Massively-Parallel Computing on Cog ex Machina
Greg Snider

HP Laboratories
HPL-2012-179

Keyword(s):
parallelism; GPU; high performance computing; cognitive computing

Abstract:
Cog ex Machina is a software framework for building massively-parallel applications on commodity, multicore hardware. Complex models may be expressed in a simple, abstract programming model, while hiding the complexities (threads, locks, synchronization, communication) of the underlying hardware platform.
Massively-Parallel Computing on Cog ex Machina

Greg Snider
Hewlett-Packard Laboratories

Multicore Future
The last decade has seen the number of processing cores per chip explode while clock rates have increased only modestly. From an application perspective, this is not a particularly welcome development: programming multiple cores executing in parallel is currently trickier and more labor intensive than a single core. But limitations of physics and current technology do not allow economical multi-terahertz processors, so we must adapt to what the market can provide.

Graphics processing units, GPUs, are now ubiquitous and cheap, providing more than 1000 cores on a chip at very low cost. They are not ideal platforms for building high performance applications. A GPU typically has a streaming memory model, SIMD-like execution structure, limited on-chip memory, limited I/O bandwidth, and a high launch time overhead. Memory caching must, to a large extent, be done in software. One must do some clever programming to use them efficiently.

But GPUs are evolving away from their graphics legacy to more general platforms. On-chip GPU hardware caches are increasing in size, and some newer designs include much more on-chip memory per core, much higher memory bandwidth, and the potential for MIMD processing (for example, one start-up claims an architecture that supports up to 64K cores per chip, up to 1 MByte per core, with a memory bandwidth of 4 bytes per FLOP). The CUDA and OpenCL software toolkits allow programmers to write applications which run directly on GPUs while interfacing with conventional software on a CPU.

Learning to repair video streams. The video stream on the left has a permanent occlusion simulating the way retinal veins block images in human eyes. A short Cog program learns the location of the veins and dynamically paints in the missing information in real time.
We anticipate that technology coming within the next ten years, (such as nanostores, multicore chips with embedded, low-power memory, and photonic interconnect), will make commodity, multicore hardware computationally more powerful while using much less energy. We will be able to build “big data” applications running on servers containing many millions of cores, and low-level cognitive applications, such as speech and visual pattern recognition, running on mobile platforms containing thousands of cores. There is a broad spectrum of interesting compute-intensive applications between those extremes.

But how in the world do you program millions of cores? That is the topic of this paper.

**Cog ex Machina**

Cog ex Machina is a software framework for building applications on massive, multicore hardware. Cog is aimed especially at “cognitive applications,” applications which must autonomously and adaptively interact with a changing and uncertain world. The programming paradigm contains only two abstractions: *dynamic fields*, which represent state information as multi-dimensional arrays; and *operators*, which combine field states to produce new states for dynamic fields. The hardware platform is abstracted away so that programmers do not see—nor need to worry about—cores, threads, locks, communication or synchronization. Computation is deterministic, race- and deadlock-free. Cog applications may interface with conventional software such as databases, file systems, graphical user interfaces, etc.

**Hardware Platform**

Cog targets clusters of compute nodes interconnected with a network, with each node consisting of a CPU and one or more GPUs. Cog uses the GPUs for the hardcore number crunching, and the CPUs for communication of information between the nodes and to synchronize the global computation:

**Software Framework**

Cog software is a framework rather than a conventional software library, an approach sometimes referred to as *inversion-of-control*. Cog applications are written in a declarative manner (below, right), describing the computation using field and operator abstractions, rather than imperatively implementing it (left):
Because of its declarative nature, building Cog applications feels a lot like building synchronous, digital hardware. Dynamic fields hold state information and are analogous to hardware registers. Just as with hardware registers, dynamic fields have a fixed size and may have an initial state value that the system forces into them upon "reset," and they advance their state by storing the data on their input when “clocked.” Operators transform field information and are analogous to networks of logic gates. A Cog application is a state machine that synchronously advances its state each “clock cycle” using the operators to determine the next states of the dynamic fields:

Dynamic Fields
A dynamic field is a multidimensional container of algebraic objects. The algebraic objects, called "lattices" within Cog, are themselves represented by multidimensional arrays of...
floating point numbers. The simplest, zero-dimensional lattice is a scalar consisting of a single floating point number; a dynamic field of scalars is called a `DynamicScalarField`. A one-dimensional lattice is a vector, so a dynamic field of vectors is called a `DynamicVectorField`. Other lattices include matrices, complex numbers, and pixels (holding four floating point numbers representing red, green, blue, and alpha components) with the associated field names of `DynamicMatrixField`, `DynamicComplexScalarField` and `DynamicColorField` respectively. More elaborate dynamic fields, such as quaternion fields or tensor fields, may be constructed from the basic fields supplied in Cog.

