Cognitive Computing Programming Paradigm: A Corelet Language for Composing Networks of Neurosynaptic Cores

Arnon Amir, Pallab Datta, William P. Risk, Andrew S. Cassidy, Jeffrey A. Kusnitz,
Steve K. Esser, Alexander Andreopoulos, Theodore M. Wong, Myron Flickner,
Rodrigo Alvarez-Icaza, Emmett McQuinn, Ben Shaw, Norm Pass, and Dharmendra S. Modha
IBM Research - Almaden, San Jose, CA 95120

***Abstract*—Marching along the DARPA SyNAPSE roadmap, IBM unveils a trilogy of innovations towards the TrueNorth cognitive computing system inspired by the brain's function and efficiency. The sequential programming paradigm of the von Neumann architecture is wholly unsuited for TrueNorth. Therefore, as our main contribution, we develop a new programming paradigm that permits construction of complex cognitive algorithms and applications while being efficient for TrueNorth and effective for programmer productivity. The programming paradigm consists of (a) an abstraction for a TrueNorth program, named *Corelet*, for representing a network of neurosynaptic cores that encapsulates all details except external inputs and outputs; (b) an object-oriented *Corelet Language* for creating, composing, and decomposing corelets; (c) a *Corelet Library* that acts as an ever-growing repository of reusable corelets from which programmers compose new corelets; and (d) an end-to-end *Corelet Laboratory* that is a programming environment which integrates with the TrueNorth architectural simulator, Compass, to support all aspects of the programming cycle from design, through development, debugging, and up to deployment. The new paradigm seamlessly scales from a handful of synapses and neurons to networks of neurosynaptic cores of progressively increasing size and complexity. The utility of the new programming paradigm is underscored by the fact that we have designed and implemented more than 100 algorithms as corelets for TrueNorth in a very short time span.**

I. Introduction

A. Context

To usher in a new era of cognitive computing [1], we are developing TrueNorth (Fig. 1), a non-von Neumann, modular, parallel, distributed, event-driven, scalable architecture—inspired by the function, low power, and compact volume of the organic brain. TrueNorth is a versatile substrate for integrating spatio-temporal, real-time cognitive algorithms for multi-modal, sub-symbolic, sensor-actuator systems. TrueNorth comprises of a scalable network of configurable neurosynaptic cores. Each core brings memory ("synapses"), processors ("neurons"), and communication ("axons") in close proximity, wherein inter-core communication is carried by all-or-none spike events, sent over a message-passing network.

Recently, we have achieved a number of milestones: first, a demonstration of 256-neuron, 64k/256k-synapse neurosynaptic cores in 45nm silicon [2], [4] that were featured on the cover of *Scientific American* in December 2011; second, a demonstration of multiple real-time applications [5]; third, Compass, a simulator of the TrueNorth architecture, which simulated over 2 billion neurosynaptic cores exceeding 10^{14} synapses [3], [6]; and, fourth, a visualization of the long-distance connectivity of the Macaque brain [7]—mapped to TrueNorth architecture—that was featured on the covers of *Science* [8] and *Communications of the ACM* [1].

We unveil a series of interlocking innovations in a set of three papers. In this paper, we present a programming paradigm for hierarchically composing and configuring cognitive systems that is effective for the programmer and efficient for the TrueNorth architecture. In two companion papers [9][10] we introduce a versatile and efficient digital spiking neuron model that is a building block of the TrueNorth architecture, as well as a set of algorithms and applications that demonstrate the potential of the TrueNorth architecture and value of the programming paradigm.

B. Motivation

Turing-completeness bounds the computational expressiveness of programmed systems. ENIAC (circa 1946), the first electronic digital, programmable, Turing-complete machine, was the inspiration behind the formulation of the von Neumann architecture [11]. Driven by dual and often conflicting objectives of machine and programmer efficiency, the programming paradigm has evolved from machine code, to assembly language, to high-level languages. High-level languages prevalent today trace their genesis to FORTRAN [12]. As noted by Backus et al. [13], "The fundamental unit of a program is the basic block; a basic block is a stretch of program which has a single entry point and a single exit point." This implies that a program for a von Neumann architecture is fundamentally a linear or sequential construct.

Like von Neumann machines, TrueNorth is Turing-complete [9]. However, they are complementary in that each is efficient for different classes of computation. A TrueNorth *program* is a complete specification of a network of neurosynaptic cores, and all external inputs and outputs to the network, including the specification of the physiological properties (neuron parameters, synaptic weights) and the anatomy (inter- and intra-core connectivity). The job of a TrueNorth *programmer* is to translate a desired computation into a specification that efficiently executes on TrueNorth, namely, a completely specified network of neurosynaptic cores, its inputs, and its outputs. In this context, the linear programming paradigm of the von Neumann architecture is not ideal for TrueNorth programs. Therefore, we set out to develop an entirely new programming paradigm that can permit construction of complex cognitive algorithms and applications while being efficient for TrueNorth and effective for programmer productivity.

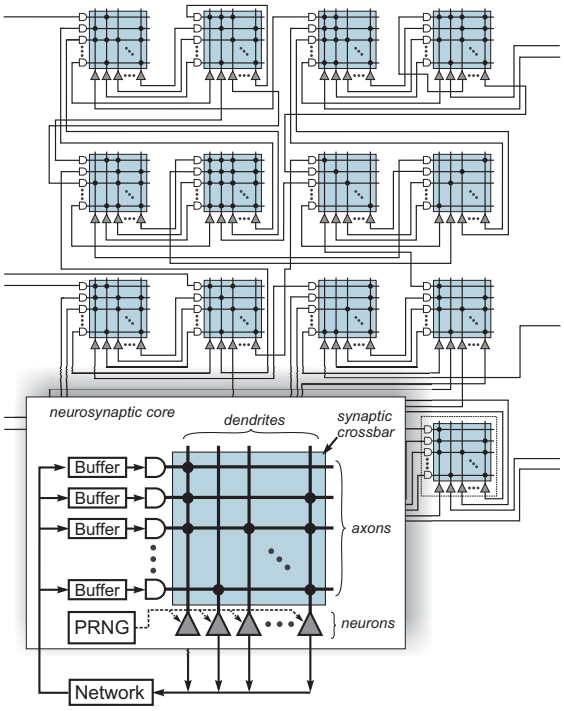

Fig. 1. TrueNorth is a brain-inspired chip architecture built from an interconnected network of lightweight neurosynaptic cores [2], [3]. TrueNorth implements "gray matter" short-range connections with an intra-core crossbar memory and "white matter" long-range connections through an inter-core spike-based message-passing network. TrueNorth is fully programmable in terms of both the "physiology" and "anatomy" of the chip, that is, neuron parameters, synaptic crossbar, and inter-core neuron-axon connectivity allow for a wide range of structures, dynamics, and behaviors. *Inset:* The TrueNorth neurosynaptic core has 256 axons, a 256×256 synapse crossbar, and 256 neurons. Information flows from axons to neurons gated by binary synapses, where each axon fans out, in parallel, to all neurons thus achieving a 256-fold reduction in communication volume compared to a point-to-point approach. A conceptual description of the core's operation follows. To support multi-valued synapses, axons are assigned types which index a synaptic weight for each neuron. Network operation is governed by a discrete time step. In a time step, if the synapse value for a particular axon-neuron pair is non-zero and the axon is active, then the neuron updates its state by the synaptic weight corresponding to the axon type. Next, each neuron applies a leak, and any neuron whose state exceeds its threshold fires a spike. Within a core, *PRNG* (pseudorandom number generator) can add noise to the spike thresholds and stochastically gate synaptic and leak updates for probabilistic computation; *Buffer* holds incoming spikes for delayed delivery; and *Network* sends spikes from neurons to axons.

