Cog ex Machina 5.0

Massively scalable computation made easy

Programming

Tutorial

13 May 2016





Table of Contents

[1. Introduction 3](#_Toc402799736)

[2. New features of Cog 5.0 4](#_Toc402799737)

[2.1 Lazy feedback 4](#_Toc402799738)

[2.2 Predicates 4](#_Toc402799739)

[2.3 Native-code linkable interface 4](#_Toc402799740)

[2.4 New time-model 4](#_Toc402799741)

[2.5 Multi-GPU and multi-node 5](#_Toc402799742)

[2.6 Others? 5](#_Toc402799743)

[3. Cog 4.1 features to be removed 6](#_Toc402799744)

[3.1 Keyword-based methods on Fields 6](#_Toc402799745)

[3.2 Seldom-used functions that will be moved to a library 6](#_Toc402799746)

[3.3 ComplexFields and ComplexVectorFields as Fields 6](#_Toc402799747)

[3.4 Scalarfield convolve MatrixField 7](#_Toc402799748)

[Appendix A: Greg’s whitepaper on lazy feedback and pipelining 9](#_Toc402799749)

# 1. Introduction

Cog ex Machina 5.0 is currently under development and not available to users. At the point of first release, this document will fully describe the Cog 5.0 language API, much as the Cog Ex Machina 4.1 Programming Tutorial does today for Cog 4.1. In the meantime, this document describes only the changes a user is likely to see in moving from Cog 4.1 to Cog 5.0.

Since Cog 5.0 represents an increment of the major version number, a user should expect some models will not compile or run. Thus, there will be a porting activity to move existing models from Cog 4.1 to Cog 5.0. We have attempted to deprecate features of Cog 4.1 that will be removed in Cog 5.0 to minimize the number of models that need to be altered. In addition, we will distribute a porting guide that covers the common issues in moving a model from Cog 4.1 to Cog 5.0.

The remainder of the document is organized in two sections:

1. New features that will be added to Cog 5.0, and
2. Current features of Cog 4.1 that will go away.

Keep in mind that the additions and deletions described in this document are under discussion and should not yet be relied upon.

# 2. New features of Cog 5.0

## 2.1 Lazy feedback

The Cog user currently adds state to his/her Cog model by employing the feedback operator <==. The Cog 4.1 mechanism is fairly rigid however since the value of the right hand side of the feedback operator must be computed within a single clock tick of the Cog discrete-time simulation. In practice, models such as deep neural networks have a much more relaxed state-update constraint. For example, the new filter weights that would result from training on a single image need not immediately replace the existing weights before training on a second image can begin. The weight update can take place a number of clock ticks later, i.e. in a lazy fashion.

A model’s use of lazy feedback will help drive the Cog 5.0 compiler’s *pipeliner*, an important part of the process of partitioning a model amongst multiple GPUs.

The precise form for specifying lazy feedback has yet to be determined. It may involve a new operator syntax like <~~. Whether the allowed decay is specified on a per-operator basis or by some global parameter setting is yet to be determined. The latest thoughts in this area from Greg are included in a white paper appended as Appendix 1 to this document.

## 2.2 Predicates

Currently, the Cog runtime has no way to optimize for model state that is only occasionally updated. A field that holds model state is copied back and forth between two buffers on each simulation clock tick. Currently, a modeler that wants to update a field only if a field enable is active must write:

val stateFulField = ScalarField(Rows, Columns)

stateFulField <== enable \* newFieldState + (1.0f – enable) \* stateFulField

The proposal is to add new syntax to expose to the compiler that a field state update is *predicated* on another field. While the precise form has not been discussed, the following is one approach that would be sufficient to inform the compiler:

stateFulField <== ifThenElse(enable, newFieldState, stateFulField)

Of a related nature is the case of a kernel updating a small portion of a field on each clock tick (perhaps based on some winner-take-all logic). If we support such a selective merge operation, the runtime could eliminate the bulk field copy.

## 2.3 Native-code linkable interface

Cog 5.0 will be able to appear as a linkable library to natively compiled C/C++ code. This is further described in the Cog ComputeGraph, FunctionInterface, Runtime, SensorActuator and TensorField ERS’s.

## 2.4 New time-model

We’ve often discussed ways of describing real-world sensor and actuator timing such that the compiler could partition a model to meet real-time constraints. Since we’re breaking the API, should we add any hooks for this now?

## 2.5 Multi-GPU and multi-node

While not an API change per se, with Cog 5.0 we intend to enable a model to be spread across multiple GPUs and compute nodes.

## 2.6 Others?

Cog team members are encouraged to add their favorite Cog 5.0 enhancement here for discussion.

