Cog X Native Runtime

External Reference Specification

Version 3

Greg Snider

# 1. Overview

The Cog X Native Runtime is a linkable library that executes a Cog X Compute Graph on a single node or cluster of nodes containing GPUs or other massive multicore hardware. It handles all initialization, scheduling, communication and synchronization of the Compute Graph computation. The Native Runtime replaces the existing Cog X Runtime system, which runs on the Java Virtual Machine (JVM), and extends its functionality to handle distributing the computation across a compute cluster containing multiple GPUs or other multicore devices on multiple nodes.

Note: non-streaming applications will find that the Cog Function Interface, a glue layer on top of the Native Runtime, presents a much simpler interface. See the Cog Function Interface External Reference Specification for details.

Cog App

Cog Native Runtime

GPU

GPU

GPU

GPU

#### 1.1 Functionality

The Runtime performs the following functions:

1. *Optimizes* the Compute Graph with constant folding and pipelining transformations.
2. *Analyses* an optimized Compute Graph for resource needs (GPUs, memory).
3. *Allocates* GPU internal resources needed for the computation.
4. *Controls* initialization and execution of the Compute Graph.
5. *Enables* sensors, actuators, debugging, checkpointing, and restarting with coherent read and write functions.

The interface is designed for *temporal* applications that need to process streams of data, such as video streams, in which recognizing temporal patterns (movements, behaviors) is crucial. For temporal applications, inputs and outputs of the computation are streams of tensor fields. The tensor fields are exchanged between the application and library using callbacks.

in stream

in stream

out stream

out stream

Cog Native Runtime Library

Application

compute hardware

*Streaming (temporal) application*

callback

callback

callback

callback

*Non-temporal* applications (those recognizing or reacting to static patterns), are also supported, but the simpler Function Interface is recommended for them. See the Function Interface External Reference Specification for details.

#### 1.4 Components

The Cog X Runtime system comprises four components:

1. A linkable Runtime Library that presents a C interface to an application.
2. A Compute Graph Controller than manages the execution of GPUs on a single node.
3. A Messaging System that abstracts data transport, allowing a Cog application to run on everything from a laptop or cellphone to multiple nodes in a compute cluster.
4. Various transports, such as shared memory, pipes, sockets, RDMA, etc., plus future transports such as those being developed for “*The Machine*.”

The following figure shows how these components interact:

CompGraph Controller

CompGraph Controller

CompGraph Controller

Process

Process

Process

GPU

GPU

GPU

GPU

GPU

GPU

Cog X Runtime Lib

Cog X App / Debugger

Process

Messaging System

shared memory

pipes

sockets

… etc.

*transports*

The application linked to the Runtime library runs in its own process. (Note that the Cog graphical debugger, normally used to develop Cog X applications, appears to the runtime system like any other application—it is not privileged. Thus it is possible for a user to write a custom debugger.)

The runtime library spawns off one or more Compute Graph Controller processes to control GPUs, either locally or across multiple nodes in a cluster. On a laptop, for example, the runtime library would spawn a single Compute Graph Controller process to handle GPU execution, communication and synchronization; this process dies when the application terminates.

The Messaging System provides control and data transfer between the processes, using an appropriate transport depending on the location of the processes. For example, processes on the same node would exchange data using shared memory, while those on different nodes might use TCP/IP, RDMA, Infiniband verbs, etc., depending on the capabilities of the platform. Abstracting the transport in this way makes deployment of a Cog application on a spectrum of platforms easy to support, and enables adding new transports (such as “gen Z” transports for “The Machine”) without rewriting code.

Each of the processes implementing the distributed runtime shown above may be packaged into units called Docker containers. Each container represents a subset of the resources on a single node in the cluster, and includes code for managing one or more GPUs on that node. The Compute Graph Controller containers are identical replicants, independent of the application. When the application (or the Cog Debugger) is started, the local runtime library linked with the application takes control of the containers and partitions the application’s Compute Graph across them. Thus the Docker containers differentiate at runtime to cooperatively execute the application under the direction of the runtime library linked to the application.

A small Compute Graph may also be packaged with the application into a single process using an “in-process memory” transport if there are enough local resources:

CompGraph Controller

GPU

GPU

Cog X Runtime Lib

Cog X App / Debugger

Process

Messaging System

in-process memory transport

# 2. Interface

The Runtime library may be statically or dynamically linked with a Cog X application. It depends upon, and dynamically links with, several other libraries: OpenCL, [others to be determined]. The API for the library is a set of C datatypes and functions that may be called by the application.