C. Contributions

As stated earlier, a TrueNorth *program* is a complete specification of a network of neurosynaptic cores, along with its external inputs and outputs. As the size of the network increases, to completely specify such a network while being consistent with TrueNorth architecture becomes increasingly difficult for the programmer. To help combat the complexity, we propose a divide-and-conquer approach whereby a large network of neurosynaptic cores is constructed by interconnecting a set of smaller networks of neurosynaptic cores, where each of the smaller networks, in turn, could be constructed by interconnecting a set of even smaller networks, and so on, until we reach a network consisting of a single neurosynaptic core, which is the fundamental, non-divisible building block.

To this end, as our fundamental contribution, we develop a new programming paradigm that consists of (a) a *corelet*, namely an abstraction that represents a TrueNorth program that only exposes external inputs and outputs while encapsulating all other details of the network of neurosynaptic cores; (b) an object-oriented *Corelet Language* for creating, composing, and decomposing corelets; (c) a *Corelet Library* that acts as an ever-growing repository of reusable corelets from which to compose new corelets; and (d) an end-to-end *Corelet Laboratory* that is a programming environment that integrates with the TrueNorth architectural simulator, called Compass [3], and supports all aspects of the programming cycle from design, through development, debugging, and into deployment.

Corelets, Composition, and Decomposition (Sec. II): A *corelet* (Fig. 2) is an abstraction of a network of neurosynaptic cores that *encapsulates* all intra-network connectivity and all intra-core physiology and only *exposes* external inputs to and external outputs from the network. We group inputs and outputs into *connectors*. A corelet user has access only to input and output connectors.

Given a set of corelets, *composition* is an operation for creating a new corelet. For ease of exposition, we refer to the constituent corelets as *sub-corelets*. Fig. 2(d)-(f) illustrates three key steps of composition: (a) interconnect some of the outputs of the sub-corelets to some of the inputs of the sub-corelets; (b) encapsulate all intra-corelet connectivity; and (c) expose only external inputs to and external outputs from the corelet. The process of composition can be hierarchically repeated to create progressively more complex corelets. Therefore, it is possible to think of any corelet as a *tree* of sub-corelets, where the leaves of the tree constitute individual neurosynaptic cores.

The corelet abstraction is designed to boost programmer productivity, but cannot be directly implemented on TrueNorth. Given a corelet, *decomposition* (Fig. 3) is the logical inverse of composition, that is, it is an operation for removing the nested tree structure and for removing all layers of encapsulation to produce a network of neurosynaptic cores that can be implemented on the TrueNorth architecture, either in simulation or hardware.

Corelet Language (Sec. III): The fundamental *symbols* of the language are the neuron, neurosynaptic core, and corelet. The connectors constitute the *grammar* for composing these symbols into TrueNorth programs. Together, the symbols and the grammar are both necessary and sufficient for expressing any TrueNorth program. We implement these primitives in object-oriented methodology.

Corelet Library (Sec. IV): The library is a repository of consistent, verified, parameterized, scalable and composable functional primitives. To boost programmer productivity, we have designed and implemented a repository of more than 100 corelets in less than one year. Every time a new corelet is written, either from scratch or by composition, it can be added back to the library, which keeps growing in a self-reinforcing way. Further, by virtue of composability, the expressive capability of the library grows exponentially as some power, > 1, of its size.

Corelet Laboratory (Sec. V): Eventually, a corelet must be decomposed and implemented on TrueNorth, either in