Creating a dynamic field requires specifying the “shape” of the field itself (the number of dimensions and the number of discrete elements along each dimension) and the shape of the lattice at each point in the field. Any given field holds lattices of identical shape and type. A dynamic field may optionally be given an initializer function that defines the state to be assigned whenever the Cog application is started or reset. If no initializer is specified, the dynamic field is initialized to zeroes.

**Operators**

An operator can be applied to the content of a dynamic field, with optional arguments consisting of constants or contents of other dynamic fields.

Algebraic operators include binary operators (`+` `-` `*` `/` `%` `max` `min` `pow` `atan2`), unary operators (`abs` `acos` `asin` `cos` `cosh` `exp` `log` `rectify` `signum` `sin` `sinh` `sq` `sqrt` `tan` `tanh`), and comparison operators (`>` `>=` `<` `<=`). Operators useful for filtering and correlating fields include `convolve`, `crossCorrelate`, `FFT`, among others. Many other operators for changing field sizes (`subsample`, `upsample`), nonlinear mappings (`warp`) and many other special functions are supplied.

**Cog is implemented in the Scala programming language, so Cog applications are also written in Scala. Here's a very simple Cog application with one dynamic field and one operator:**

```scala
val counter = new DynamicScalarField
counter <=\= counter + 1
```

The dynamic field, named `counter`, is zero-dimensional containing a single real number that, by default, is initialized to zero. The operator here is “`+ 1`” which takes the output of `counter` as its input. The line containing the `<=\=` symbol defines how `counter` evolves in time and may be read as “the next state of `counter` will be the current state of `counter` with the operator ‘`+ 1`’ applied to it.” In other words, this is nothing more than a state machine holding a floating point number that is initialized to zero and incremented each clock cycle.

Here’s a slightly more complicated example that takes two dynamic color fields and “averages” them (admittedly not a particularly useful thing to do):
val hummingbird: DynamicColorField = ...
val butterfly: DynamicColorField = ...
val average = new DynamicColorField(...)
average <= (hummingbird + bufferfly) / 2

This program yields the following color video streams when executed:

![hummingbird](image1.png) ![butterfly](image2.png) ![average](image3.png)

Since it’s frequently the case that the definition of a dynamic field is followed by an operator expression defining the next state for that field, Cog allows the two operations to be combined. For example, the `average` field in the previous example could have been written as:

```scala
val average = DynamicColorField((hummingbird + bufferfly) / 2)
```

Using that shortcut, here’s a program that blurs a video stream with a Gaussian filter:

```scala
val butterfly: DynamicScalarField = ...
val blurred = DynamicScalarField(butterfly convolve Gaussian(5.0f))
```

producing the following streams:

![butterfly](image4.png) ![blurred](image5.png)

A final example that illustrates learning in Cog implements the application shown at the very beginning of this paper: learning to repair defective video streams due to static occlusions or dead pixels. The algorithm is simple: look for pixels that don’t change over time. When such pixels are found, assume that they are “dead” and need to be filled in. The filling-in algorithm is isotropic diffusion, using the non-dead pixels as Dirichlet (fixed) boundary conditions. Here’s the code:
val input: DynamicColorField = ...
val delayedInput = DynamicColorField(input)
val delta = DynamicColorField((delayedInput - input).abs)

val motion = new DynamicScalarField(ScalarField.random(...))
motion <= motion * 0.9975f + (motion * -1f + 1f) *
  (delta.red + delta.green + delta.blue) * 0.5f

val stationaryPixels = DynamicField(motion > 0.5f)
val retinalFilling = DynamicColorField(input.diffuseDirichlet(colorField(stationaryPixels)))

and another generated video stream from the `retinalFilling` dynamic field as it evolves over time, initially doing no filling in:

![Generated video streams](image1.png)  ![Generated video streams](image2.png)  ![Generated video streams](image3.png)

**Cognitive Library**

The Cog distribution comes with a library of modules implemented with Cog primitives that supplies commonly needed functionality such as various filter banks, local polynomial expansions, boundary completion, phase congruency, a Poisson solver, plus many others. Here’s an example of the phase congruency module extracting edges from a video stream:

