Title

Improving Performance of Structure-memory, Data-Intensive Applicationson Multi-core Platforms via a Space-Filling Curve Memory Layout

Authors

Bethel, E. Wes
Camp, David
Donofrio, David
et al.

Publication Date

2015-05-29

Improving Performnace of Structure-memory, Data-Intensive Applications on Multi-core Platforms via a Space-Filling Curve Memory Layout

E. Wes Bethel, David Camp, David Donofrio
Lawrence Berkeley National Laboratory, Berkeley, CA, USA

Mark Howison
Lawrence Berkeley National Laboratory, Berkeley, CA, USA
Brown University, Providence, RI, USA

February, 2015

IPDPS 2015 Workshops
International Workshop on High Performance Data Intensive Computing (HPDIC 2015)
Hyderabad, India
May 2015

Acknowledgment

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center.

Legal Disclaimer

Improving Performance of Structured-memory, Data-Intensive Applications on Multi-core Platforms via a Space-Filling Curve Memory Layout

E. Wes Bethel, David Camp, David Donofrio
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
{*ewbethel, dcamp, ddonofrio*}*@lbl.gov*

Mark Howison
Lawrence Berkeley National Laboratory and
Brown University
Providence, RI, USA
mhowison@brown.edu

***Keywords*-memory layout; space-filling curve; memory locality; shared-memory parallel; data-intensive applications; stencil operation; volume rendering**

***Abstract*—Many data-intensive algorithms—particularly in visualization, image processing, and data analysis—operate on structured data, that is, data organized in multidimensional arrays. While many of these algorithms are quite numerically intensive, by and large, their performance is limited by the cost of memory accesses. As we move towards the exascale regime of computing, one central research challenge is finding ways to minimize data movement through the memory hierarchy, particularly within a node in a shared-memory parallel setting. We study the effects that an alternative in-memory data layout format has in terms of runtime performance gains resulting from reducing the amount of data moved through the memory hierarchy. We focus the study on shared-memory parallel implementations of two algorithms common in visualization and analysis: a stencil-based convolution kernel, which uses a structured memory access pattern, and raycasting volume rendering, which uses a semi-structured memory access pattern. The question we study is to better understand to what degree an alternative memory layout, when used by these key algorithms, will result in improved runtime performance and memory system utilization. Our approach uses a layout based on a *Z*-order (Morton-order) space-filling curve data organization, and we measure and report runtime and various metrics and counters associated with memory system utilization. Our results show nearly uniform improved runtime performance and improved utilization of the memory hierarchy across varying levels of concurrency the applications we tested. This approach is complementary to other memory optimization strategies like cache blocking, but may also be more general and widely applicable to a diverse set of applications.**

***Keywords*-memory layout, data intensive algorithms, image analysis, visualization, multi-core CPUs, GPU algorithms, performance optimization**

I. Introduction

The performance of data-intensive codes, such as those found in data analysis and visualization, is often limited by memory access time. As we move towards the exascale regime, the combination of increasingly deepening and more complex memory hierarchies, combined with the increased cost of moving data through the memory hierarchy, motivates the need to optimize data movement. The primary focus of our work here is to study the degree to which changing the layout of data in memory can impact the performance of data-intensive codes in terms of runtime and memory system utilization. Our approach is to compare a traditional *array-order* (or, row-major order) layout with an alternative layout based upon a *Z*-order (a variation of the Morton-order) space-filling curve (*SFC*) layout.

Previous approaches for optimizing memory access have included cache blocking and data tiling strategies. These have worked reasonably well, though they have limitations that may preclude their use in a broad set of data intensive applications. Other previous works have studied the use of *SFC* approaches for optimizing memory access, though those have focused primarily on numerical and scientific computing methods and have an emphasis upon methods that use structured and predictable memory access patterns.

The main contributions of this work are as follows. First, we study the effects resulting from use of a *SFC* in terms of runtime performance and memory system utilization for two algorithms that are often-used in the field of visual data analysis. One algorithm, the bilateral filter, uses a structured memory access pattern, while the other, raycasting volume rendering, uses a "semi-structured" memory access pattern. The use of these two different algorithms, which have different memory access patterns that are broadly representative of many algorithms in visualization and analysis, helps provide a diversity of problems to evaluate the locality gains to be had by using a *SFC* for memory layout. Second, we conduct this study on modern architectures, the Intel MIC and IvyBridge, and report that the performance gains seen from use of *SFC* in the past in other applications continue to hold true today for two data-intensive algorithms. Third, we measure and report the locality effects gained by using a *SFC* by including a measure of memory system performance counters in our experiment. Previous studies have tended to focus on absolute runtime as a metric, while our work here includes use of memory system performance counters that serve as an indicator for quantifiably comparing the reduction in data movement resulting from a more cache-friendly layout.

Our results show improved runtime performance and improved memory system utilization when using the *Z*-order memory layout. The advantages of this approach are that its implementation is nearly transparent to the application, and it is generally beneficial to any data-intensive code that works with structured data using structured, semi-structured, or even unstructured access patterns.

II. Background and Previous Work

A. Optimizing Data Access – Blocking and Tiling

Over the years, a number of efforts have studied the issue of improving access to data through techniques that aim to improve locality of reference, both spatial and temporal. Temporal locality refers to reuse of the same datum within a short window of time, while spatial locality refers to use of data values that are nearby in memory. The basic idea is that the cost associated with accessing a datum is less when it is nearby in either space or time [1], and subsequent research efforts have focused on maximizing locality of reference.

The key idea behind *cache blocking* is to process a larger dataset iteratively by operating on a subset of it, where the subset is sized such that it fits entirely into high-speed cache, thereby maximizing locality of reference. Early research, such as Lam et al. 1991 [2], focused on deriving the optimal blocking factor size using a model that included cache line size and code characteristics.