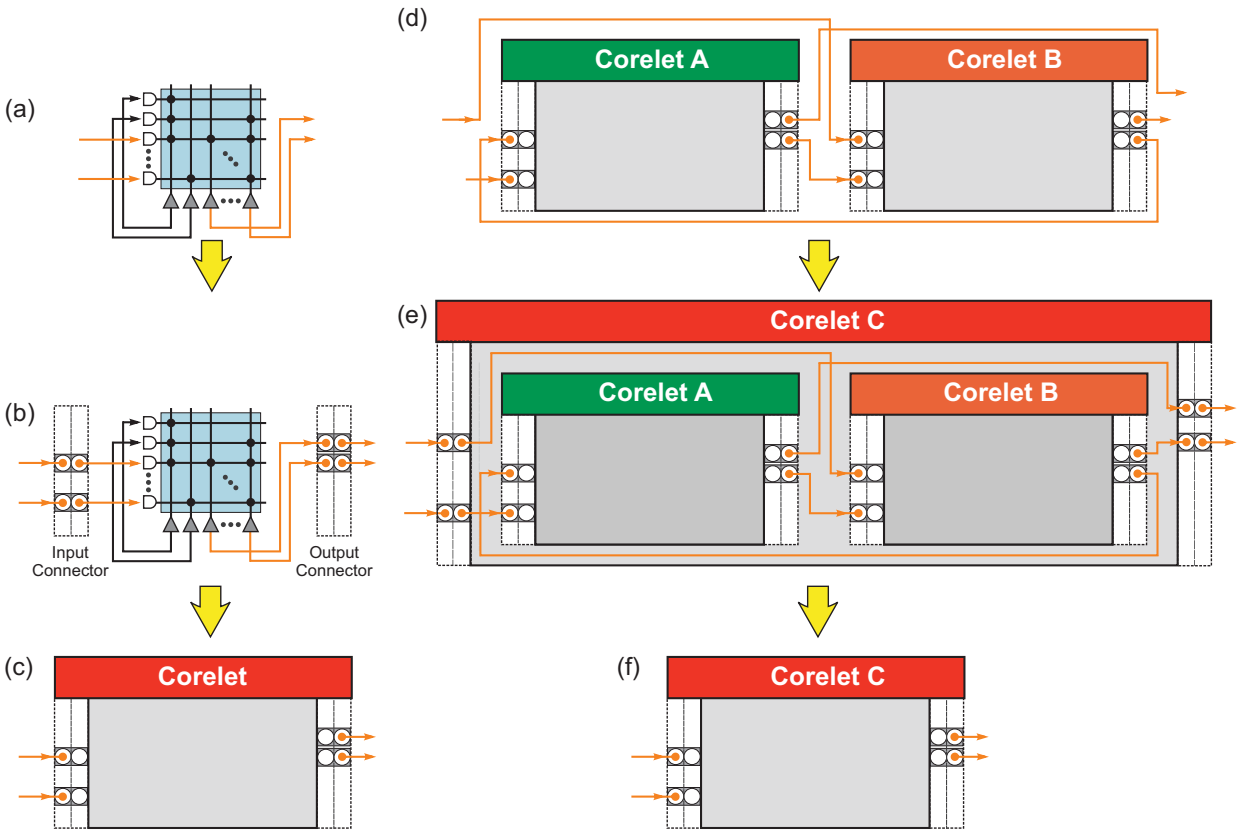

Fig. 2. **Panels (a), (b), and (c) illustrate construction of a *seed corelet* while panels (d), (e), and (f) illustrate construction of a corelet via *composition* of two sub-corelets.** (a) Create recurrent connections, by connecting a set of neurons on the core with a set of axons on the core. Configure the synaptic crossbar to connect axons to neurons. A neuron is a *source* of spikes and an axon is a *destination* of spikes. (b) Unconnected axons, that receive spikes from outside the core, are grouped into an enumerated list known as an *input connector*. Unconnected neurons, that send spikes outside the core, are grouped into an enumerated list known as an *output connector*. (c) The seed corelet encapsulates the intra-core neuron-axon connectivity, synaptic crossbar, and neuron parameters while exposing the input and output connectors. The corelet developer sees all corelet internals, but a corelet user only sees the exposed external interfaces. (d) Create connections between sub-corelets by interconnecting a set of output connector pins to a set of input connector pins. (e) Unconnected input connector pins, that receive spikes from outside the composed corelet, are grouped into a new input connector. Unconnected output connector pins, that send spikes outside of the composed corelet, are grouped into a new output connector. (f) The composed corelet encapsulates the sub-corelets and the internal connectivity between sub-corelets, while exposing the new input and output connectors. The developer of the composed corelet sees all the internals of the composed corelet but not the internals of the sub-corelets due to encapsulation. However, a user of the the composed corelet only sees the exposed external interfaces.

hardware or simulation, and interact with the environment via event-driven, spiking sources (for example, sensors) and destinations (for example, actuators) that connect to it. To facilitate this process, the Corelet Laboratory provides a complete end-to-end framework.

The value of the new paradigm to the programmer is: freedom from thinking in terms of low-level hardware primitives; availability of tools to design at the functional level; ability to use a divide-and-conquer strategy in the process of creating and verifying individual modules separately; a new way of thinking in terms of simple modular blocks and their hierarchical composition, rather than having to deal with an unmanageably large network of neurosynaptic cores directly; guaranteed implementability on TrueNorth; ability to verify correctness, consistency, and completeness; ability to reuse code and components; ease of large-scale collaboration; ability to configure more neurosynaptic cores per line of code and unit of time; access to an end-to-end environment for creating, compiling, executing, and debugging; and the ability to use the same conceptual metaphor across functional blocks that range from a handful of synapses and neurons to networks of neurosynaptic cores with progressively increasing size and complexity.

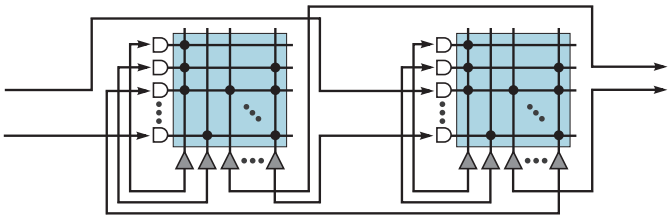

Fig. 3. Example of corelet *decomposition*. Assuming that "Corelet A" and "Corelet B" in panel (d) of Fig. 2 are both instances of the "Corelet" in panel (c) of Fig. 2, the composed corelet shown in panel (f) of Fig. 2 is decomposed by progressively removing all layers of encapsulation to produce a network of neurosynaptic cores along with its external inputs and outputs, resulting in a TrueNorth *program*. The program can be executed on TrueNorth hardware as well as simulated using Compass [3].

II. Corelet Definition

A. Seed Corelet

A *seed corelet* is a TrueNorth program consisting of a single neurosynaptic core that exposes only inputs and outputs to the core while encapsulating all other details, including neuron parameters, synaptic weights, and intra-core connectivity. The *corelet programmer* specifies both the internal details and external interfaces, while the *corelet user* uses only external interfaces. Panels (a), (b), and (c) of Fig. 2 show an example of seed corelet construction.

A neuron is a *source* of spikes, and an axon is a spike *destination*. For axons that receive inputs from neurons within the same seed corelet, the corelet programmer can unambiguously specify and pair these axons/neurons together during corelet development. For axons that receive inputs from outside their seed corelet, the source neuron is unknown to the corelet programmer at development time. These axons are grouped into an enumerated list known as an *input connector*. The sources of these axons will be specified only later, when a user instantiates and connects this corelet.

Similarly, for neurons that send outputs to axons within the same seed corelet, the corelet programmer can unambiguously specify and pair these axons/neurons together during corelet development. For neurons that send output outside this seed corelet, the destination is unknown to the corelet programmer at development time. These neurons are grouped into an enumerated list known as an *output connector*. The destinations of these neurons will be specified only later, when a user instantiates and connects this corelet.

B. Corelet

As mentioned earlier, we can *compose* a set of corelets into a new corelet. Through this process, the original corelets become the *sub-corelets* of the newly composed corelet. For all sub-corelets, their output connectors are spike *sources* and their input connectors are spike *destinations*.

Composition is illustrated in panels (d), (e), and (f) of Fig. 2. First, we create connections between sub-corelets, connecting some of the source and destination pins. Second, some of the sub-corelet's input pins that were not connected are grouped into the composed corelet's input connector, and some of the sub-corelet's output pins that were not connected are grouped into the composed corelet's output connector. The new corelet's connectors are exposed, but the sub-corelets and their local connectivity are encapsulated by the composed corelet. In general, corelets can be composed from both neurosynaptic cores and sub-corelets.

Decomposition is the logical inverse of composition. It operates on a composed corelet, removing the hierarchical structure layer by layer until a flat network of cores is obtained, which in turn can be expressed as a TrueNorth Program and written into a model file. Composition and decomposition are described in detail later in Section III-E.

III. Corelet Language

Why Object-Oriented Methodology?

From the perspective of a language designer, object-oriented programming (OOP) is the ideal method for implementing corelets for at least three reasons.

First, by definition, a corelet encapsulates all the details of a TrueNorth program except for external inputs and outputs. Encapsulation is also a fundamental feature of OOP. Therefore, corelet encapsulation can be guaranteed by defining a *corelet class* and then instantiating corelets as *objects* from this class.

Second, all corelets must use similar data structures and operations, and must be accessed by users in similar ways. This similarity can be achieved by another fundamental feature of OOP, *inheritance*, which allows the underlying data structures and operations to be defined once for an abstract *class* and then passed down to abstract *subclasses* derived from it, as well as to *object* instances of the class.

Third, we need to invoke operations such as "decompose" (to translate them into a TrueNorth program) and "verify" (to ensure that they are correct and consistent with respect to TrueNorth) on all corelets. Each operation is named homogeneously across mutiple corelets, but can be heterogeneously defined for different corelets. These operations can be implemented by *polymorphism* - another fundamental feature of OOP.

Therefore, defining a corelet as a class in an OOP framework grants us encapsulation, inheritance, and polymorphism; and dramatically improves the design, structure, modularity, correctness, consistency, compactness, and reusability of code. Starting from a base corelet class, different corelets can be written as sub-classes, and different concrete corelets would be *object* instances of these classes[1]. We have implemented the Corelet Language using MATLAB OOP, which has the additional advantage of being a compact language for matrix and vector expressions and computations.

