Alternative Memory Models for ITK Images

Release 1.00

Dan Mueller[1]

July 3, 2009

[1]Philips Healthcare Informatics (PII), Philips Healthcare, Best, Netherlands

Abstract

By default ITK images use a contiguous memory model. This means pixel elements are stored in a single 1-D array, where each element is adjacent in memory to the previous element. However in some situations it is not desirable to use this memory model. This document describes three alternative memory models: slice contiguous, sparse, and single-bit binary images. Slice contiguous images are three-dimensional images, in which each slice is stored in a contiguous 1-D array, but the slices are not necessarily adjacent in memory. Slice contiguous images are well suited for interoperability with applications representing images using DICOM. Sparse images are n-dimensional images, in which each pixel is stored in a hash table data structure. This memory model is well suited for images with very large dimensions, but few pixels which are actually relevant. Single-bit binary images internally represent each pixel as a single-bit, in contrast to eight-bits required to represent a boolean. Single-bit binary images allow very compact representations for on-off masks. Source code, tests, and examples are provided to allow easy reproduction and use.

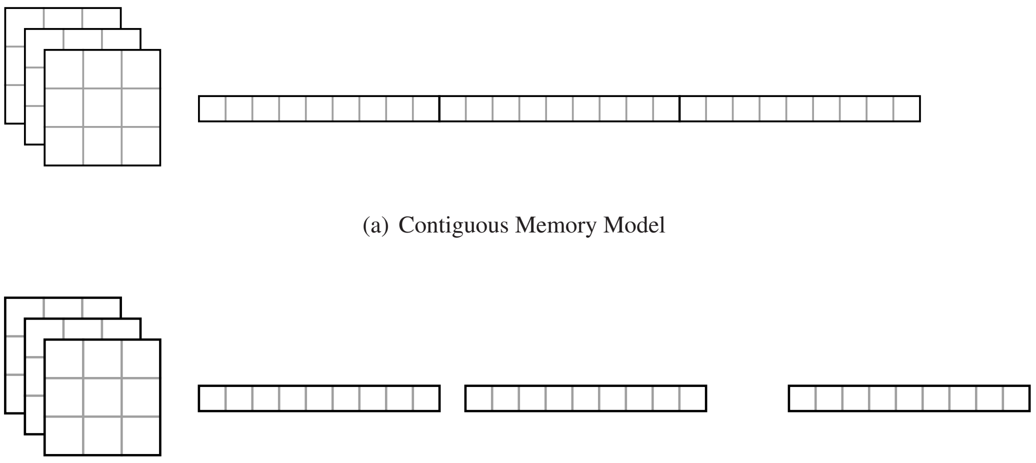
Contents

1 Introduction

ITK images currently employ an implicit contiguous memory model. By implicit we mean that the memory model is tightly coupled with many other classes within the toolkit. By contiguous we mean that pixel elements are stored as a 1-D array, where each element is adjacent in memory to the previous element. This underlying pixel array is not hidden behind a layer of abstraction but can be easily accessed using `itk::Image::GetBufferPointer()`.

Performing a quick `grep` over the codebase reveals numerous classes which directly access the image pixel array using `itk::Image::GetBufferPointer()`. Such tight coupling makes it difficult to abstract away the underlying memory model used by the image to store the pixel data. Some of the important classes which directly access the pixel array include: image iterators, image adpaters, image file readers/writers, and the VTK image export filter. There are also a number of other (minor) classes, such as: octree, watershed segmenter, BSpline deformable transform, optimized Mattes mutual information metric, etc.

These classes which directly access the pixel array, as well as ITK's backward compatibility policy, make it difficult — but not impossible — to introduce images with alternative memory models. This paper uses the same approach employed by `itk::VectorImage` to realize three new memory models for ITK images. The new image types do not require changes to the toolkit, and function with a majority of the existing filters and iterators. However they are only interoperable with classes which do not directly access the pixel array. For example it is not possible to use image IO, export to VTK, or use the optimized registration framework.