Over the years, as memory systems have vastly increased in complexity, and cache replacement strategies are often unknown, thereby making it extremely difficult to define an accurate model for determining the optimal cache block size, the idea of *auto-tuning* has emerged as a methodology for empirically determining the optimal blocking factor, as well as other tunable algorithmic parameters, for a given code on a given platform [3]. Recognizing the increasing difficulties posed by changing and increasingly complex architectures, projects like Datta et al., 2008 [4] used auto-tuning methods to find the values for tunable algorithmic parameters, including cache block size, to achieve optimal performance for a stencil-based computation on several multi-core platforms.

The concept of *tiling* is closely related to that of blocking, in that a larger problem is decomposed into smaller ones, with the idea being that operating on smaller chunks, or tiles, that are sized appropriately will make better use of the memory hierarchy, which, in turn, will lead to improved performance. Work by Unat et al., 2013 [5] presents language-level constructs to enable tiling and parallelism, with the key observation that "each tile represents an atomic unit of work." Their work shows performance improvements of about 25% for a Navier-Stokes solver that result from better use of the memory hierarchy.

One theme common in these approaches is that the fundamental unit of work is a chunk of data: a single operation is performed on a block or tile of data. These methods aim to induce a high degree of locality in data accesses by reducing a larger problem size into a number of smaller problems, where each of the smaller problems is sized to fit into cache. Many types of problems, namely structured memory access patterns on structured data, fall into this category. Achieving the benefits of these methods requires varying degrees of code modification, from the addition of pragmas to loop unrolling and reordering. Of greatest significance is the fact that this design pattern concept, of data-based problem decomposition, does not apply in a straightforward way to many types of data-intensive codes, such as those that use other than structured memory access patterns.

B. Optimizing Data Access – Space-filling Curves

An alternative approach to achieving a high degree of locality in data accesses is to simply lay out the data in memory such that the intrinsic nature of the layout promotes a higher degree of locality.

In other words, considering an array-order layout, one fundamental problem that inhibits spatial locality is that accesses that may be nearby in *index space* may not be be spatially nearby in memory. For example, if A is a two-dimensional array of 4-byte floats having dimensions 1024×1024, then $A[i, j]$ and $A[i + 1, j]$ are adjacent in physical memory, but $A[i, j]$ and $A[i, j + 1]$ are $4K$ bytes apart in memory. Blocking and tiling strategies aim to reduce the effect of this trait by reducing a large problem into a number of smaller ones. This problem, and its adverse impact on memory performance, is exacerbated by increases in data size and dimensionality.

In contrast, *SFC* approaches lay out data in memory differently, so that an access that is nearby in index space is likely nearby in physical memory. There are a number of *SFC* approaches, including *Z*-order (also known as Morton-order curves), Hilbert-order, and others, which are presented in a comprehensive fashion in Bader, 2013 [6]. While these approaches differ in the exact way they index a subspace, they are all known for having favorable spatial locality characteristics when compared to the traditional array-order layout.

The idea of using *SFC* layouts to accelerate performance has been well studied, particularly in the area of scientific computing, and to some degree, in the visualization literature.

Pascucci and Frank, 2011 [7] compare the performance of array-order, *Z*-order, and 3D blocking (tiling) data layouts for unstructured access to structured data within the context of a remote visualization system that supports taking arbitrary slice planes through 3D blocks of data at prescribed levels of subsampling. There, the *Z*-order layout shows significant runtime performance advantages over array-order and tiling that result from the spatial locality characteristics of the *Z*-order layout. The *Z*-order layout provides the ability

to quickly load data from disk at varying levels of resolution, thereby enabling interactive exploration of large data sets.

DeFord and Kalyanaraman, 2013 [8] study the benefits to scientific computing applications of *SFC* ordering and partitioning of data in terms of its impact on communication for an implementation of the Fast Multipole Method for n-body problems. Their findings, which focus on evaluating different efficiency metrics based upon communication distance in a distributed-memory context, suggest that using a *SFC* data layout for data and task-processor assignment can result in significantly less communication compared to using an array-order layout. Bader, 2013 [6] uses *SFC* to lay out data in memory for codes performing matrix operations, and shows how traversal of the matrix elements in a cache-friendly way has runtime performance benefits.

Reissmann et al., 2014 [9] compared the performance of a matrix-matrix multiply code using array-order with *Z*-order and Hilbert-order *SFC* layouts. They report runtime performance and power cost estimates in each configuration, and find that the *Z*-order layout offers performance and power use advantages in many configurations, but that the cost of computing the Hilbert-order curve indexing, which is more complex than that of the *Z*-order method, exceeds benefits that might be gained by locality effects.

Bethel and Howison, 2012 [10] showed that the *Z*-order in-memory layout results in better runtime performance and better use of the memory hierarchy, in terms of L2 cache misses on an NVIDIA Fermi GPU, for a raytracing volume renderer. The work we describe here goes into more detail to show how the *Z*-order layout is less sensitive to performance degradation caused by "against-the-grain", or non-optimal memory access patterns, that result in substantial performance variation when using array-order memory layout, as well as to study these effects in more detail, and on current platforms. This feature is borne out quite clearly by the results shown in Section IV-D, where the methodology explicitly tests this idea.

III. Implementation

We implement and test two different shared-memory parallel algorithms as part of this study. The 3D bilateral filter (Section III-A) uses structured memory access patterns, while the raycasting volume renderer (Section III-B) uses a semi-structured memory access pattern. Both algorithms operate on a single block of 3D data, where each accesses via an interface that encapsulates the *Z*-order or array-order indexing in a way transparent to the application (Section III-C).

Both implementations use C/C++ and POSIX threads for shared-memory parallelism. There are two primary reasons for using POSIX threads rather than a compiler-assisted approach, like OpenMP. First, the raycasting implementation implements multiple types of work assignment strategies. The best-performing strategy is a dynamic approach implemented using a worker-pool model, which doesn't lend itself to automatic loop parallelization characteristic of compiler-assisted approaches. Second, the MIC platform, at this point in time, provides certain capabilities for thread management that are accessible only through the POSIX threads interface.