The language is composed of four main classes and their associated methods. The classes are the neuron, neurosynaptic core, connector, and corelet. While the implementation uses a number of innovative data structures and optimizations, in what follows we present only the most essential ideas for the sake of brevity and for ease of exposition.

A. The Neuron Class

Per the neuron model in [9], the *Neuron* class contains all of the properties of the TrueNorth neuron model, such as initial membrane potential, thresholds, leaks, and reset modes. The neuron's `get()` and `set()` methods, used for setting and retrieving these properties, ensure that all values are compatible with TrueNorth. For convenience, we provide default values for a set of commonly used neuron behaviors.

[1] At this point, it is noteworthy to make a distinction between two different trees. The hierarchical composition of corelets from sub-corelets must not be confused with the hierarchy of corelet classes. The former is a tree of corelet *objects*, formed in the computer memory when the corelet code is executed. Each corelet object keeps handles (references, like pointers) to its sub-corelets, hence forming a tree. The later is a tree of code, with a child class inheriting properties and methods from their parent class, which comprises the *Corelet Library*.

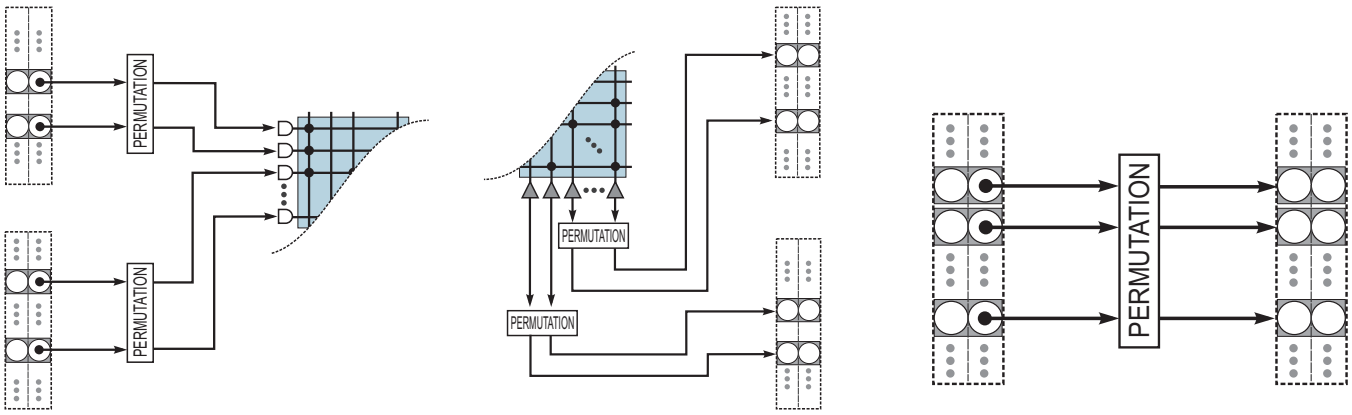

Fig. 4. The three connectivity patterns between connectors and cores are illustrated, with arrows indicating the flow of spikes. These three connectivity patterns, when combined, allow to create any network of TrueNorth neurosynaptic cores. (a) Connecting the destination side of some pins in two connectors to some input axons on a core. (b) Connecting the source side of some pins in two connectors to some output neurons on a core. (c) Connecting the destination side of connector C_1 to the source side of connector C_2 (written in Corelet Language as `C1.busTo(C2,P)`, where the permutation vector `P` may be omitted when equal to the identity permutation).

B. The Core Class

The core class models the TrueNorth neurosynaptic core. The essential properties of the core class include: a vector of 256 axons that stores the axon types; a vector of 256 neuron objects, instantiated from the neuron class; a vector of 256 target axonal destinations, one per neuron; and a 256×256 matrix, representing the binary connections in the synaptic crossbar. In addition to the usual `get()` and `set()` methods, we provide methods to ensure that all values are compatible with TrueNorth. We also provide methods for assigning connectivity between neuron-axon pairs.

C. The Connector Class

Each corelet assigns a unique local identifier to each of its constituent neurosynaptic cores and sub-corelets. Each axon and each neuron on a core is assigned a unique identifier with respect to the core. Similarly, each external output of a corelet and each external input to a corelet is assigned a unique identifier with respect to the corelet.

We identify the a^{th} axon on a core, named $core$, as a *destination address* by writing $[a, core]$ and we identify the n^{th} neuron on $core$ as a *source address* by writing $[n, core]$. Similarly, we write the destination address of the a^{th} external input on corelet C as $[a, C]$ and the source address of the n^{th} external output on corelet C as $[n, C]$.

Suppose we have a set of neurosynaptic cores and corelets. A *pin* is either an external input or an external output of a core or a corelet; it holds a source address and a desitination address, as well as a true or false *state* for each address that indicates if its value has been specified. Before the composition process, the pin's source and destination addresses are not known. Therefore, we mark an uninitialized pin that corresponds to the p^{th} input or output of core or corelet C as $([p, C], [p, C])$, and set the pin's *state* to (false,false). Starting with this uninitialized state, via the composition process, each pin attains a final state such that its source and destination addresses are completely specified, for example $([n, C_1], [a, C_2])$, and its state is (true, true).

To connect an uninitialized pin that corresponds to the p^{th} external input of a core or a corelet C to the a^{th} input of core or corelet C_2, we update the pin's value to $([p, C], [a, C_2])$ and its state to (false, true). Specifically, we update the pin's destination address and destination state. Similarly, to connect an uninitialized pin that corresponds to the p^{th} external output of a core or a corelet C to n^{th} input of core or corelet C_1, we update its value as $([n, C_1], [p, C])$ and its state as (true, false). Specifically, we update the pin's source address and source state.

Given an output pin p of the form $([n, C_1], [p, C_2])$ and state as (true, false), meaning that its destination address is unknown, and an input pin 1 of the form $([q, C_3], [a, C_4])$ and state (false, true), meaning that its source address is unknown, the process of *pairing* them involves updating the states as follows: We update the value of pin p as $([n, C_1], [q, C_3])$ and the value of pin q as $([p, C_2], [a, C_4])$. Then we update the states of both pins to (true, true).

To summarize, whenever a pin is connected to a destination, its destination address and state are updated. Similarly, whenever a pin is connected to a source, its source address and state are updated. When a pin connects to another pin, these operations happen in tandem on both the pins.

The properties of the *connector class* consist of an ordered list of pins, where each pin has a tuple of two addresses and a state as defined above. The connector class offers several connectivity methods for connecting cores to connectors as well as for connecting connectors to other connectors, as shown in Fig. 4.

The inter-connectivity between neurosynaptic cores can be thought of as a bipartite graph between neurons and axons. With N neurons and axons, there are $N!$ possible connectivity patterns. Direct specification of such complex neuron-axon wiring does not scale and is highly error-prone. In this context, the connector is an extremely powerful primitive that provides a semantic representation of connectivity.

