zorder-lib: Library API for Z-Order Memory Layout

E. Wes Bethel

Lawrence Berkeley National Laboratory
Berkeley, CA, USA, 94720

April, 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, through the grant "Towards Exasacle: High Performance Visualization and Analytics," program manager Dr. Lucy Nowell. This research used resources of the National Energy Research Scientific Computing Center.

Legal Disclaimer

zorder-lib: Library API for Z-Order Memory Layout

E. Wes Bethel

Lawrence Berkeley National Laboratory
v1.0, April 2015

1 Introduction

This document describes the motivation for, elements of, and use of the *zorder-lib*, a library API that implements organization of and access to data in memory using either *a-order* (also known as "row-major" order) or *z-order* memory layouts.

The primary motivation for this work is to improve the performance of many types of data-intensive codes by increasing both spatial and temporal locality of memory accesses. The basic idea is that the cost associated with accessing a datum is less when it is nearby in either space or time [3].

In *a-order* layout, data is laid out in memory in a contiguous fashion. When the data array is multidimensional (e.g., 2D arrays in the case of images, 3D arrays in the case of volumes, etc.) 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. We refer to this as the *index-space locality problem*.

Previous work in increasing locality includes *blocking* and *tiling* techniques, both of which aim to break a larger problem into a number of smaller problems that fit into high-speed, on-chip cache memory. Early research, such as Lam et al. 1991 [4], focused on deriving the optimal blocking factor size using a model that included cache line size and code characteristics. In these approaches, data remains organized in *a-order* layout, but if the blocking or tiling strategy is working properly, because it fits into cache, the cost of the *index-space locality problem* is presumably negligible. These approaches require modification to the underlying code to implement the blocking or tiling strategy, and are therefore intrusive.

An alternative approach for increasing locality is to use a *space-filling curve* (*SFC*) layout. *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 [1]. 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 *a-order* layout. Fig. 1 illustrates the difference between *a-order* and *z-order* layouts.

In this implementation, we are using the *z-order* layout, rather than one of the other *SFC* layouts (e.g., Hilbert curve) due to the fact that *z-order* indices are relatively inexpensive to compute.

How much performance can be gained by using a *z-order* layout? The answer varies, depending upon a code's memory access pattern and the underlying platform. Bethel et al. 2015 [2] explore

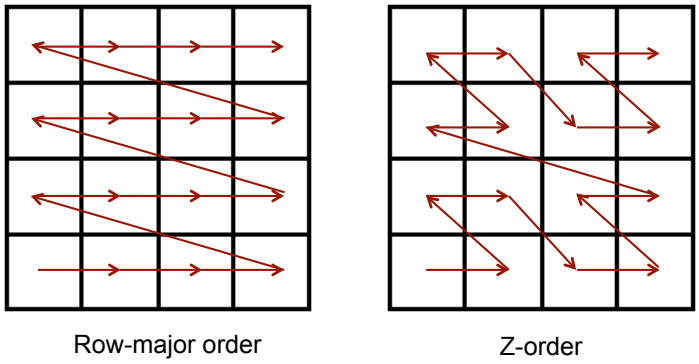

Figure 1: This illustration shows the difference between *a-order* and *z-order* memory layout formats. This *SFC* layout uses the *Z-order* memory layout, which is a variation of the Morton-order space-filling curve.

this very question, studying two different algorithms (raycasting volume rendering and a bilateral smoothing filter, which is based upon a 3D convolution kernel of user-defined size) on two different platforms (Intel Ivy Bridge and Intel Knight's Corner) and find that in nearly all cases, the *z-order* layout results in faster runtime and better utilization of the memory hierarchy in the form of better use of the cache hierarchy. The absolute amount varies, but ranges from nearly equal (*a-order* is a small bit faster in a very few cases) to situations where the *z-order* code is upwards of 800% faster in a few cases.

2 *zorder-lib* Implementation and Use

2.1 Overview

zorder-lib is a C-language library callable from C or C++ programs. It is best suited for use with applications that will access data stored in 2D or 3D arrays using structured, semi-structured, or unstructured (i.e., unpredictable) memory access patterns.