A. Structured Memory Access: 3D Bilateral Filtering

The bilateral filter is an anisotropic, edge-preserving image smoothing method first introduced for 2D images by Tomasi and Manduchi, 1998 [11] (for a detailed introduction and comparison to other image smoothing methods, see Howison and Bethel, 2014 [12]). It can be extended to 3D volumes, and is essentially a two-stage operation involving first an $N \times N \times N$ Gaussian convolution kernel followed by a normalization step over the entire stencil of filtered values to take into account data-dependent characteristics. This type of stencil-based access pattern forms the basis of many algorithms in data analysis and visualization where a final answer is computed as a function of neighboring data values.

The output at each image pixel/voxel $D(i)$ is the weighted average of the influence of nearby image pixels/voxels $\bar{i}$ from the source image S at location i. The "influence" is computed as the product of a geometric spatial component $g(i,\bar{i})$ and signal difference $c(i,\bar{i})$.

$$D(i) = \frac{1}{k(i)} \sum g(i,\bar{i}) c(i,\bar{i}) \tag{1}$$

where $k(i)$ is a normalization factor that is the sum of all weights $g(i,\bar{i})$ and $c(i,\bar{i})$, computed as:

$$k(i) = \frac{1}{\sum g(i,\bar{i}) c(i,\bar{i})} \tag{2}$$

While it is possible to precompute the portions of $k(i)$ contributed by $g(i,\bar{i})$, which depend only on the 3D Gaussian filter weights, the set of contributions from $c(i,\bar{i})$ are not known *a priori* as they depend upon the actual set of photometric differences observed across the neighborhood of $c(i,\bar{i})$ and will vary depending upon the source image contents and target location i. For this reason, the bilateral filter formulation is more computationally intensive than a simple convolution kernel.

Tomasi and Manduchi define g and c to be Gaussian functions that attenuate the influence of nearby points such that those nearby in geometric or signal space have greater influence, while those further away in geometric or signal space have less influence according to a Gaussian distribution. So,

$$g(i,\bar{i}) = e^{-\frac{1}{2}\left(\frac{d(i,\bar{i})}{\sigma}\right)^2} \tag{3}$$

Here, $d(i,\bar{i})$ is the spatial distance between a pixel i and nearby pixels in the neighborhood $\bar{i}$. With the Gaussian

filter weights, the idea is to blend values of pixels in the neighborhood such that those closer to pixel i have a greater contribution, while those further away make less of a contribution. The constant σ is a user-definable parameter defining the standard deviation of the distribution. The impact of this parameter is as follows: larger values result in a greater degree of smoothing; smaller values produce less smoothing.

The photometric similarity influence weight $c(i, \bar{i})$ uses a similar formulation, but $d(i, \bar{i})$ would be replaced with the absolute difference $\|S(i) - S(\bar{i})\|$ between the value of the source pixel $S(i)$ and the value of the nearby pixel $S(\bar{i})$. Here, the idea that nearby pixels having a value that is "close to" pixel i will have more influence, while those with a value "far from" pixel i will have less influence. It is this particular trait that makes the bilateral filter an "edge preserving" filter. This machinery is generally applicable to other problems, such as edge-preserving mesh smoothing [13].

This type of computation can be done in "embarrassingly parallel" fashion: the computation of each output pixel/voxel $D(i)$ is completely independent from every other output pixel/voxel. We approach parallelizing the problem by assigning a "pencil" of output voxels—from a width-, height-, or depth-row from the volume—to each thread, with work being handed out in round-robin fashion to all threads.

The choice of width, height, or depth row assignment of voxels to threads is significant, and can have a noticeable impact on performance. For example, Bethel, 2012 [14] found that doing a depth-row assignment resulted in a two-fold performance gain over width- and height-row assignments when run on a GPU. This performance gain resulted from the benefits associated with coalesced memory accesses on the GPU; the depth-row method generated coalesced memory access patterns, whereas the others did not. We see similar performance differences of this type later in Section IV.

B. Semi-structured Memory Access: Raycasting Volume Rendering

Volume rendering is a common technique for displaying 2D projections of 3D sampled data [15], [16] and is computationally, memory, and data I/O intensive.

Raycasting volume rendering is an *image-order* method in which the outer loop iterates over image pixels, casting rays from the viewpoint through the image plane, and computing the color of each output pixel by sampling the data from the 3D volume intersected by the ray and compositing those samples into a final shade. In contrast, *object-order* methods iterate over input voxels, and through a process akin to rasterization, compute the colors for each of the output pixels covered by the input voxel. Over the years, there has been a great deal of research in the are of parallelizing volume visualization methods (see Kaufman and Mueller, 2005 [17] for an overview). Parallelization methods differ according to whether the algorithm is image-order or object order.

Our implementation of a parallel raycasting volume renderer uses thread-based shared-memory parallelism within a node, and MPI-based, distributed-memory parallelism for use at large scale [18]. Since our study here is on intra-node memory utilization and its relationship to runtime, we are using only on the shared-memory parallel portion of this code. The problem is parallelized by dividing the output image into tiles, then assigning a thread to compute the final output pixel values for each tile. Bethel and Howison, 2012 [10] found that the choice of tile size can have a profound impact on the runtime of this algorithm on both multi-core CPUs and many-core GPUs. For our tests in this work, we use a tile size of 32×32 pixels, as that size had consistently good performance across a diversity of platforms in our previous studies.

When using orthographic projection, all the rays are parallel, and as such, they all traverse through the 3D block of data in the same way. In contrast, with perspective projection, which is what we are using here, the rays diverge from one another along their length as one moves away from the eyepoint. Along each ray, the memory access pattern is consistent and predictable, but each ray in perspective projection will use a slightly different traversal through memory.