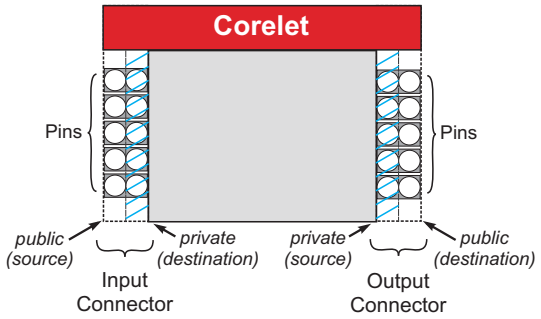

Fig. 5. A corelet is an abstraction of a network of neurosynaptic cores. The corelet has an *input* connector and an *output* connector. Each connector has a *public side* and an encapsulated *private side*. The figure also shows *pins* on each connector. On the private side, pins of the input connector connect to input axons within the network of neurosynaptic cores, and pins of the output connector connect to output neurons in the network of neurosynaptic cores.

D. The Corelet Class

The properties of the *corelet class* constitute sub-corelets, neurons, cores, input connectors, and output connectors. While all properties of a corelet are *private*, the source address of its input connector and the destination addresses of its output connector are *public*. To facilitate composition, the corelet class offers methods for: (a) recursively creating constituent sub-corelet objects; (b) creating neuron types; (c) creating constituent core objects; (d) creating input and output connectors; and (e) interconnecting the connectors, cores, and sub-corelets within the corelet. Since corelets can be composed hierarchically, several of the corelet methods are built to recursively traverse corelet trees and operate on them. Finally, the corelet class offers a method for decomposition that recursively travels the corelet object tree to remove all layers of abstractions and produce a TrueNorth program. Corelet composition and decomposition are illustrated in Fig. 6. As described later in Sec. V, we use the decomposition method to output a *model file* in a JSON mark-up language. The file is then used to run the TrueNorth program on the Compass Simulator.

The key design criteria is to ensure that all TrueNorth programs are expressible in the corelet language, and, conversely, any corelet that the language allows is TrueNorth compatible. In other words, the language should be complete and consistent with respect to the TrueNorth architecture. To this end, we have identified a set of invariants: (a) a corelet has all its constituent cores, neurons, and synaptic crossbars fully specified and configured. (b) a corelet has all its cores, sub-corelets and connectors fully connected; (c) each neuron in a corelet is either assigned with a destination address of an axon, is defined as *disconnected*, or is connected to a pin in an output connector; (d) each axon in a corelet is either connected to a source neuron, is defined as *disconnected*, or is connected to a pin in an input connector. Given the recursive nature of a corelet, all these invariants are recursively and hierarchically asserted and enforced throughout.

E. Composition and Decomposition

The connectivity within the corelet and its sub-corelets is constructed in two phases by the process of *composition* and *decomposition*. The process of composition creates a doubly linked path between each source neuron in the system and its corresponding destination axon — while following and preserving corelet encapsulation. The process of decomposition (a logical inverse of composition) removes all layers of indirection along the path one by one and eventually replaces the entire path with a direct link between the source neuron and its target axon. This process creates a TrueNorth program, expressed in terms of a directly connected network of neurosynaptic cores without any encapsulation. The two processes are illustrated in Fig. 6, using an example that we will fully develop in Sec. V.

The recursive process of composition takes place during the recursive corelet construction process. As seen in the figure, it creates a doubly linked path between the source neuron $[n_1, core_1]$ and the destination axon $[a_2, core_2]$ which can be described by the list of pins along the path, namely $[n_1, core_1] \rightleftarrows [p_1, C_1] \rightleftarrows [p_2, C_2] \rightleftarrows [p_3, C_3] \rightleftarrows [p_4, C_4] \rightleftarrows [a_2, core_2]$. Note that each pin along the list is linked to the pin on its left and the pin on its right, therefore creating a doubly linked path, while traversing through multiple corelets and preserving encapsulation. It can be seen that within a corelet, due to encapsulation, only its sub-corelets can connect with one another. At the end of the composition process, all source neurons are connected via paths to their corresponding destination axons.

The recursive process of decomposition traverses the composed corelets, processing from the bottom up (depth-first order), removing one pin from the path at a time. Starting to operate in Corelet L, it removes $[p_1, C_1]$ from the path by connecting $[n_1, core_1]$ to $[p_1, C_1]$ and passing its cores to its parent, Corelet LSM. As a result, the path becomes $[n_1, core_1] \rightleftarrows [p_2, C_2] \rightleftarrows [p_3, C_3] \rightleftarrows [p_4, C_4] \rightleftarrows [a_2, core_2]$, and the encapsulation of Corelet L is removed. This process continues until all the cores are passed to Corelet MCR and the remaining path is the direct connection $[n_1, core_1] \rightleftarrows [a_2, core_2]$.

IV. Corelet Library

The *Corelet Library* is the foundation from which new systems and new corelets are composed (Fig. 7). It contains all the corelets that are available, and continuously grows as more corelets are developed. Any corelet from the library can be used for building a new system, and the new system itself can then be added back into the library in the form of a new corelet, thereby growing and enriching the library in a bottom-up fashion. In less than a year since we started to use the Corelets Language, the team has developed more than 100 corelets.

All corelets are sub-classes of the corelet base class, so the Corelet Library is also a tree of sub-classes. A general-purpose corelet, such as a linear filter corelet, can then have sub-classes such as a Gabor filter corelet, non-linear filters, and recursive IIR filters. In our companion paper, we describe in detail several applications that use this hierarchical approach to build functional systems [10].

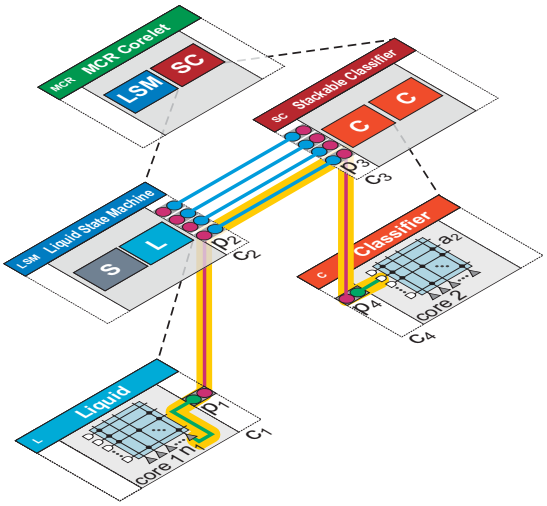

Fig. 6. The processes of *composition* and *decomposition* are illustrated in details for the assignment of the destination address $[a_2, core_2]$ to neuron $[n_1, core_1]$ in the TrueNorth program constructed by the Corelet MCR. Composition takes place during the recursive corelet construction. Corelet MCR starts by creating Corelet LSM, which in turn creates Corelets L and S (S not shown for clarity). Corelet L, when created, connects output neuron $[n_1, core_1]$ to pin $[p_1, C_1]$. Next, Corelet LSM connects connector C_1 to C_2, thereby connecting $[p_1, C_1]$ and $[p_2, C_2]$. Similarly, Corelet C, when constructed, connects $[p_4, C_4]$ to input axon $[a_2, core_2]$. Next, Corelet SC connects $[p_4, C_4]$ with $[p_3, C_3]$. Finally, after both Corelets LSM and SC are constructed and composed, Corelet MCR connects connector $C2$ with connector $C3$, thereby connecting $[p_2, C_2]$ with $[p_3, C_3]$ and completing the doubly linked path between neuron $[n_1, core_1]$ and axon $[a_2, core_2]$, captured by the four pins in the four connectors along the path. The consequent recursive *decomposition* process traverses the corelets in the same order. At each corelet, it removes its own pin from the path by directly connecting the pin referenced by its source address with the pin referenced by its destination address, hence reducing the path length by one. When the decomposition process completes, all connector pins are removed from the path and source neuron $[n_1, core_1]$ is connected directly to its destination axon $[a_2, core_2]$, as part of the TrueNorth program.