#### Summary

A quick look at the Runtime functions, detailed in following sections. Nearly all functions return an error code of type CogErr which should be checked after each invocation.

// 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)

// Sensors and Actuators

CogErr bindSensor(int handle, int sensorIndex, struct CogSensor sensor)

CogErr bindActuator(int handle, int actuatorIndex, struct CogActuator actuator)

int pipelineDelay(int handle, int sensorIndex, int actuatorIndex)

// Compute graph execution (pipeline hidden)

CogErr reset(int handle)

CogErr step(int handle)

CogErr run(int handle, int steps, CogErr \*done(int handle, long time))

CogErr stop(int handle)

long time(int handle)

// Compute graph execution (pipeline exposed)

CogErr clock(int handle)

CogErr drainPipeline(int handle)

// Checkpoint / restart

CogErr checkpoint(int handle, char\* checkpointService, char\* key)

CogErr restore(int handle, char\* checkpointService, char\* key)

// Debugging support

CogErr writeOnce(int handle, int fieldIndex, float\* fieldBuffer,

CogErr (\*done)(int handle, int fieldIndex, float\* buffer))

CogErr readOnce(int handle, int fieldIndex, float\* fieldBuffer,

CogErr (\*done)(int handle, int fieldIndex, float\* buffer))

// Termination

CogErr shutdown(int handle)

Functions with callbacks (such as sensors and actuators) use a separate thread for each callback invocation. Hence a computation with 3 sensors and 2 actuators will use 5 threads to execute the callbacks.

If a synchronous call hangs, the Runtime will timeout and force the caller to receive an error code in the returned value.

If the computation is running, the time() function may be called periodically to verify progress.

Sizes of tensor fields may be obtained by querying the computeGraphFile passed to initialize. See the Cog Resource Serialization External Reference Specification for details.

#### majorVersionNumber

The major version number of this library. This must be compatible with the version number of Compute Graph linked to the library with initialize().

int majorVersionNumber()

#### minorVersionNumber

The minor version number of this library. This must be compatible with the version number of Compute Graph linked to the library with initialize().

int minorVersionNumber()

#### createHandle

Creates a handle (a small integer) for a Compute Graph. The Compute Graph will be instantiated by the initialize() function. An application may create multiple handles and thus instantiate multiple Compute Graphs.

int createHandle()

#### initialize()

Instantiate a Compute Graph defined in a compute graph file and bind it to a handle. Use resourcesFile to determine the compute resources (such as those in a cluster) available for the computation. See the Cog Resource Serialization External Reference Specification for the information contained in the two files and how the application may access it. This will fail with an error code if the version number embedded in the computeGraphFile is incompatible with the version number of the library.

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

#### requiredResources()

Return the requiredResources needed to implement the ComputeGraph bound to handle.

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

#### bindContainers()

Bind Docker containers for implementing the computation to the runtime. If containers is 0, the Runtime will run locally using local GPUs only. If containers is 1 or greater, containerNames is an array of names (IP address: port) for the Docker containers.

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

#### bindSensor()

Bind a CogSensor struct to the sensor with the index sensorIndex. See the Cog Sensor Actuator External Reference Specification for details

CogErr bindSensor(int handle, int sensorIndex, struct CogSensor sensor)

#### bindActuator()

Bind a CogActuator struct to the actuator with the index actuatorIndex. See the Cog Sensor Actuator External Reference Specification for details

CogErr bindActuator(int handle, int actuatorIndex, struct CogActuator actuator)

#### pipelineDelay()

Pipeline delay, in “ticks,” between the specified sensor and the specified actuator. This is only meaningful when using the run() command which exploits pipelining to maximize throughput.

int pipelineDelay(int handle, int sensorIndex, int actuatorIndex)

#### reset()

Reset the computation to its initial state, stopping the computation if currently running, and reset the internal time (measured in “ticks”) to zero. This is a synchronous call that returns when complete and leaves the computation in a coherent state that makes the Compute Graph appear to be unpipelined. This will fail with an error message if there is an unbound sensor or actuator.

CogErr reset(int handle)

#### step()

Step the computation one tick, causing sensor and actuator callbacks to be invoked. This is a synchronous call that returns when complete and leaves the computation in a coherent state that makes the Compute Graph appear to be unpipelined. This function is slow since it doesn’t take advantage of pipelining and is primarily used for debugging. This will fail with an error message if there is an unbound sensor or actuator.