More specifically, if the slope of a ray's path is $(\delta x, \delta y, \delta z)$, then in orthographic projection, all rays will have that same slope. In perspective projection, each ray will have a unique $(\delta x, \delta y, \delta z)$ slope. Therefore, in perspective projection, each ray uses a memory access pattern that is distinct and different from all other rays. For this reason, we refer to this class of algorithm as using a *semi-structured* memory access pattern.

In the case of an array-order memory layout, we assume that if a ray's path is parallel to the *x-axis* of the 3D volume, and has a slope of the form $(\delta x, \sigma y, \sigma z)$, where $\sigma y, \sigma z \approx 0$, that the integration along that ray's path will result in a more favorable memory access pattern. As data samples are taken along the ray's length, additional sample along the ray will access a data value that is located approximately $(\delta x, \sigma y, \sigma z)$ from the previously taken sample. Such a sampling pattern results in contiguous samples being taken from nearby locations in physical memory.

Similarly, we assume that when the ray's path does not closely follow the most-quickly varying index direction of the array-order layout, e.g., where its slope more closely resembles $(\sigma x, \delta y, \delta z)$ and $\sigma x \approx 0$, that contiguous ray samples will be accessing locations that, while nearby in index space, will be significantly more distant in physical memory when compared to the $(\delta x, \sigma y, \sigma z)$ access pattern.

It is this very problem we are attempting to solve by using a *Z*-order layout. The hypothesis we wish to test is whether or not replacing the array-order layout with a more cache friendly *Z*-order layout will result in contiguous ray samples accessing data that is located nearby in physical memory,

regardless of the ray's slope.

C. Accessing Memory

Our intention is to streamline and unify access to memory, regardless of whether using *Z*-order and array-order layouts, for use by each of the above algorithm implementations. Conceptually, each of the above algorithms will access memory using something like $data = A[i, j, k]$, though the computation of i, j, k varies as a function of algorithm.

Our implementation is a lightweight library that implements both *Z*-order and array-order indexing in a way that puts the indexing computation cost on more or less equal footing. During an initialization call, the library constructs some static tables, which are then referenced during the indexing calculation. The tables themselves are of modest size, as follows using a 3D structured dataset of size 512^3 as an example.

For array-order indexing, we construct two tables of `off_t` values: a *yoffset* table that is $ysize = 512$ in length, and a *zoffset* table that is $zsize = 512$ in length. Each j'th entry of the *yoffset* table contains the value $j * xsize$, and each k'th entry of the *zoffset* table contains the value $k * xsize * ysize$. Computing an array-order index for some arbitrary (i, j, k) location requires two table lookups and two additions.

For *Z*-order indexing, we use a similar approach to that described in Pascucci and Frank, 2001 [7]: we construct three tables of length $max(xsize, ysize, zsize)$ in length, then each i'th entry of each table contains precomputed bit-shifted values for the *Z*-order index. Computing a *Z*-order index for some arbitrary (i, j, k) location requires three table lookups and two OR operations.

Conceptually, this difference is transparent to the application: after some one-time initialization, during which time the static indexing tables are constructed, the application would simply make a *getIndex(i,j,k)* call and the library returns the array-order or *Z*-order index. We used this approach because it is a very efficient way to construct the *Z*-order indices, and to put the array-order and *Z*-order index calculations on more or less equal footing so that in the performance results, the cost of index computation is essentially equal, and the differences in performance will reflect the gains due to memory layout.

IV. Methodology and Results

A. Test Platforms

We ran our study on two different platforms located at the National Energy Research Scientific Computing (NERSC) facility:

`edison.nersc.gov` is a Cray XC30 system comprised of 5,576 compute nodes, each of which is comprised of two 2.4GHz Intel Ivy Bridge processors, twelve cores each, and having 64GB of DDR 1300MHz memory. On this processor, each core has its own 64KB L1 and 256KB L2 cache, and all cores share a single 30MB L3 cache.

`babbage.nersc.gov` is a testbed system containing one login and 45 compute nodes, each containing two 2.6GHz, 8-core Intel Sandy Bridge processors, and two 60-core Intel MIC/Knight's Corner 5100P accelerators having 8GB of GDDR5 memory.

On both platforms, we make use of PAPI [19] to collect a variety of hardware performance counters.

B. Methodology

Our hypothesis is that the *Z*-order memory layout will result in better runtime due to better use of the memory hierarchy. In our case, we are defining "better use" as increased cache hit rates, which will allow the kernel to avoid the long-latency (100's of cycles) main-memory access.

1) Performance Counters: After looking at several different PAPI metrics on our two test platforms, and considering realistic problem sizes, we found that:

Ivy Bridge. The PAPI_L3_TCA metric is closely correlated to relative performance: increases/decreases in runtime are generally reflected as increases/decreases in PAPI_L3_TCA counts. The idea is that an L3 cache access occurs because a memory request can't be satisfied by L1 or L2 cache. So, when PAPI_L3_TCA is lower, then the memory access requests are being satisfied by L1 or L2 cache.

Intel MIC. The L2_DATA_READ_MISS_MEM_FILL is a proxy for memory bandwidth. The Intel MIC has two levels of caching, as opposed to three in Ivy Bridge, and to test our theory that *Z*-order performance is closely related to LLC cache utilization, we wish to examine the relative memory bandwidth between the two kernels as a way to quantify how effectively each kernel uses the memory hierarchy. We found that increases/decreases in runtime are reflected by increases/decreases in L2 read miss numbers.

2) Scaled, Relative Differences: The numbers we report in most of the figures that follow are "scaled, relative differences." If we let a represent the measurement for the array-order code and z represent the measurement for the *Z*-order code, then scaled relative difference, d_s, is computed as:

$$d_s = (a - z)/z \tag{4}$$

The result of this computation is that when d_s is less than zero, the measurement for a is less than that for z. Conversely, when d_s is greater than zero, then the measurement for a is greater than z.