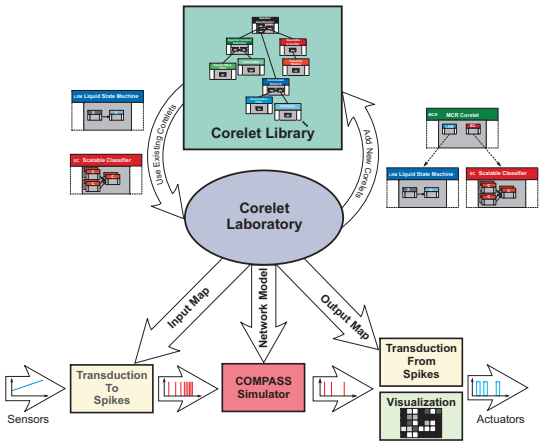

Fig. 7. The Corelet Laboratory. We depict the complete development cycle with all the tools. The *Corelet Library* is a hierarchical library of all corelets created by developers. The Corelet development process typically involves creating and inheriting from the *Corelet Library*, composing sub-corelets together into new corelets, and submitting the new corelet into the library after verification. The new corelet then becomes available for use by other applications, thus allowing composition. A concrete application is created by instantiating objects from the created corelet class. The instantiated object is then decomposed into a network model file that represents a TrueNorth program. This, in turn, is used to setup the Compass simulator. The external input and output connectors are used, respectively, to create input and output map files that show where sensory input should be targeted and from where the output is to be received. During execution, the simulator receives sensor transduced input spikes and generates output spikes that can be used to interpret classification, drive actuators, or create visualizations.

The corelets currently in the Corelet Library include scalar functions, algebraic, logical, and temporal functions, splitters, aggregators, multiplexers, linear filters, kernel convolution (1D, 2D and 3D data), finite-state machines, non-linear filters, recursive spatio-temporal filters, motion detection, optical flow, saliency detectors and attention circuits, color segmentation, a Discrete Fourier Transform, linear and non-linear classifiers, a Restricted Boltzmann Machine, a Liquid State Machine, and more. The corelet abstraction and unified interfaces enable developers to easily replace a library corelet with an alternative implementation without disrupting the rest of the system.

V. Corelet Laboratory

Now that we have all the pieces together, we present an end-to-end *Corelet Laboratory*, the programming environment that integrates with the TrueNorth architectural simulator, Compass [3], and supports all aspects of the corelet programming cycle from design, through development, debugging, and into deployment, as shown in Fig. 7.

A. Sample Application: Music Composer Recognition

We now illustrate the Corelet Laboratory using a concrete application. Given a musical score by Bach or Beethoven, consider the problem of identifying the music piece's composer. We adopt the approach of using a liquid state machine [14] in conjunction with a hierarchical classifier. In this paper, our focus is not so much the application itself but the programming paradigm used to create it. The reader interested in more details about the application can consult the companion paper [10].

The corelet developed for the application, named Corelet MCR, is illustrated in Fig. 8. The figure illustrates how the hierarchical nature of corelets, corelet composition, encapsulation, modularity, and code reusability are useful in creating complex cognitive systems.

B. Corelet Composition and Decomposition

The code for constructing the Corelet MCR is illustrated in Listing 1. Notice that the creation of its sub-corelet uses two recursive corelet instantiation calls, with parameters being passed to the sub-corelets in a top-down order from the parent corelet to the children. Connectivity is handled after these calls are completed, in a bottom-up order (first the sub-corelets are interconnected, then the parent). Recursive corelet constructor calls apply heterogeneously to all corelet classes. The Corelet MCR constructor does not create any cores or neurons, rather these are created by its sub-corelets. Observe that the developer of the Corelet MCR does not need to worry about either the construction or the connectivity within its constituent Corelet LSM and Corelet SC.

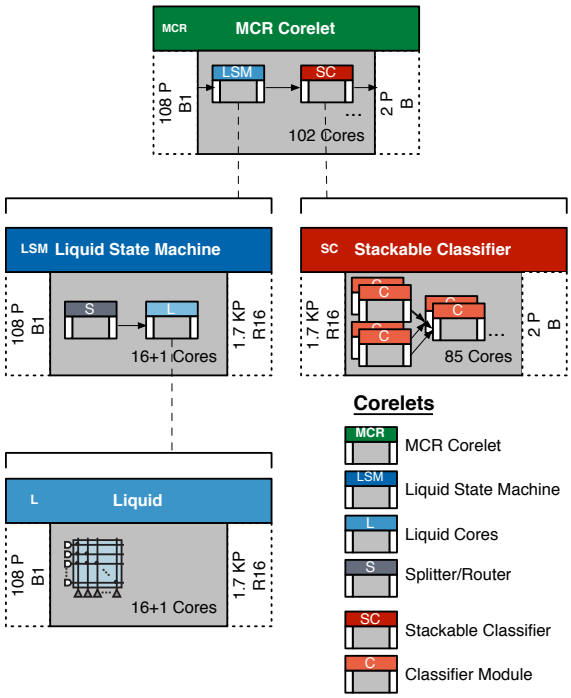

Fig. 8. A corelet diagram of the Music Composer Recognition system, provided here as a Corelet Language "Hello World" example. The application is written as a corelet class: MCR. It is hierarchically composed of two sub-corelets; a parametric Liquid State Machine (LSM) corelet and a Stackable Classifier (SC) corelet. The Corelet LSM has, in turn, two sub-corelets: Corelets Liquid and Splitter. Likewise, the Corelet SC is composed of a hierarchical classifier.

C. Creation of TrueNorth Program

The Corelet Laboratory provides functionality to instantiate, compose, and execute TrueNorth programs with the TrueNorth simulator, Compass. The pseudocode required to instantiate the Corelet MCR (shown in Fig. 8) as `app` and run it is illustrated in Listing 2. First, the corelet is created by invoking its constructor (Line 2). The external input and output interfaces of `app` are declared in Lines 3-4. The corelet is verified (Line 5) to ensure that its model is complete and valid for the TrueNorth architecture. When the verification stage is successful, the network model file is created by the modelGen method via decomposition (Line 6), generating a model file in JSON format, describing the network of neurosynaptic cores, that is, a TrueNorth program.