(a) Contiguous Memory Model

(b) Slice Contiguous Memory Model

Figure 1: Contiguous and slice contiguous memory models.

2 Proposed Image Memory Models

2.1 Slice Contiguous Images

A slice contiguous image is implicitly three-dimensional. Each slice is stored in a contiguous 1-D array, but the slices are not necessarily adjacent in memory. This representation is shown in Figure 1. The new `itk::SliceContiguousImage` may be useful for interoperability with existing code representing images using a slice contiguous memory model. This scenario is common for third party applications using DICOM.

This paper proposes an implementation for an image with such a memory model. This is realized by creating a new image class with custom pixel and neighborhood accessor functions; similar to `itk::VectorImage`. The pixel and neighborhood accessor functions provide a layer of indirection for iterators to access the image pixel data. Given the incoming offset for a contiguous memory model, the new accessors compute the index of the slice containing the pixel and the offset within that slice. The new image class is templated over the pixel type; it is not templated over the number of dimensions because it is always three.

One important different between `itk::Image` and `itk::SliceContiguousImage` is that for the later `GetBufferPointer()` always returns `NULL` ie. the pixel buffer makes no sense. Ideally this method would not even exist for slice contiguous images, but unfortunately too many other classes assume its existence.

A slice contiguous image is created in a very similar fashion to a “normal” image:

```
// Typedefs
typedef float PixelType;
typedef itk::SliceContiguousImage< PixelType > SliceContiguousImageType;

// Create image
SliceContiguousImageType::Pointer image = SliceContiguousImageType::New();
SliceContiguousImageType::IndexType start;
SliceContiguousImageType::SizeType size;
start.Fill( 0 );
size[0] = sizeSlice[0]; size[1] = sizeSlice[1]; size[2] = 4;
SliceContiguousImageType::RegionType region( start, size );
image->SetRegions( region );
image->Allocate( );
```

An additional step requires the pixel container to be configured with a list of slice pointers:

```
// Set slice pointers
SliceContiguousImageType::PixelContainer::SliceArrayType slices;
slices.push_back( slice2->GetBufferPointer() );
slices.push_back( slice4->GetBufferPointer() );
slices.push_back( slice3->GetBufferPointer() );
slices.push_back( slice1->GetBufferPointer() );
SliceContiguousImageType::PixelContainerPointer container =
  SliceContiguousImageType::PixelContainer::New();
container->SetImportPointersForSlices( slices, size[0]*size[1], false );
image->SetPixelContainer( container );
```

The slice contiguous image can now be used like any other image:

```
// Use the image with a filter
typedef itk::ShiftScaleImageFilter<SliceContiguousImageType,
  SliceContiguousImageType> ShiftScaleFilterType;
ShiftScaleFilterType::Pointer filter = ShiftScaleFilterType::New();
filter->SetInput( image );
filter->SetShift( 10 );
filter->Update( );
```

2.2 Sparse Images

A sparse image is an *n*-dimensional image in which each pixel is stored in a hash table data structure. Each time a pixel is set, a new offset-value pair is added to the hash table. Such a memory model means that little or no memory is allocated when the image is created, but the memory footprint grows as more and more pixels are set. This memory model is well suited for images with very large dimensions, but few pixels which are actually relevant. It should be noted that (at least for the moment) filters using images with this memory model

A sparse image can be created in the typical fashion:

```cpp
// Typedefs
const unsigned int Dimension = 2;
typedef unsigned short PixelType;
typedef itk::SparseImage< PixelType, Dimension > SparseImageType;

// Create image
SparseImageType::Pointer image = SparseImageType::New();
SparseImageType::IndexType start;
start.Fill( 0 );
SparseImageType::SizeType size;
size.Fill( 1000000 ); // try this with a "normal" image!
SparseImageType::RegionType region( start, size );
image->SetRegions( region );
image->Allocate( );
```

A sparse image requires a “background” value to be specified for undefined pixels:

```cpp
image->FillBuffer( 100 );
```