We use a scaling factor, namely z, to normalize the results, since some measures, like runtime, are a few 1000s milliseconds, whereas some of the other performance counters can generate numbers into the 10s of millions. The scaling factor makes it easier to compare the relative (scaled) difference

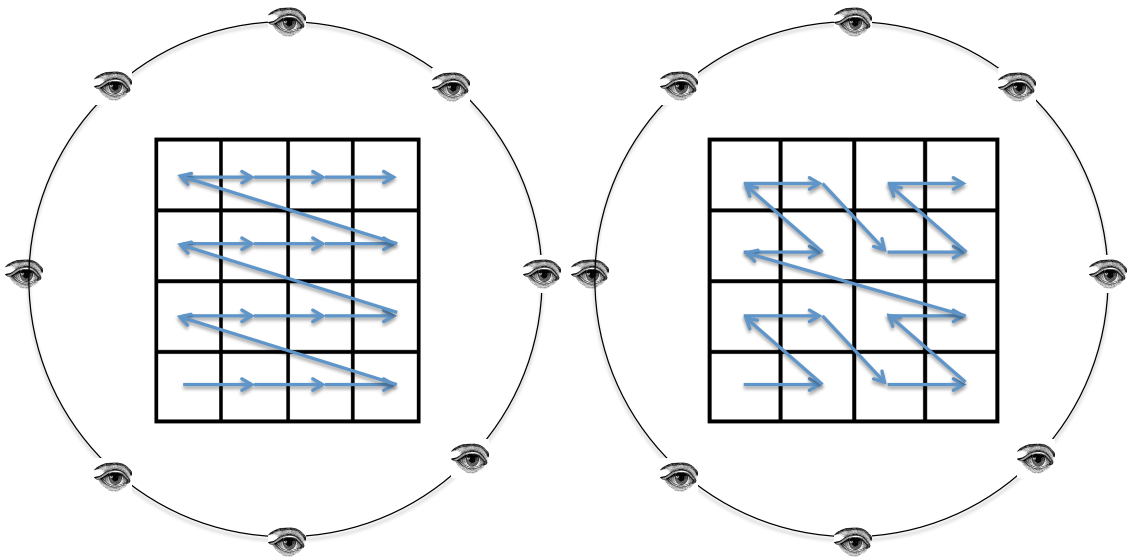

Figure 1: In the case of the array-order layout (left), the rays from some viewpoints align well with memory layout, while in others, the rays align poorly with memory layout. In contrast, with the *Z*-order layout (right), rays from all viewpoints do not have particularly unfavorable alignment.

between different measures, like runtime and cache access counters.

The d_s is similar to, but not exactly the same as, a percentage. For example, a value of 0.1 means, for example, that there is a 10% difference between a and z; a value of 1.0 means there is 100% difference between a and z; and a value of 10.0 means there is a 1000% difference between a and z. It is an effective way to measure the difference between a and z in a normalized fashion.

3) Bilateral Filter: We test this algorithm using as input a dataset, the source of which is an MRI instrument at the University of California–Davis, that is 512^3 in size and that consists of 4-byte floating point values. We vary the following three parameters during testing.

Stencil size, from a smaller $3\times3\times3$ to a larger $11\times11\times11$. These different stencil sizes are reflected in the results as $r1$, $r3$, and $r5$, which correspond to 3^3, 5^3, and 11^3 stencil sizes, respectively.

Concurrency, which varies by platform (see Section IV-B5).

Voxel-row assignment, and stencil processing order. We test using both width- and depth-row voxel assignments (see Section III-A), as well as stencil iteration order. The idea with stencil iteration order is to change the way that the loops are structured so that an *xyz* order means that the innermost loop iterates over x, the most quickly varying order in memory (in the array-order sense), then over y, the next most quickly varying order, then over z. In contrast, *zyx* order iterates over z in the innermost loop, which is the least favorable for array-order layouts. The intention is to purposefully induce a potentially unfavorably memory access pattern into the stencil code.

4) Raycasting Volume Rendering: We test this algorithm using as input a dataset, the source of which is a combustion simulation code, that is 512^3 in size and that consists of 4-byte floating point values. We vary the following two parameters during testing:

Viewpoint. In these tests, we vary the viewpoint by orbiting around the center of the dataset over the course of a test run; each individual run and the associated performance data corresponds to a single viewpoint. Our intention is to vary the data access pattern such that in some views, the rays' data access pattern align well with memory layout (in the array-order sense), while in other views, the rays' data access pattern will align poorly with memory. Figure 1 illustrates this idea with a 2D example.

Concurrency, which varies by platform (see Section IV-B5).

5) Concurrency: For both algorithms, we use the same set of varying-concurrency levels across the test battery.

On the Ivy Bridge platform, where there are two 12-core processors, we vary concurrency over $\{2, 4, 6, 8, 10, 12, 18, 24\}$ threads. This platform supports alternative ways of mapping threads to cores, and we used the "compact" method for these tests. In this way, when running up to 12 threads, they are all placed on the same processor.