D. Input and Output Map Files

Input and Output maps define the external interfaces of the TrueNorth program by providing look-up tables of the destination axons and the source neurons connected to *External* input and output connectors. External input axons can be fed by sensory input, and external output from neurons can send spikes to actuators, or other devices. Lines 7-8 in Listing 2 show how the Corelet Laboratory enables generation of input and output map files for `app`. When `app` is built, its input and output serve as the external interfaces to the system. These connectors are set as *external* connectors, and are saved as the input and output map files of the network.

```
1
classdef MCR < corelet
  % Corelet MCR(nInputs,nLayers,nClasses,W)
  % nInputs - number of inputs per layer
  % nLayers - number of layers in LSM
  % nClasses - number of Music Composers
  % W - classifier weight matrix
  methods % public
      function obj=MCR(nInputs,nLayers,nClasses,W)
      obj.name='Music Composer Recognition Corelet';
      % create sub-corelets
      lsm = LSM(nInputs,nLayers);
      sc = SC(nClasses,W);
      obj.subcorelets=[lsm, sc];
      % create connectors
      obj.inp(1)=connector(nInputs,'input');
      obj.out(1)=connector(nClasses,'output');
      % connect sub-corelets and connectors:
      obj.inp(1).busTo(lsm.inp(1));
      lsm.out(1).busTo(sc.inp(1));
      sc.out(1).busTo(obj.out(1));
    end % of constructor
  end % of methods
end % of classdef
```

Listing 1. The constructor of the Corelet MCR for Music Composer Recognition which is a sample application described in Fig. 8. The `MCR` class is derived from the corelet class (Line 1). The constructor, a MATLAB function, receives four parameters and returns a handle (reference) to the constructed corelet object, `obj`. It creates its two sub-corelets, `lsm` and `sc`, by recursively invoking their corresponding corelet constructors with the appropriate parameters. Each sub-corelet call returns a handle to the corelet object it created. In Line 13 the two sub-corelet handles are stored in `obj.subcorelets`, making it a 1×2 array of corelets. Next, the Corelet MCR creates its own input and output connectors (Lines 15-16). Finally in Lines 18-20 it *composes* its sub-corelets by creating the connections which are marked by the three little arrows inside the Corelet MCR in Fig. 8. Using the connector's `busTo()` method it connects its own input connector to the input of `lsm`; the output of `lsm` to the input of `sc`; and the output of `sc` to its own output. The processes of composition and decomposition are further described in Fig. 6.

E. Transduction and Simulation

When the corelet is executed, the utility function `videoToSpikes`(Line 9) is called to generate an input spike file from an indicated video file. The individual pixel gray levels from the input video are converted to spikes. The spikes of a pixel are mapped to a core-axon tuple, corresponding to a specific pin of the input connector defined in the input map file (Line 9). This process of converting data to spikes is called transduction. The generated spikes are then stored in an input spikes file. A raster of a typical spike file is shown in Fig. 9.

Simulation of the model file with its input spikes occurs outside of the MATLAB environment using the Compass [3] simulator. The simulator expects three arguments: a configuration file (app.conf), a model file (app.json), and an input spike file (v1.sfbi). It simultes the model operation with the input spikes and produces an output spikes file.

After the simulation completes, a utility function loads the generated spikes into an array `s` (Line 11), where they can be plotted or analyzed. The `read_spike_file` function uses the output map to interpret the spike addresses. Finally, the spikes in `s` are visualized as a video.

```
corelet_init;
app = MCR(nInputs,nLayers,nClasses,W)
app.inp(1).setExternalInput('audioIn');
app.out(1).setExternalOutput('classOut');
if (app.verify())
  app.modelGen('app.json');
  app.writeInputMapFile('app_iMap.bin');
  app.writeOutputMapFile('app_oMap.bin');
  videoToSpikes('v1.avi','v1.sfbi','app_iMap.bin');
  !compass app.conf app.json v1.sfbi
  s = read_spike_file('v1.sfbo','app_oMap.bin');
  spikePlayback(s);
else
  disp('Verification failed.');
end
```

Listing 2. The corelet generation and running script, starting from instantiation of the sample Corelet MCR (described in Fig. 8), verification, generation of a network model file, I/O map files and spikes input files, running the network with the input spikes, and reading and visualizing the output spikes.

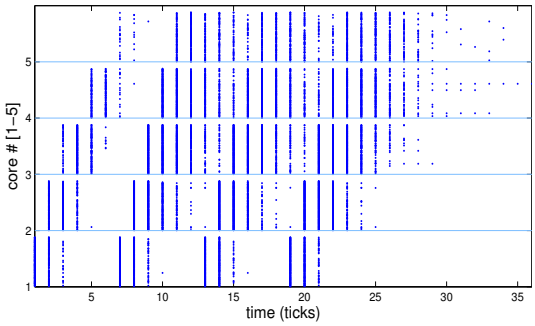

Fig. 9. A spike raster of a five-layers TrueNorth test program, with each layer composed of one seed corelet. Output spikes from the 256 neurons in each neurosynaptic core are stacked along the y-axis. Spike time is along the x-axis (1tick=1mSec).

VI. Related Work

Mark-up languages (NeuroML[15]), simulator-independent languages (PyNN[16], [17]), and domain-specific languages (OptiML [18][19]) have been developed for neural modeling and simulation in software, independent of any neurosynaptic hardware. In contrast, the corelet software is tied to the underlying TrueNorth hardware architecture. In this context, the corelet programming paradigm has similarities with Hardware Description Languages (VHDL and Verilog [20]) in terms of hardware-compatibility, concurrency, encapsulation, and composition.

Queueing Petri Net Modeling Environment[21] is a system for modeling, simulation and analysis of processes using Petri nets. It includes a graphical user interface for editing and composition of hierarchical models. However, a graphical editor might be ineffective for networks of the size and connectivity complexity of typical TrueNorth programs. The recent Connection-Set Algebra is a high level language for specifying connectivity patterns between groups of neurons [22] by using set algebra and matrix operations to create connectivity lists and adjacency matrices. In a similar vein, the corelet programming paradigm employs connectors and permutations for long-distance "white-matter" connectivity and synaptic crossbar matrices for short-distance "gray-matter" connectivity, and leverages MATLAB's rich algebraic and matrix operators.

A language for expressing the inherent massive parallelism of neural networks was presented in [23]. The network is converted into an abstract internal representation and then mapped onto multiple von-Neumann processors, which is different from the corelet programming paradigm, where the network is mapped to inherently parallel TrueNorth architecture. Finally, C* [24] was developed in 1988, as an extension to C, to build applications for the Connection Machine[25], a non von-Neumann architecture.

VII. Conclusion

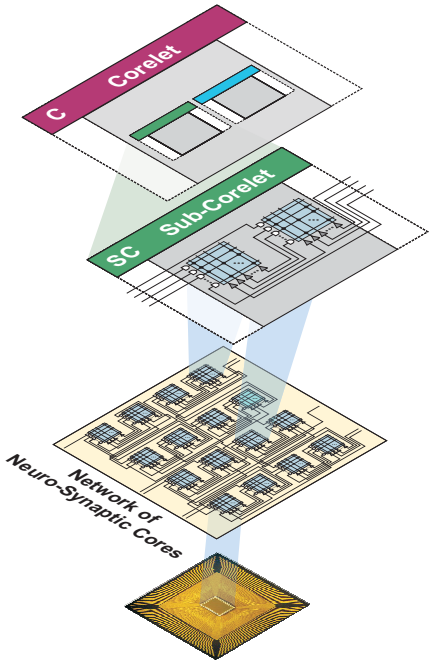

Fig. 10. A corelet is an abstraction of a TrueNorth program and is composed hierarchically from sub-corelets (other corelets) while ensuring correctness, consistency, and completeness with respect to the TrueNorth architecture.

