and H. Yi, "A survey of the marching cubes algorithm," *Computers & Graphics*, vol. 30, no. 5, pp. 854 – 879,

2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0097849306001336

[21] I. Kitware, "vtkmContour Class Reference," /url-https://vtk.org/doc/nightly/html/classvtkmContour.html, last accessed March 2020.

[22] K. Moreland *et al.*, "The vtk-m user's guide, vtk-m version 1.5," Oct. 2019, http://m.vtk.org/images/c/c9/VTKmUsersGuide-1-5.pdf.

[23] A. Agranovsky, D. Camp, C. Garth, E. W. Bethel, K. I. Joy, and H. Childs, "Improved Post Hoc Flow Analysis Via Lagrangian Representations," in *Proceedings of the IEEE Symposium on Large Data Visualization and Analysis (LDAV)*, Paris, France, Nov. 2014, pp. 67–75, lBNL-6731E. Best paper award.

[24] G. Roth, G. Roth, J. Mellor-crummey, J. Mellor-crummey, K. Kennedy, K. Kennedy, R. G. Brickner, and R. G. Brickner, "Compiling stencils in high performance fortran," in *In Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM*. ACM Press, 1997, pp. 1–20.

[25] J. Donatelli *et al.*, "Camera: The center for advanced mathematics for energy research applications," *Synchrotron Radiation News*, vol. 28, no. 2, pp. 4–9, 2015.

[26] C. Sovinec, A. Glasser, T. Gianakon, D. Barnes, R. Nebel, S. Kruger, D. Schnack, S. Plimpton, A. Tarditi, and M. Chu, "Nonlinear magnetohydrodynamics simulation using high-order finite elements," *Journal of Computational Physics*, vol. 195, no. 1, pp. 355 – 386, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999103005369

[27] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[28] e. a. Thomas Gruber, "likwid-perfctr: Measuring applications' interaction with the hardware using the hardware performance counters," Jan. 2020, web location: https://github.com/RRZE-HPC/likwid/wiki/likwid-perfctr, last accessed October 2019.

[29] "Intel® VTune™Profiler User Guide: Instructions Retired Event," Feb. 2020, web location: https://software.intel.com/en-us/vtune-help-instructions-retired-event, last accessed February 2020.

[30] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design - The Hardware / Software Interface (5th Edition)*, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2014.

[31] T. Perciano, D. M. Ushizima, E. W. Bethel, Y. D. Mizrahi, D. Parkinson, and J. A. Sethian, "Reduced-complexity image segmentation under parallel markov random field formulation using graph partitioning," in *2016 IEEE International Conference on Image Processing (ICIP)*, Sept 2016, pp. 1259–1263.

[32] K. Li, "OpenMP Accelerator Support for GPUs," openMP ARB website, /urlhttps://www.openmp.org/updates/openmp-accelerator-support-gpus/, last accessed March 2020.