On the MIC platform, where there are 60 cores per MIC card, we vary concurrency over $\{59, 118, 177, 236\}$ threads. According to our system documentation, one core is needed to run O/S and other functions, so we use the remaining

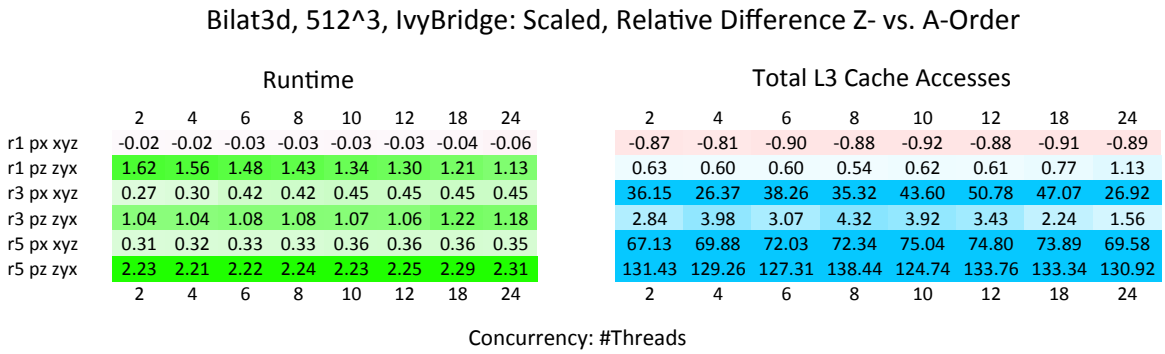

Figure 2: A comparison of the differences in runtime (left) and total L3 cache accesses (right) for the bilateral3d code on the Ivy Bridge platform.

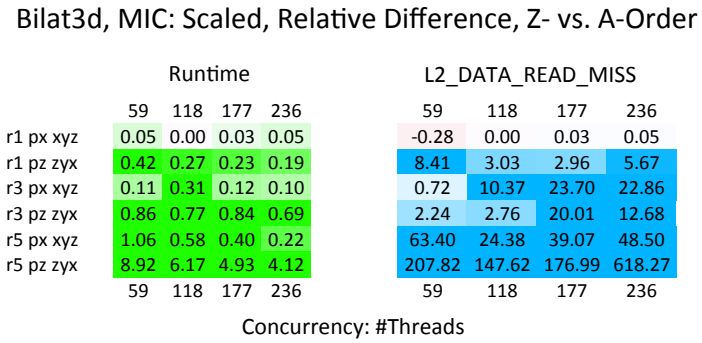

Figure 3: A comparison of the differences in runtime (left) and L2_DATA_READ_MISS_MEM_FILL (right) for the bilateral3d code on the MIC platform.

59 cores for our application. The four different concurrency levels reflect each core's ability to accommodate up to 4 threads each.

C. Bilateral Filter—Results

Fig. 2 shows scaled, relative differences in runtime (left) and PAPI_L3_TCA (right) for the bilateral3d code on the Ivy Bridge platform. Positive numbers, shaded in hues of green (runtime) and blue (L3 cache accesses) indicates in which configurations the Z-order code is running faster and has fewer total L3 cache accesses compared to the array-order code. The rows are individual tests, where we use three different stencil sizes, 3^3 ($r1$), 5^3 ($r3$), and 11^3 ($r5$), vary the assignment of voxel pencils (px vs. pz) to threads, and vary the stencil processing order (xyz vs. zyx).

Except for the smallest stencil size ($r1$) and the stencil processing order most favorable to array-order (px, xyz), the Z-order code shows better runtime, ranging from about 27% difference to about a 231% difference. Except for the $r1, px, xyz$ configuration, differences in PAPI_L3_TCA rates are lower for Z-order than array-order from as little about half an order of magnitude up to over two orders of magnitude. In the cases where we see the relatively large difference in PAPI_L3_TCA, we generally see the biggest improvements in runtime. Some of these differences are magnified by the relative comparison and show how much more effectively (i.e. higher hit rates) the L2 cache is being used in the Z-order case.

We observe one unusual feature in this data, namely the $r3$, PAPI_L3_TCA data. In the $r1$ and $r5$ cases, there is a substantial decrease in PAPI_L3_TCA count going from the px to pz configuration. In the $r3$ case, the converse is true: runtime decreases but PAPI_L3_TCA increases. Even so, it is the case that the Z-order code shows uniformly better performance for both runtime and PAPI_L3_TCA counts compared to the array-order implementation. We continue to investigate other performance counter data to refine our understanding of the correlation between runtime and memory system utilization.

Test results from the MIC platform, shown in Fig. 3, indicate that the Z-order code runs faster than the array-order code in all but one configuration. The L2_DATA_READ_MISS_MEM_FILL plot shows similarities to that for Ivy Bridge: the difference in memory utilization grows substantially with both increase stencil

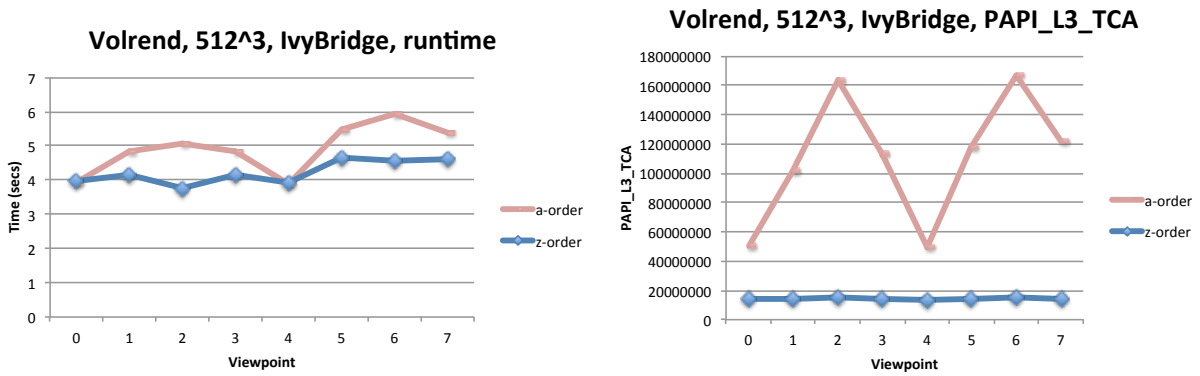

Figure 4: Comparing array-order and *Z*-order runtime (left) and PAPI_L3_TCA (right) performance on the Ivy Bridge platform for one specific volume rendering test configuration. The array-order code's performance is best when the viewpoint rays are well aligned with memory, at viewpoints 0 and 4, but falls off as the viewpoint rays become increasingly misaligned with memory. In contrast, the performance of the *Z*-order implementation appears to be uncorrelated with viewpoint, indicating better utilization of the memory system.