The linear sequential programming paradigm developed for von Neumann is wholly unsuited for the TrueNorth parallel cognitive architecture. Effective tools for thinking transform cognitively complex tasks into simpler ones at which humans can excel [26]. This process involves choosing the appropriate metaphors to scaffold thinking and harnessing the expressive capacity of the language to effectively translate a desired computation into an efficient, executable program.

Therefore, starting from first principles, in this paper, we have defined a novel metaphor of a TrueNorth program, namely, corelets, see, Fig. 10. Leveraging this notion, we have developed an entirely new programming paradigm that can permit construction of complex cognitive algorithms and applications while being efficient for TrueNorth and effective for programmer productivity. The paradigm consists of a language for expressing corelets; a library of corelets; and a laboratory for experimenting with corelets.

Philosophically, the corelet programming paradigm has affinity with Gottlob Frege's *Principle of Compositionality* [27]:

> The meaning of a complex expression is a function of the meanings of its constituents and the way they are combined.

The Corelet Library is a store of accumulated knowledge and wisdom that can be repeatedly re-used. New corelets are written and added to the library, which keeps continually growing in a self-reinforcing way. Based on the compositionality of corelets and on our experience regarding the combinatorial growth of the Corelet Library, we posit an empirical law for TrueNorth's software capability C_S:

$$C_S \quad \propto \quad L^{\lambda}$$

where L is the size of the library and $\lambda > 1$ is a constant.

Here, we have focused on essential concepts underlying the programming paradigm and presented them in their simplest form to aid understandability. However, the Corelet Language supports powerful primitives, such as *parametric corelets* that can instantiate a rich variety of corelet objects at run-time from a single corelet class, and *meta-corelets* that operate on other corelets to compactly create extremely large and powerful TrueNorth programs. With a view towards large-scale TrueNorth programs, we are currently extending the programming paradigm using MATLAB's Parallel Computing Toolbox.

When referring to the *von Neumann bottleneck*, John Backus, in his 1972 Turing Lecture [28] said: "it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand." We now live in an instrumented world that is inundated with a tsunami of data from sensors. Most of this data is parallel in nature and is ideal for parallel processing by TrueNorth. However, to invent effective algorithms and applications, we need to move away from long, sequential von Neumann thinking to short, parallel thinking. We believe that the corelet programming paradigm is the right paradigm for capturing complex, parallel thinking and for composing complex cognitive systems.

ACKNOWLEDGMENTS

This research was sponsored by DARPA under contract No. HR0011-09-C-0002. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of DARPA or the U.S. Government. We would like to thank David Peyton for his expert assistance in revising this manuscript.

REFERENCES

[1] D. S. Modha *et al.*, "Cognitive computing," *Communications of the ACM*, vol. 54, no. 8, pp. 62–71, 2011.

[2] P. Merolla *et al.*, "A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm," in *IEEE Custom Integrated Circuits Conference (CICC)*, Sept. 2011, pp. 1–4.

[3] R. Preissl *et al.*, "Compass: A scalable simulator for an architecture for cognitive computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2012)*, Nov. 2012, p. 54.

[4] J. Seo *et al.*, "A 45nm CMOS neuromorphic chip with a scalable architecture for learning in networks of spiking neurons," in *IEEE Custom Integrated Circuits Conference (CICC)*, Sept. 2011, pp. 1–4.

[5] J. V. Arthur *et al.*, "Building block of a programmable neuromorphic substrate: A digital neurosynaptic core," in *The International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2012, pp. 1–8.

[6] T. M. Wong *et al.*, "10^{14}," IBM Research Divsion, Research Report RJ10502, 2012.

[7] D. S. Modha and R. Singh, "Network architecture of the long distance pathways in the macaque brain," *Proceedings of the National Academy of the Sciences USA*, vol. 107, no. 30, pp. 13 485–13 490, 2010.

[8] E. McQuinn *et al.*, "2012 international science & engineering visualization challenge," *Science*, vol. 339, no. 6119, pp. 512–513, February 2013.

[9] A. S. Cassidy *et al.*, "Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores," in *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2013.

[10] S. K. Esser *et al.*, "Cognitive computing systems: Algorithms and applications for networks of neurosynaptic cores," in *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2013.

[11] J. E. Sammet, "Programming languages: history and future," *Communications of the ACM*, vol. 15, no. 7, pp. 601–610, 1972.

[12] E. Levenez, "History of programming languages," February 2013. [Online]. Available: http://oreilly.com/news/graphics/prog_lang_poster.pdf

[13] J. Backus *et al.*, "The fortran automatic coding system," in *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*. ACM, 1957, pp. 188–198.

[14] L. Pape, J. de Gruijl, and M. Wiering, *Speech, Audio, Image and Biomedical Signal Processing Using Neural Networks*. Springer, 2008, ch. Democratic Liquid State Machines for Music Recognition, pp. 191–215.

[15] P. Gleeson *et al.*, "NeuroML: a language for describing data driven models of neurons and networks with a high degree of biological detail," *PLoS computational biology*, vol. 6, no. 6, p. e1000815, 2010.

[16] A. Davison *et al.*, "PyNN: a common interface for neuronal network simulators," *Frontiers in neuroinformatics*, vol. 2, 2008.

[17] M. Diesmann and M. Gewaltig, "NEST: An environment for neural systems simulations," *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis*, vol. 58, pp. 43–70, 2001.

[18] A. K. Sujeeth *et al.*, "OptiMLl: An implicitly parallel domain-specific language for machine learning," in *Proceedings of the 28th International Conference on Machine Learning, ICML*, 2011.

[19] T. Rompf *et al.*, "Building-blocks for performance oriented DSLs," *arXiv preprint arXiv:1109.0778*, 2011.

[20] P. Ashenden, *The designer's guide to VHDL*. Morgan Kaufmann, 2008, vol. 3.

[21] S. Kounev, C. Dutz, and A. Buchmann, "Qpme-queueing petri net modeling environment," in *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*. IEEE, 2006, pp. 115–116.

[22] M. Djurfeldt, "The connection-set algebra–a novel formalism for the representation of connectivity structure in neuronal network models," *Neuroinformatics*, pp. 287–304, 2012.

[23] A. Strey, "Epsilonna specification language for the efficient parallel simulation of neural networks," *Biological and Artificial Computation: From Neuroscience to Technology*, pp. 714–722, 1997.

[24] M. Norton, "Simulation neural networks using c*," in *Frontiers of Massively Parallel Computation, 1988. Proceedings., 2nd Symposium on the Frontiers of*. IEEE, 1988, pp. 203–205.

[25] W. Hillis, *The connection machine*. MIT press, 1989.

[26] D. Norman, *Things that make us smart: Defending human attributes in the age of the machine*. Perseus Books, 1993.

[27] M. Werning, W. Hinzen, and E. Machery, *The Oxford Handbook of Compositionality*. OUP Oxford, 2012.

[28] J. Backus, "Can programming be liberated from the von neumann style?: a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.