Pixels can be retrieved or set directly (using `Get`/`SetPixel`) or via an iterator:

```cpp
image->FillBuffer( 100 );
SparseImageType::IndexType indexA = {100, 100};
PixelType pixelA = image->GetPixel( indexA );
SparseImageType::IndexType indexB = {10000, 10000};
image->SetPixel( indexB, 5 );
PixelType pixelB = image->GetPixel( indexB );
```

2.3 Single-bit Binary Images

It is very common in image processing to create mask images which indicate regions which are “on” or “off”. Currently the smallest pixel element available in ITK for representing such images is `unsigned char` which is stored in a single byte (eight-bits). Even though a `bool` value can only represent `0` or `1`, it too is stored in a single byte (you can check this by writing out `sizeof(bool)`).

Single-bit binary images internally represent each pixel as a single-bit. As such the memory footprint for on-off masks can be lowered by (nearly) a factor of eight. Similar to slice contiguous images, single-bit binary images provide custom pixel accessor functions which convert the incoming offset to the relevant bit mask for the underlying data storage. Unlike slice contiguous images, single-bit binary images fit slightly better within the existing ITK framework so `GetBufferPointer()` makes sense in this context.

A single-bit binary image can be created in the typical fashion:

```
// Typedefs
const unsigned int Dimension = 2;
typedef bool PixelType;
typedef itk::SingleBitBinaryImage< Dimension > BinaryImageType;

// Create image
BinaryImageType::Pointer image = BinaryImageType::New();
BinaryImageType::IndexType start;
start.Fill( 0 );
BinaryImageType::SizeType size;
size.Fill( 65 );
BinaryImageType::RegionType region(start, size);
image->SetRegions( region );
image->Allocate( );
image->FillBuffer( true );
```

The bits are stored in blocks of 32, so the above code will actually allocate a buffer with size 96×96. As with all the presented image class, the binary image can be used with any iterator and/or filter which does not directly access the pixel buffer via `GetBufferPointer()`.

3 Performance

Although the memory models presented above may be advantageous for reducing memory usage in certain scenarios, they have a performance penalty. Accessing pixels stored in a contiguous array can be highly efficient, whereas the three new images require additional computation each time a pixel is accessed. This section provides a simple analysis of the performance for each of the proposed images.

The performance test measured four properties: (1) time to allocate the buffer, (2) time to fill the buffer, (3) time to set all pixels using an iterator, and (4) time to get all pixels using an iterator. Each test was run on a $512 \times 512 \times 512$ image, executed on my notebook (Intel Core 2 Duo, T7250 @ 2GHz, 3GB RAM, Windows Vista SP1 32-bit) a total of 5 times with mean times (in seconds) reported in the table below.

As can be seen, all the images have similar values for buffer allocation and getting pixels using an iterator. The sparse image was the fastest for filling the buffer because no memory is actually set at this moment, only a single “background” value. However it was also the slowest for setting pixels using an iterator, most likely because each pixel being set must be added to the hash table. The single-bit binary image was also fast for filling the buffer because the pixels are set in groups of 32-bits, rather than individual elements.

4 Conclusion

This submission proposed three images with alternative memory models: slice contiguous, sparse, and single-bit binary images. For the moment there is no IO support for any of the proposed images, however it might be possible to provide such support in the future. The proposed images should work seamlessly with most of the existing ITK filters — assuming they access the pixel data using iterators rather than

Image Type	Allocation	FillBuffer	Iterate Set	Iterate Get
Contiguous	< 0.001	0.110	0.103	0.047
Slice Contiguous	0.002	0.174	0.401	0.047
Sparse	< 0.001	< 0.001	17.6	0.046
Binary	< 0.001	0.004	0.159	0.049

Table 1: Performance timings in seconds for a $512 \times 512 \times 512$ image.

`GetBufferPointer();` These new images are just a few of the possible alternate memory models and should provide a good guide for others wanting to develop their own.