Cog X Function Interface

External Reference Specification

Version 2

Greg Snider

# 1. Overview

The Cog X Function Interface is a thin veneer on the Cog X Native Runtime that, for a broad class of algorithms, can make invocation of a Cog X Compute Graph look like a function call.

Cog Function Interface

Cog Native Runtime

GPU

GPU

GPU

GPU

Cog App

*function call (reentrant, thread-safe)*

The Function Interface is simpler to use than the raw Native Runtime: the caller passes in one or more tensor fields as inputs to a single evaluate() function, then receives one or more tensor fields as outputs for the result. The evaluate() function is reentrant and thread-safe so that the application may contain multiple threads launching independent calls to evaluate(). The reentrancy provides greater throughput since it allows the Runtime to interleave multiple evaluate() requests in a Compute Graph pipeline, providing higher utilization of hardware resources without significantly impacting latency.

thread

thread

thread

thread

evaluate()

evaluate()

evaluate()

evaluate()

Cog Function Interface

Application

Cog Native Runtime

The limitation is that the Function Interface works only for *non-temporal* Compute Graphs—Compute Graphs that are trained to respond to isolated inputs rather than streaming inputs. *Temporal* Compute Graphs, on the other hand, deal with time and input sequences, such as recognizing actions rather than static patterns. Such temporal algorithms must use the Native Runtime interface: see the Cog Native Runtime External Reference Specification for more details.

Although the Function Interface can support Compute Graphs that learn, one must be careful to either: (1) use learning algorithms that are not subject to catastrophic interference when given input streams that don’t match desired statistics; or (2) carefully control the statistics of the input streams to match those of the expected use case.

In short: if the deployed application is non-temporal and does not learn, the Function Interface may safely be used to simplify interaction and maximize throughput.

# 2. Interface

The Function Interface is a linkable library that is dynamically linked with the Native Runtime library. The API for the library is a set of C data types and functions that may be called by the application.

#### Summary

A quick look at the Function Interface functions, detailed below.

// Versioning

int majorVersionNumber()

int minorVersionNumber()

// Startup

int createHandle()

CogErr initialize(int handle, char\* computeGraphFile, char\* resourcesFile)

CogErr requiredResources(int handle, struct Resources\* requiredResources)

CogErr bindContainers(int handle, int containers, char\*\* containerAddresses)

// Function interface

int optimalThreadCount(int handle)

CogErr inputs(int handle, int\* inputCount, char\*\* inputNames[])

CogErr outputs(int handle, int\* outputCount, char\*\* outputNames[])

CogErr evaluate(int handle, float\*\* inputBuffers, float\*\* outputBuffers)

// Termination

CogErr shutdown(int handle)

The “Versioning,” “Startup,” and “Termination” functions are identical to those in the Native Runtime library. See the Cog Native Runtime External Reference Specification for descriptions. Descriptions of the “Function Interface” functions follow.

#### optimalThreadCount()

Returns the number of concurrent threads in the application that will maximize throughput of a single instance of a Compute Graph. Using more threads than this will not generally reduce throughput much, though it will put a strain on memory resources. If the application requires more throughput, it may create multiple instances of a Compute Graph (thus using more GPUs) using createInstance(), since optimalThreadCount() applies independently to each instance. It may also launch multiple processes, each containing a single instance of a Compute Graph, to achieve greater robustness.

int optimalThreadCount(int handle)

#### inputs()

Returns the number and names of the input tensor fields in the inputCount and inputName return parameters. The handle parameter references a specific ComputeGraph (since a program may create several) using the identifier returned by createInstance(). The calling program may determine the sizes and types of these fields by querying the computeGraphField passed in to the initialize() function. See the Cog Resource Serialization External Reference Specification for details. All calls to evaluate() must present input field data buffers in the order defined by this function. The function returns an error status, defined in the Native Runtime External Reference Specification.

CogErr inputs(int handle, int\* inputCount, char\*\* inputNames[])

#### outputs()

Returns the number and names of the output tensor fields in the outputCount and outputName return parameters. The calling program may determine the sizes and types of these fields by querying the computeGraphField passed in to the initialize() function. See the Cog Resource Serialization External Reference Specification for details. All calls to evaluate() must present output field data buffers in the order defined by this function. The function returns an error status, defined in the Native Runtime External Reference Specification.

CogErr outputs(int handle, int\* outputCount, char\*\* outputNames[])

#### evaluate()

Evaluates a Compute Graph for inputs held in inputBuffers, writing results to outputBuffers. This is a blocking call which returns when the computation is complete and outputBuffers have been written.

CogErr evaluate(int handle, float\*\* inputBuffers, float\*\* outputBuffers)

# 3. Protocol

The Runtime system must be initialized in the following order:

1. A Compute Graph handle must be created by a call to createHandle().
2. The initialize() function must be called to bind a Compute Graph to the Runtime and analyze it for resource requirements.
3. The bindContainers() function must be called if the application is a service requiring multiple nodes in a compute cluster. An application runnable on a single node may omit this.

After that, evaluate() may be called by multiple threads.

The shutdown() function should always be called at the end of an application to free up resources.