Volrend, 512^3, IvyBridge: Relative Z- vs. A-Order

Runtime

Viewpoint \ Concurrency: #Threads	2	4	6	8	10	12	18	24
0	-0.01	-0.01	0.00	0.00	0.01	0.02	0.02	0.05
1	0.17	0.19	0.18	0.23	0.18	0.19	0.16	0.23
2	0.34	0.33	0.34	0.32	0.33	0.34	0.33	0.33
3	0.17	0.21	0.18	0.24	0.18	0.20	0.17	0.23
4	-0.01	-0.01	-0.01	0.00	0.01	0.00	0.04	0.05
5	0.17	0.16	0.17	0.14	0.16	0.15	0.16	0.16
6	0.30	0.30	0.30	0.29	0.29	0.30	0.31	0.30
7	0.16	0.15	0.16	0.13	0.16	0.15	0.15	0.13
	2	4	6	8	10	12	18	24

Total L3 Cache Accesses

Viewpoint \ Concurrency: #Threads	2	4	6	8	10	12	18	24
0	0.85	0.84	0.79	0.83	0.86	0.84	0.88	0.89
1	2.72	3.27	3.18	3.58	3.17	3.30	2.79	3.66
2	3.99	3.40	3.87	3.18	3.80	3.51	3.76	2.96
3	3.15	3.93	3.40	4.42	3.53	3.79	3.29	4.27
4	0.83	0.82	0.84	0.82	0.93	0.89	1.03	0.92
5	3.30	3.86	3.46	4.08	3.35	3.51	2.99	3.65
6	4.22	3.81	3.85	2.98	3.94	3.56	4.35	2.73
7	3.35	2.94	3.01	3.33	2.97	2.99	2.94	3.59
	2	4	6	8	10	12	18	24

Figure 5: A comparison of the differences in runtime (left) and PAPI_L3_TCA (right) for the volume rendering code on the Ivy Bridge platform. As with Fig. 4, the array-order code's performance is at its best at viewpoints 0 and 4, where the viewpoint rays are aligned well with memory.

size and varying voxel-row assignment as well as stencil processing order. The combination of large stencil size and processing order reveals the distinct advantages of the *Z*-order layout.

D. Raycasting Volume Renderer—Results

As an indicator of the effect of how moving the viewpoint from a location where rays are well aligned with array-order memory order to one where they are poorly aligned, Fig. 4 shows both runtime and PAPI_L3_TCA for one particular test configuration on the Ivy Bridge platform. Here, we see that the array-order runs fastest when the view is positioned such that rays are well aligned with array-order memory layout, at viewpoints 0 and 4. We also see that runtime increases as the rays become increasingly unaligned with memory layout. The impact on memory system utilization of this misalignment is clearly visible in the PAPI_L3_TCA chart in Fig. 4. In contrast, the *Z*-order runtime and memory system performance is much less, if at all, impacted by viewpoint.

Looking at all test results, Fig. 5 shows a comparison of the differences in runtime (left) and PAPI_L3_TCA (right) for the volume rendering code on the Ivy Bridge platform. Positive numbers, shaded in hues of green (runtime) and blue (PAPI_L3_TCA) indicates in which configurations the *Z*-order code is running faster and has fewer total L3 cache accesses compared to the array-order code. Each row corresponds to a different viewpoint, where we orbit the viewpoint about the origin from 8 different locations.

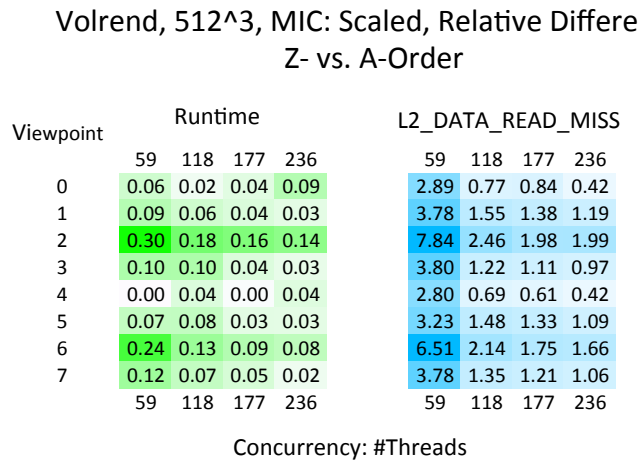

Figure 6: A comparison of the differences in runtime (left) and L2_DATA_READ_MISS_MEM_FILL (right) for the volume rendering code on the MIC platform.

Rows 0 and 4 correspond to viewpoints where the rays are parallel to the X axis, or in other words, that each ray's data access pattern is well aligned with memory. In these cases, we see that *Z*-order and array-order have very similar runtimes, though the *Z*-order configuration has more favorable PAPI_L3_TCA performance.

As the rays begin to fall off the X axis, and the consequently the memory access pattern becomes increasingly unaligned, we see that the *Z*-order layout results in substantially better runtimes, ranging from about 13% to about 34%. We observe a correspondence between lower runtimes and improvements in utilization of the memory hierarchy as evidenced by lower PAPI_L3_TCA counts.

Results from the runs on the MIC platform are shown in Fig. 6. Here, we see a similar pattern, where the difference in runtime between *Z*-order and array-order is less at viewpoints 0 and 4, where the rays' access pattern is more closely aligned with memory. As the viewpoint moves away from those positions, the difference becomes more pronounced.