![Extracted edges](image4.png)  ![Extracted edges](image5.png)

Another module implements the retinex algorithm, useful for compressing large dynamic ranges of light intensity without losing details in the brightest or most shaded regions:
Compiler and Runtime System
Compilation of an application model into code that can run on multiple GPUs distributed across a network is done dynamically. When a Cog application is started, the user's model of interacting dynamic fields is parsed, dynamically translated to GPU code, optimized, partitioned and placed onto available GPU resources, and downloaded into the GPUs for execution. The Cog runtime system coordinates the GPUs, exchanging dynamic field data between GPUs on different network nodes, and orchestrates the computation.

Debugging
Cog has a graphical debugger that allows an application to be probed at runtime; there is no need to specify which fields will be displayed when the application is compiled:
The graphical display on the left shows the dynamic field structure of the application, automatically extracted from the source code. Clicking on a box representing a field causes a window to pop up on the right displaying the state of that field as it evolves in time. The command bar across the top lets the user control execution by running, stopping or single-stepping. The state of the application may be saved to a file (useful to preserve learning) and restarted later from the saved state.

Brains
The largest application built on Cog so far is a simple "brain" named "MoNETA," jointly developed with our partners at Boston University. MoNETA is designed as a modular research platform for building increasingly complex brains, and it's first challenge was to learn to solve a problem in behavioral psychology.

The Morris Water Maze is a classic psychology experiment performed on mice. A mouse is thrown into a tank of water that contains a small platform just below the water's surface. Although the mouse cannot see the platform, it learns after a small number of trials to head directly to the platform, even though it is thrown into the tank at a different random location for each trial. (Swimming takes a lot of energy, so the mouse is motivated to find a resting place). Psychologists have learned that the mouse uses visual cues from the environment outside of the tank to deduce its location and plan its "escape."
The following figure shows the high level structure of the MoNETA brain. MoNETA includes a sensory system (pink), emotion system (yellow), and planning and navigation system (green). Gray blocks are submodules with labeled function (top) and corresponding biological brain region (bottom).
The figure on the right shows learning in the MoNETA brain in a simulated water maze. On its first trial, the virtual mouse panics and explores its environment, with a preference to explore unfamiliar regions. With each repeated trial, it rapidly improves its ability to navigate more quickly to the safety of the submerged platform (cross-hatched green circle).

Further information is available online: http://nl.bu.edu/research/projects/moneta

**Other Approaches**

Traditional parallel programming approaches, such as multiple threads combined with a synchronization mechanism (e.g. semaphores) will not scale to millions of cores. They are difficult and tedious to program and are prone to races and deadlock. They are also nondeterministic, making verification and debugging problematic. They require a shared-memory architecture that is difficult to emulate in a distributed, networked environment, and thus must be combined with messaging system of some kind to extend the computation beyond a single network node.

The MPI system (Message Passing Interface) is a widely-used library that enables processes distributed across a network to communicate by exchanging messages. Although powerful, the programming model is single-threaded communicating processes, thus providing no fine-grain, concurrency control needed to exploit the increasing number of cores available on processors today.

The Actor programming model eliminates shared memory, so that Actors (threads that do not share memory) can be distributed across a network. Although simple and abstract, actors also lack the needed fine-grain concurrency control and require explicit programming to achieve deterministic computation if that’s desired.

Transactional memory addresses some of the complexity of programming multi-threaded applications, but requires considerable computational overhead to ensure coherence of data structures. It is a shared-memory programming paradigm, and therefore must be augmented with a messaging system for communication across a network.

CUDA and OpenCL are very low-level systems for programming GPUs. Although they expose GPU parallelism, they require familiarity with GPU hardware architecture, and require
explicit implementation some low-level mechanisms, such as memory caching, normally taken for granted on CPUs.

**Summary**

Cog ex Machina is a pragmatic framework for developing applications requiring massive parallelism, particularly “cognitive” applications that must learn through interactions with the world. The targeted hardware platforms are networked systems of commodity, multicore processors, ranging from mobile devices that will contain thousands of cores, to workstations containing hundreds of thousands of cores, to servers that will contain millions of cores. Its state-machine programming paradigm is: (1) *deterministic*—free from races and deadlocks; (2) *minimalist*—uses only two abstractions, fields and operators, to express computations; (3) *abstract*—frees the developer from the complexities of the hardware platform (threads, cores, synchronization, communication).