# 3. Cog 4.1 features to be removed

## 3.1 Keyword-based methods on Fields

Currently, Cog operators represented by keywords can appear as methods on fields, for example:

val filtered = image.convolve(filter, BorderValid)

To streamline the Cog API, we propose to eliminate such field methods in favor of their function call equivalent:

val filtered = convolve(image, filter, BorderValid)

The arithmetic operators and feedback operator would remain as methods on Field, as in:

counter <== counter + 1

## 3.2 Seldom-used functions that will be moved to a library

The following Cog functions are deprecated in Cog 4.1 and will be removed from the core and supported from a library in Cog 5.0:

* bilateralFilter(field: Field, spatialFilter: Matrix, rangeSigma: Float) : Field
* conditionNumber(field: Field): Field
* determinant(field: Field): Field
* domainFilterColumns(field: Field, domainTransform: ScalarField, boxFilterRadius: Float): Field
* domainFilterRows(field: Field, domainTransform: ScalarField, boxFilterRadius: Float): Field
* domainTransformRows(field: Field, spaceSigma: Float, rangeSigma: Float): Field
* invertMatrices(field: Field): Field
* localMax(field: Field, neighborhood: Matrix): Field
* localMaxPosition(field: Field, neighborhood: Matrix): Field
* localMin(field: Field, neighborhood: Matrix): Field
* localMinPosition(field: Field, neighborhood: Matrix): Field
* maxPosition(field: Field): Field
* medianFilter(field: Field): Field
* nonMaximumSuppression(field: Field): Field
* orientedNonMaximumSuppression(field: Field, orientation: Field): Field

These kernels will be re-written in the GPUOperator framework, although that is the plan for all existing HyperKernels. The significance of the deprecation of these functions is that they will be supported out of a toolkit library instead of bundled with the libcog jar. The maxPosition function may be best left in the core, since it’s a helper to the winnerTakeAll function which is not on the deprecation list.

## 3.3 ComplexFields and ComplexVectorFields as Fields

Now that Cog supports multi-output GPUOperators and kernels, the support of ComplexFields and ComplexVectorFields as a primitive field types should be removed. This would simplify the Cog core functions relating to field layout and addressing. This might also simplify advanced merging strategies that rely on bundling of a kernel’s output vector into “small tensors.” Finally, ComplexVectorField support was never added to GPUOperators in anticipation of ComplexFields being moved out of the core. If we leave ComplexFields in the core, then ComplexVectorFields support should be added to GPUOperators.

The bulk of Cog’s complex field functionality could be preserved even if ComplexField and ComplexVectorField become classes in a non-core library. The biggest change is likely that ComplexField and ComplexVectorField would not be subclasses of Field, which is a change that could conceivably break user code. Issues of supporting non-core ComplexFields in the debugger need to be explored.

## 3.4 Scalarfield convolve MatrixField

Cog’s approach to anisotropic convolution, occuring as a ScalarField convolved with a MatrixField, is confusingly supported by the same convolve keyword as is used for myriad forms of isotropic convolution. This should be cleaned up.

The simple case of a ScalarField image isotropically convolved with a ScalarField filter is cleanly represented as:

filteredScalarField = convolve(imageScalarField, filterScalarField, BorderValid)

To handle the cases of an image isotropically convolved with a bank of filters, or a bank of images convolved against a single filter, or even a bank of images convolved with a similarly sized bank of filters, the Cog API includes various forms of convolution where the banks of images and/or filters are held as VectorFields:

filteredVectorField = convolve(imageScalarField, filterVectorField, BorderValid)

filteredVectorField = convolve(imageVectorField, filterScalarField, BorderValid)

filteredVectorField = convolve(imageVectorField, filterVectorField, BorderValid)

For API symmetry, even a “2D” bank of images can be convolved with a similarly sized 2D bank of filters, where the images and filters are held as MatrixFields:

filteredMatrixField = convolve(imageMatrixField, filterMatrixField, BorderValid)

Further, to support the needs of the neural network package and its use of image frames, various “plane mixing” strategies were introduced to the convolution of VectorFields:

filteredVectorField = projectFrame(imageVectorField, filterVectorField, BorderValid)

filteredVectorField = backProjectFrame(imageVectorField, filterVectorField, BorderValid)

filteredVectorField = convolveFilterAdjoint(imageVectorField, filterVectorField, BorderValid)

Remember that in all these cases, the filter that is applied to a particular image is the same for all field points of the image, and that the filter shape corresponds to the fieldType.fieldShape of the VectorField holding the filters.

Now consider Cog’s approach to anisotropic convolution, where the filter applied at each image field point is different. In this case, the various filters are held as a matrix field whose fieldShape matches the image fieldShape, and whose tensorShape matches the filter shape:

filteredScalarField = convolve(imageScalarField, filterMatrixField, BorderValid)

This use of the same keyword for a significantly different operation is confusing. We propose introducing an anisotropicConvolve keyword for the above case:

filteredScalarField = anisotropicConvolve(imageScalarField, filterMatrixField, BorderValid)

Then, we should either:

1. Not support image or filter banks held as matrix fields, or
2. Define convolve(ScalarField, MatrixField) as producing a MatrixField, it being the isotropic convolution of a ScalarField image and a 2D arrangement of filters held as a MatrixField whose fieldShape equals the filter shape.

# Appendix A: Greg’s whitepaper on lazy feedback and pipelining