CogErr step(int handle)

#### run()

Advance the computation by steps ticks. If the computation is already running, this adds steps ticks to number of cycles the computation will run. This uses pipelining to maximize throughput and is most useful for streaming applications. This may be used in conjunction with pipelineDelay() to correlate input Sensors and output Actuators if desired. When the specified number of steps has been completed, the Compute Graph pipeline is drained to present a coherent (unpipelined) state, and the done() callback is invoked with the current value of time as an argument. This is an asynchronous call that returns immediately. This will fail with an error message if there is an unbound sensor or actuator.

CogErr run(int handle, int steps, void (\*done)(int handle, long time))

#### stop()

Stop a running computation. Does nothing if not running, otherwise it stop execution, drain the pipeline to present a coherent (unpipelined) state, and invoke the done() callback specified in run(). This is an asynchronous call that returns immediately.

CogErr stop(int handle)

#### time()

Execution time, in ticks, since last reset.

long time(int handle)

#### clock()

Clock the Compute Graph pipeline, leaving the pipeline exposed. This is a low-level function useful for implementing the Function Interface, allowing multiple computations to be interleaved to maximize throughput. Not generally useful for applications.

CogErr clock(int handle)

#### drainPipeline()

Drains the Compute Graph pipeline to present a coherent (unpipelined) view of the computation.

CogErr drainPipeline(int handle)

#### checkpoint()

Checkpoint a computation, saving all stateful fields. The checkpointService argument is a string defining an (IP address, port number) for the state saving service, and key is a string (that must be unique to the applicaton) that the service will associate with the saved state. This will temporarily stop a running computation in order to retrieve coherent state, but automatically resume it once coherent state data has been read, allowing the actual computation to proceed in parallel with the checkpoint service.

CogErr checkpoint(int handle, char\* checkpointService, char\* key)

#### restore()

Restore a Compute Graph state from a saved checkpoint. If the Compute Graph is currently running, it is first stopped. The checkpointService argument is a string defining an (IP address, port number) for the state saving service, and key is a string (that must be unique to the applicaton) that the applicated used to store the state using the checkpoint() function. After the restore completes, the computation is left in a stopped state.

CogErr restore(int handle, char\* checkpointService, char\* key)

#### writeOnce()

Write a field state once. This is a hook allowing a user, via a debugger, to experimentally write a field value. When the write has been completed, the done() callback function is invoked. The fieldBuffer is owned by the caller.

CogErr writeOnce(int handle, int fieldIndex, float\* fieldBuffer,

CogErr (\*done)(int handle, int fieldIndex, float\* buffer))

#### readOnce()

Read a field state once. This is a hook allowing a debugger to experimentally read a field for display. When the read has been completed, the done() callback function is invoked. The fieldBuffer is owned by the caller.

CogErr readOnce(int handle, int fieldIndex, float\* fieldBuffer,

CogErr (\*done)(int handle, int fieldIndex, float\* buffer))

#### shutdown()

Shutdown the computation and release all resources (such as Docker containers, memory, GPU resources, etc.).

#### type CogErr

Error codes returned by Runtime functions. All successful functions return COG\_SUCCESS which is equal to zero.

typedef enum {

COG\_SUCCESS = 0,

... (to be determined)

} CogErr;

#### struct Resources

GPU and memory resources required to execute a Compute Graph. Currently this uses a single GPU as a coarse unit of allocation; this may be refined in the future.

struct Resources {

int gpus; // Number of GPUs required

int fieldCount; // Number of fields in the compute graph.

}

# 3. Protocol

The Runtime system must be initialized in the following order:

1. createHandle() must be called to create a reference for a Compute Graph.
2. The initialize() function must be called to bind a Compute Graph to a handle and analyze it for resource requirements. This can take a bit of time (seconds) depending on the size of the Compute Graph.
3. The bindContainers() function may be called to supply the necessary compute resources. Not needed, though, if the computation can fit on a single node.
4. All sensors and actuators must be connected into the Compute Graph using the bindSensor() and bindActuator() functions.

After that, the computation may be controlled with the reset(), step(), run(), and stop() functions. The sensor and actuator callback functions will be called automatically in response to step() or run() at the proper times.

Checkpoint / restart may be implemented, if desired, using the checkpoint() and restore() functions. Checkpoint will cause a slowdown in the computation when invoked, so invocation frequency is a trade-off a user must make.

The readOnce() and writeOnce() functions are primarily intended for the Debugger, but a user application may use them if desired.

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