Tuning VTK data arrays

As data structures get more laden with features, performance tends to degrade over times. Bread-and-butter data containers in VTK suffer from this as well. On the bright side, we find that simple optimization doubles the speed of two common operations, `vtkDataArray::InsertNextTuple` and `vtkDataSetAttributes::CopyData` , with no sacrifice to usability. As a result, the speed of `vtkSynchronizedTemplates3D` is improved by 6 percent.

We hope that this experience serves as motivation to extent this effort to other common VTK data container operations. We believe that such efforts will lead to significant across-the-board performance improvement.

1 How slow are we, really?

To get a sense of how fast VTK is and how fast it "should" be. We compare it with STL and memcpy. STL makes a more relevant comparison here, while memcpy gives us a sort of time unit. From the results below, one can perhaps say that while VTK is slow, it is not embarrassingly slow.

	VTK	STL	memcpy
`vtkDataArray::InsertNextTuple`	0.21	0.16	0.02
`vtkDataSetAttributes::CopyData`	0.52	0.13	0.06

Table 1: Time (in seconds) of inserting 1M points 10 times and averaged over 10 runs. The compilation is in CMAKE RelWithDebInfo mode.

Code References

- Run `./arraytest -op insert` to get the VTK insertion time.
- Run `./arraytest -op insert -c` to get the memcpy insertion time.
- Run `./arraytest -op insert -stl` to get the stl insertion time.

2 Speeding up `vtkDataArray::InsertNextTuple`

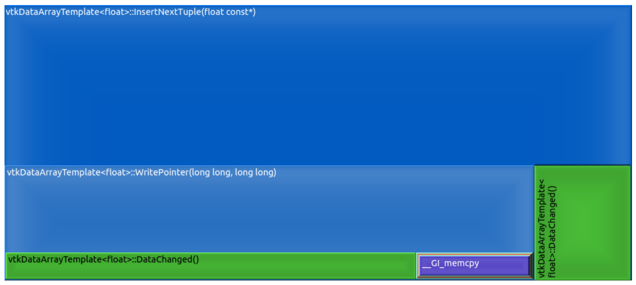

Figure 1: Profiling of `vtkDataArray::InsertNextTuple` using callgrind and visualized in kcachegrind

The profiling shows that the bottle-neck is WritePointer() and in the body of InsertNextTuple().

To attack the first bottleneck, we introduce a protected method AppendPointer() that takes advantage of the fact that the caller is already calling DataChanged(), so its own DataChanged() can be skipped. We can also remove some extra variables and a conditional because we know we are inserting to the tail.

The slowest statement in the second bottle neck turns out to be a division that computes the next id from the total length and the number of components. This takes up more than half of the time! We remove this division by introducing an extra counter variable and incrementing it instead.

The optimization more than doubles the speed. We are now beating the STL implementation.

	VTK (before)	VTK (after)	Speed-up
`vtkDataArray::InsertNextTuple`	0.21	0.1	×2.1

Table 2: Speed up of the test referenced in Table 1

Code References

- Run `./arraytest -op insert`.

3 Speeding up `vtkDataSetAttributes::CopyData`

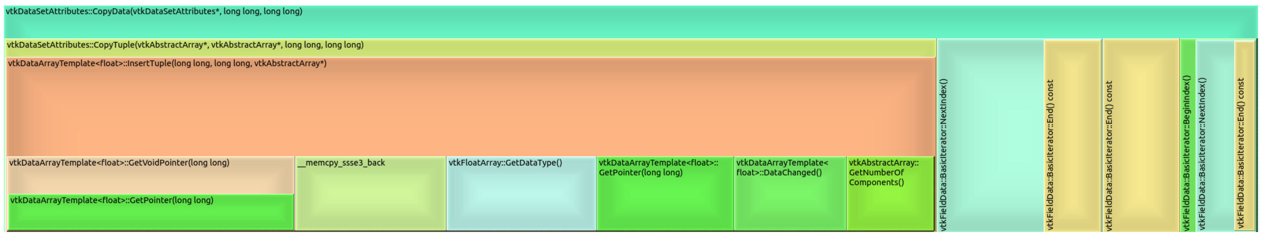

Figure 2: Profiling of `vtkDataSetAttributes::CopyData` using callgrind and visualized in kcachegrind

The profiling shows a lot of calls to GetNumberOfComponents() and GetDataTypes() calls, all coming from safety checks done in InsertTuple(). We know these are completely unnecessary if the copy source does not change. So our solution is to create a CopyHelper class that is instantiated with the source and destination and provides a copy method that just has the source and destination index, but not the source object pointer. Since the source does not change, we can safely call a trimmed protected method InsertTupleFast() that does not do those safety checks. Inside CopyHelper, we also use simple array data structures to support traversal of the various fields, which are faster than using vtkFieldData::BasicIterator.

Using CopyHelper on the caller side amounts to adding one call to construct CopyHelper, and replace `destination->CopyData(i,j,source)` by `copyHelper->Copy(i,j)`.

The optimization yields a 50 percent speed-up.

`vtkDataSetAttributes::CopyData`	CopyHelper	Speed-up
0.52	0.34	×1.52

Table 3: Speed up of the test referenced in Table 1

Code References

- For using the existing vtkDataSetAttributes::CopyData, run `./arraytest -op copy`.
- For using the copy helper, run `./arraytest -op copy -helper`.

3.1 "Lazy" Copy: A Coherence-Sensitive Copier

We also investigated using "lazy copy" to replace copying ranges of data by memcpy, since memcpy() is so fast. When a caller calls `LazyCopy::Copy(i,j)`, instead of actually performing the copying, it tries to accumulate a contiguous range and perform copying only when the continuous range ends—or when the caller explicitly calls `LazyCopy::Flush`.

Unlike the previous optimization techniques, lazy copying loses some functionality: During the copying process, the caller cannot read destination data since copying might not have occurred; the caller also must remember to "flush" in the end.

The actual speed-up gained also varies depending on whether these continuous ranges occur in the input. Nonetheless, in the best case —as in our simple test, we gain an order of magnitude speed-up by performing exactly one memcpy:

`vtkDataSetAttributes::CopyData`	LazyCopy	*Speed-up
0.52	0.05	×10

Table 4: Speed up of the test referenced in Table 1. We mark Speed-up with * because the result represents the best achievable.

Code References

- Run `./arraytest -op copy -lazy`.

4 Impact on filters

We compare the speed of vtkSynchronizedTemplates before and after the speed improvement. The performance gain is small but noticeable:

`vtkDataSetAttributes::CopyData`	CopyHelper(speed-up)	LazyCopy(speed-up)
1.32	1.24(×1.06)	1.26(×1.04)

Table 5: Speed up of vtkSynchronizedTemplates.

The relatively small percentage gain is because, for this particular filter, the time spent on inserting points and copying is a fairly small percentage of the overall time. We suspect that optimizing other data container routines such as vtkCellArray will give us further speed-up.

We also note that lazy copy does not help here, this is because the average range length is only about 2, so the saving is not enough to overcome the overhead.

5 Discussion

We need to have performance benchmark tests. There is no hard success/failure, but somehow we need to be able to keep track of the performance in various configurations. The performance score can be in terms of standard c libraries.