Typical use will consist of first performing some one-time initialization (Sec. 2.2), data conversion from *a-order* to *z-order* format (Sec. 2.3), then accessing array elements from memory to perform the desired computations (Sec 2.4).

2.2 Initialization

Initialization of the library consists of invoking the routine `zoInitZOrderIndexing` specifying input parameters that indicate the (i, j, k) size of the array that will be used, along with a memory layout enumerator `ZORDER_LAYOUT` or `AORDER_LAYOUT`.

This routine returns a pointer to a `ZOrderStruct`, which the application will use for the remainder of the run to access data items in the array.

The routine `zoInitZOrderIndexing` performs various initialization steps, which consist of precomputing some relatively small-sized tables, which are part of the `ZOrderStruct`, and that are used to accelerate the computation of both *a-order* and *z-order* indices later on.

The memory layout enumerator, `ZORDER_LAYOUT` or `AORDER_LAYOUT`, is referenced only when using the generic index access operator, which is described in Sec. 2.4.1.

A word of warning: the *z-order* layout method has an intrinsic limitation, which is that it really wants to work only with data that is an even power of two in size in each dimension. This *zorder-lib* implementation will function with data that is not an even power of two in size, but at a cost, which will be described in Sec. 2.3 and Sec. 3.1.

2.3 Data Conversion

Assuming your application has a multidimensional array of data, which is laid out in *a-order* format, then you will need to convert it to *z-order* with a call to the routine `zoConvertArrayOrderToZOrder`.

That routine takes as input a `ZOrderStruct` created by `zoInitZOrderIndexing`, along with a pointer to your input data, let's call it `A`, cast to a `void *`, as well as information about the size of each array element element. For example, if your input array consists of doubles, then you would specify the size of each element as `sizeof(double)`.

The routine will return a pointer to a new array, also cast to a `void *`, which contains the contents of your input array `A` rearranged into *z-order* layout in the new array, which we'll call `Z`.

Several points of warning:

1. No zero-copy. In this implementation, we make a copy of your array `A` and put it into `Z`, but rearrange `A` in the process so that it is in *z-order* format. There is no workaround for this situation, making a copy of the input data, unless you are able to have your data in `A` in *z-order* layout to begin with. In which case, you would not need to call `zoConvertArrayOrderToZOrder` to rearrange it.

2. Potential for significant inflation of memory footprint for uneven power-of-two sizes. **This warning cannot be overstated.** When the input data is an even power of two in size in each dimension, there will be no inflation. When the input data is not an even power of two in size, there will be an inflation of the memory footprint such that the size of `Z` will be larger than the size of `A`, sometimes by a lot. This limitation is explained in more detail in Sec. 3.1.

2.4 Accessing Data

Once data has been converted from *a-order* to *z-order*, the application will access data by a two-stage process:

1. First, obtain the index of some (i, j, k) location in an array, which may be laid out using *a-order* or *z-order* layout approaches. You will use the *zorder-lib* library routine(s) to obtain this index.

2. Second, access the datum directly using the index provided by the first step.

The process of obtaining an index at (i, j, k) location can be done using one of two approaches, either the "generic" method, or one of two layout-specific methods.

Note that the routines described in the following sections will compute an index for some (i, j, k) location using the assumptions specified when you last called the routine `zoInitZOrderIndexing`.

The rationale behind having two different methods for computing an index, a generic one and two layout-specific ones, is to provide the means for applications to be coded in a way that does indexing agnostic of the underlying memory layout.

2.4.1 The Generic Method

The routine `zoGetIndex` takes as input a pointer to a `ZOrderStruct`, along with an i, j, k index, and will return a single `off_t` value indicating the index, or offset, along the one-dimensional *z-order* curve where that data item lives. For example:

```
off_t indx;
double d;
double *data;
ZOrderStruct *zo;

/* initialization of zo and data not shown here */

indx = zoGetIndex(zo, i, j, k); /* obtain the index */
d = data[indx];   /* obtain the datum */
```

Where `data` is an array containing doubles laid out in *z-order* format using the routine `zoConvertArrayOrderTo` and `zo` is a `ZOrderStruct` that has been initialized with `zoInitZOrderIndexing`.