For the L2_DATA_READ_MISS_MEM_FILL metric, we see that the *Z*-order code has uniformly better numbers, which indicates the *Z*-order code's memory reads are much more frequently satisfied by values resident in L2 cache when compared to the array-order code. We also see that this metric is highest for the 59-thread configuration; as we increase the number of threads per core, this metric drops. This effect is most likely caused by the lower likelihood that threads mapped onto a core are all accessing the same region of data, thereby reducing the amount of spatial locality (effectiveness of the LLC). This effect is more pronounced on the Intel MIC vs. Ivy Bridge as the L2 Cache on the MIC (512KB) is significantly smaller than the Ivy Bridge's 30MB shared LLC. Further investigation into the relationship between per-thread tile size and the resulting performance would help refine understanding in this area.

V. Conclusion

The main focus of our study has been to study the impacts on performance, as measured by runtime and memory system utilization, that result when using a *SFC* layout for two data-intensive algorithms common in visualization on modern platforms. For the two algorithms we study, the results suggest there are indeed performance gains to be had with this approach: codes run faster in most circumstances, and also make better use of the memory hierarchy. In the long run, finding ways to minimize data movement is of high priority due to the increased cost of accessing more distant locations in the memory hierarchy.

Our study focuses on comparing two metrics: absolute runtime, and one measure of memory system utilization, which varies by platform. The memory system metrics we show here, L2_DATA_READ_MISS_MEM_FILL on the MIC and PAPI_L3_TCA on the IvyBridge, are a coarse measure of memory system utilization. Even as a coarse metric, it is a useful leading indicator of the correlation between runtime and memory system utilization. There are a few individual test cases where the correlation is less clear. This result suggests that additional metrics, of which there are dozens on each platform, will help to refine our understanding of the relationship between runtime and memory system utilization.

While this approach does appear promising, it does have limitations. One significant limitation is the fact that *SFC* approaches, which are based upon the concept of recursive subdivision of space, work best when data is sized to be an even power of 2 along each axis. While the *SFC* indexing scheme can, in principle, accommodate data sizes that are not an even power of two in size, doing so requires an the

data be stored in a buffer that is an even power of two in size, which is then indexed by the *SFC*. Worthwhile future work would explore ways to overcome this limitation. Another limitation is that while it is readily applicable to structured data, it is unlikely as readily applicable to unstructured data.

Our study here focuses on two specific algorithms common in visualization and analysis. Additional studies on a broader set of algorithms from visualization and analysis will help further reveal benefits and limitations of this approach.

ACKNOWLEDGMENT

This work was supported by the Director, Office of Science, Office and Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] P. J. Denning, "The Locality Principle," *Commun. ACM*, vol. 48, no. 7, pp. 19–24, Jul. 2005. [Online]. Available: http://doi.acm.org/10.1145/1070838.1070856

[2] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: ACM, 1991, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/106972.106981

[3] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated Empirical Optimization of Software and the ATLAS Project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, 2001.

[4] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil Computation Optimization and Autotuning on State-of-the-art Multicore Architectures," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 4:1–4:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413375

[5] D. Unat, C. Chan, W. Zhang, J. Bell, and J. Shalf, "Tiling as a Durable Abstraction for Parallelism and Data Locality," in *Proceedings of Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, Nov. 2013.

[6] M. Bader, *Space-Filling Curves - An Introduction with Applications in Scientific Computing*, ser. Texts in Computational Science and Engineering. Springer-Verlag, 2013, vol. 9. [Online]. Available: http://link.springer.com/book/10.1007/978-3-642-31046-1/page/1

[7] V. Pascucci and R. J. Frank, "Global Static Indexing for Real-time Exploration of Very Large Regular Grids," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '01. New York, NY, USA: ACM, 2001, pp. 2–2. [Online]. Available: http://doi.acm.org/10.1145/582034.582036

[8] D. DeFord and A. Kalyanaraman, "Emperical Analysis of Space-Filling Curves for Scientific Computing Applications," in *42nd International Conference on Parallel Processing (ICPP)*, Oct. 2013, pp. 170–179.

[9] N. Reissmann, J. C. Meyer, and M. Jahre, "A Study of Energy and Locality Effects using Space-filling Curves," in *Proceedings of the 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014) and IPDPS 2014 Workshops (IPDPSW 2014)*, May 2014.

[10] E. W. Bethel and M. Howison, "Multi-core and Many-core Shared-memory Parallel Raycasting Volume Rendering Optimization and Tuning," *International Journal of High Performance Computing Applications*, vol. 26, no. 4, pp. 399–412, Nov. 2012.

[11] C. Tomasi and R. Manduchi, "Bilateral Filtering for Gray and Color Images," in *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*. Washington, DC, USA: IEEE Computer Society, 1998, p. 839.

[12] M. Howison and E. W. Bethel, "GPU-accelerated denoising of 3D magnetic resonance images," *Journal of Real-Time Image Processing*, 2014. [Online]. Available: http://dx.doi.org/10.1007/s11554-014-0436-8

[13] T. R. Jones, F. Durand, and M. Desbrun, "Non-iterative, feature-preserving mesh smoothing," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 943–949, Jul. 2003. [Online]. Available: http://doi.acm.org/10.1145/882262.882367

[14] E. W. Bethel, "Exploration of Optimization Options for Increasing Performance of a GPU Implementation of a Three-Dimensional Bilateral Filter," Lawrence Berkeley National Laboratory, Berkeley, CA, USA, 94720, Tech. Rep. LBNL-5406E, 2012.

[15] M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 29–37, May 1988.

[16] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," *SIGGRAPH Computer Graphics*, vol. 22, no. 4, pp. 65–74, 1988.

[17] A. Kaufman and K. Mueller, "Overview of Volume Rendering," in *The Visualization Handbook*, C. D. Hansen and C. R. Johnson, Eds. Elsevier, 2005, pp. 127–174.

[18] 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*, vol. 18, no. 1, pp. 17–29, Jan. 2012, lBNL-4370E.

[19] U. of Tennessee Knoxville Innovative Computer Lab (ICL), "Performance Application Programming Interface (PAPI)," last accessed: October 2014. [Online]. Available: http://icl.cs.utk.edu/papi