The computation of `indx` is a function of the memory layout enumerator, either `ZORDER_LAYOUT` or `AORDER_LAYOUT`, specified to `zoInitZOrderIndexing` at initialization time.

2.4.2 The *z-order* Specific Method

If you know you want to obtain an index of some (i, j, k) location in an array laid out using *z-order* format, then your application my bypass the generic routine and use `zoGetIndexZOrder`.

The routine `zoGetIndexZOrder` takes as input a pointer to a `ZOrderStruct`, along with an i, j, k index, and will return a single `off_t` value indicating the location, or offset, along the one-dimensional *z-order* curve where that data item lives. For example:

```
off_t indx;
double d;
double *data;
ZOrderStruct *zo;

/* initialization of zo and data not shown here */

indx = zoGetIndexZOrder(zo, i, j, k); /* obtain the index */
d = data[indx];   /* obtain the datum */
```

2.4.3 The *a-order* Specific Method

If you know you want to obtain an index of some (i, j, k) location in an array laid out using *a-order* format, then your application my bypass the generic routine and use `zoGetIndexArrayOrder`.

The routine `zoGetIndexArrayOrder` takes as input a pointer to a `ZOrderStruct`, along with an i, j, k index, and will return a single `off_t` value indicating the location, or offset, along the one-dimensional *z-order* curve where that data item lives. For example:

```
off_t indx;
double d;
double *data;
```

```
ZOrderStruct *zo;

/* initialization of zo and data not shown here */

indx = zoGetIndexArrayOrder(zo, i, j, k); /* obtain the index */
d = data[indx];   /* obtain the datum */
```

3 Limitations and Future Work

3.1 Limitations

There are two noteworthy limitations of *zorder-lib*, owing to the nature of the *z-order* SFC layout. Both are described in Sec. 2.3, and are briefly repeated here.

First, there is no zero-copy conversion from *a-order* to *z-order*. When you convert an array from *a-order* to *z-order*, you are making a copy of the data. The only way around this limitation would be to, say, load data, already organized in *z-order* format, into memory for processing.

Second is the relationship between the power-of-two size limitation and memory "bloat" due to padding that will happen when you want to work with array that is not an even power-of-two in size in each dimension. Fig. 2 shows the type of memory inflation that will happen when exceeding the power-of-two limitation by 1 in one dimension; inflation may range from about 1.5× up to about 4×, depending upon various factors.

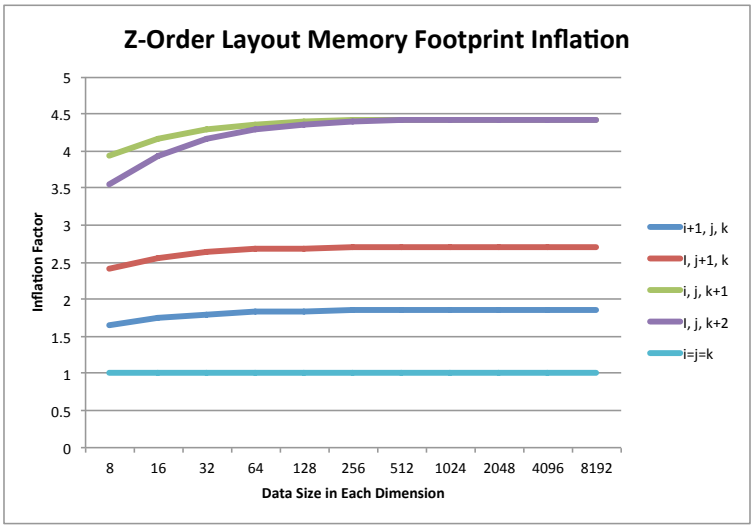

Figure 2: This chart shows the memory footprint inflation that will occur when the array `A` is not an even power of two in size in each dimension. The horizontal axis reflects array sizes $8^3, 16^3, ..., 8192^3$. When each of the i, j, k dimensions are all an even power of two, and the same size, then the size of the *z-order* array `Z` is the same size as the source array `A`. However, when the size of one of the dimensions increases by 1, the chart illustrates the effect of memory footprint increase depending upon which of i, j, k is not an even power of two. Future work on *zorder-lib* will focus on methods to reduce, if not eliminate, this inflation due to odd sized arrays.

However, memory inflation will also happen even when the data size is an even power-of-two in size in each dimension, but when the dimensions are not all the same size. This particular inflation happens due to how the *z-order* index is constructed, and a fix will be the subject of future work.

Fig. 3 shows the memory inflation effect that happens when all three dimension sizes are even powers of two, but two dimension sizes are equal and the third dimension size varies.

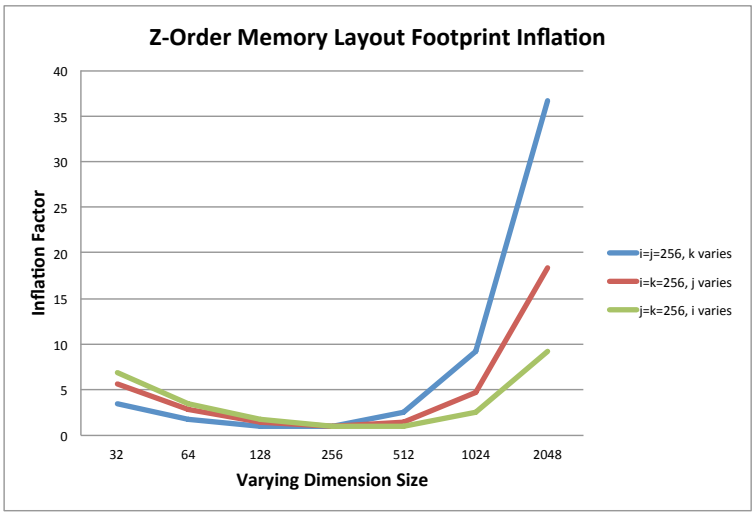

Figure 3: This chart shows the memory footprint inflation that will occur when the array `A` is an even power of two in size in each dimensions, but the dimensions are not all the same size. Here, we are holding two of the dimension sizes constant at 256, and varying the size of the third dimension.

The source of the memory inflation is due to the nature of how the *z-order* indexing works: it interleaves the *ijk* bits as *kjikjikji* rather than the more traditional *kkkjjjiii* formulation associated with *a-order* indexing.

In the absence of a more rigorous analysis, the amount of memory inflation that results from this *z-order* indexing artifact can be estimated as follows: each increase of 1 bit in resolution in a given dimension could result in up to an 8-fold increase in the amount of memory required. This is an upper bound on the cost of a single bit of resolution increase.

3.2 Future Work

There are several potential avenues for future work. One would include efforts to have the library routines be callable from other languages, like Fortran and Python.

Another area of future work is to expand *zorder-lib* to provide support for arrays of arbitrary dimension in a way that does not incur a memory inflation penalty.

In order to effectively implement the infrastructure needed to support datasets of arbitrary dimension (i.e., not an even-power-of-two in size), it is likely the case that a future version of *zorder-lib* will have a modified interface. Specifically, it will likely require applications to use a *getData* call, rather than the present approach of a generic *getIndex* call that returns a `sizeof_t` the application then uses as an index to obtain a datum from an array.

4 Obtaining the Source Code

zorder-lib has been approved for release by the LBNL Technology Transfer license under an Open Source license (BSD derivative). You may obtain the source code from this URL: `https://codeforge.lbl.gov/projects/z-order/`.

References

[1] Michael Bader. *Space-Filling Curves - An Introduction with Applications in Scientific Computing*, volume 9 of *Texts in Computational Science and Engineering*. Springer-Verlag, 2013.

[2] E. Wes Bethel, David Camp, David Donofrio, and Mark Howison. Improving Performance of Structured-memory, Data-Intensive Applications on Multi-core Platforms via a Space-Filling Curve Memory Layout. In *International Workshop on High Performance Data Intensive Computing, an IEEE International Parallel and Distributed Processing Symposium (IPDPS) workshop*, Hyderabad, India, May 2015.

[3] Peter J. Denning. The Locality Principle. *Commun. ACM*, 48(7):19–24, July 2005.

[4] Monica D. Lam, Edward E. Rothberg, and Michael 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*, ASPLOS IV, pages 63–74, New York, NY, USA, 1991. ACM.