Neuromorphic Programming: Emerging Directions for Brain-Inspired Hardware

Steven Abreu

CogniGron Center & Bernoulli Institute
University of Groningen
Groningen, Netherlands
s.abreu@rug.nl

Jens E. Pedersen

Computational Science and Technology
KTH Royal Institute of Technology
Stockholm, Sweden
jeped@kth.se

Abstract—The value of brain-inspired neuromorphic computers critically depends on our ability to program them for relevant tasks. Currently, neuromorphic hardware often relies on machine learning methods adapted from deep learning. However, neuromorphic computers have potential far beyond deep learning if we can only harness their energy efficiency and full computational power. Neuromorphic programming will necessarily be different from conventional programming, requiring a paradigm shift in how we think about programming. This paper presents a conceptual analysis of programming within the context of neuromorphic computing, challenging conventional paradigms and proposing a framework that aligns more closely with the physical intricacies of these systems. Our analysis revolves around five characteristics that are fundamental to neuromorphic programming and provides a basis for comparison to contemporary programming methods and languages. By studying past approaches, we contribute a framework that advocates for underutilized techniques and calls for richer abstractions to effectively instrument the new hardware class.

Index Terms—neuromorphic computing, brain-inspired computing, hardware-software co-design, programming techniques

I. Introduction

Computing technology is steering toward an impasse, with Dennard scaling ending and Moore's law slowing down [1]. The impasse gives rise to innovation opportunities for specialized hardware in computer architecture [2] as well as in software [3]. This 'Golden Age' of innovation has led many researchers to investigate neuromorphic computers. Taking inspiration from how the brain computes has a rich history [4] and the recent success of deep learning has demonstrated the power of neural information processing [5]. The development of event-based sensors, large-scale neuromorphic processors, and brain-computer interfaces indicate that neuromorphic computing is on the rise and expected to play an important role in the future of computing [6].

Neuromorphic computers take inspiration from the brain, both in the way information is processed and in the fact that the physical dynamics of the underlying substrate are exploited for computation [7]. A neuromorphic computer is composed of neurons and synapses which model biological neural networks. The hardware can either directly implement these biological neural network models physically [8] or use them as inspiration and guidelines to design a more general, flexible, and configurable architecture [9]. Key features of these biological networks include sparse connectivity, event-based communication, low-precision activations, and a focus on temporal processing.

The departure from fundamental assumptions in classical computing brings an urgent need for new theories to describe the computations in novel neuromorphic hardware, along with new theories and methods of programming that can make these devices useful. The former has been outlined in recent work [7, 10] whereas the latter is constrained to an as-yet limited set of "spiking neuromorphic algorithms" [11, 12]. Schuman *et al.* [11] argue that progress on neuromorphic programming requires a paradigm shift in how to think about programming. Current programming methods are adapted to *clocked digital hardware*, but with the forthcoming diversity of computer hardware and architectures [2] it is time to widen the set of hardware systems supported by our programming models.
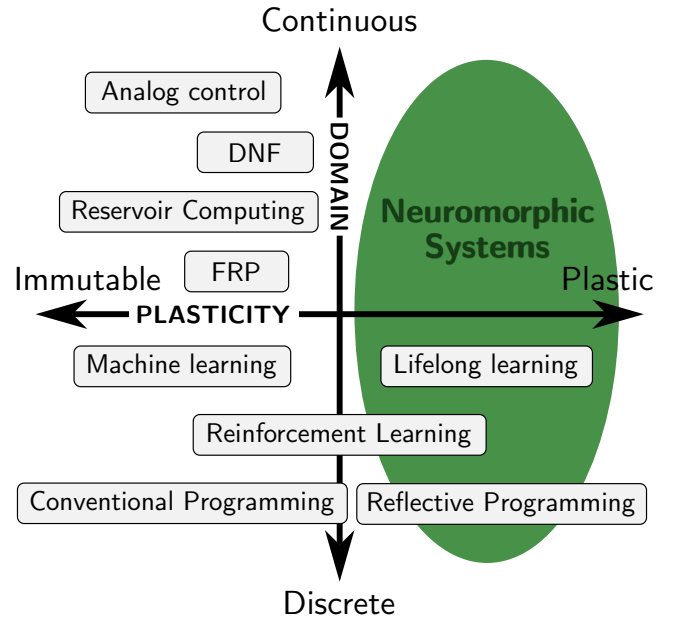

Fig. 1. Programming and computational models plotted against their operational domain (continuous vs. analog) and the malleability of the computation during execution (immutable vs. plastic). The class of neuromorphic systems is shown as a green ellipse, spanning both the continuous and discrete domains. There is a lack of models for continuous and plastic systems.

However, we need not start from scratch when designing methods and paradigms for neuromorphic programming. A lot of work has already been done that we can draw inspiration from [9], and new hardware is known to rejuvenate unconventional research ideas, as demonstrated by GPUs reviving research on neural networks trained with gradient descent [5, 13]. With the emergence of cutting-edge neuromorphic hardware, revisiting unconventional programming paradigms can uncover promising new directions [14].

In this work, we discuss the practical implications of programming neuromorphic systems and compare them with conventional methods in computer science. We extend traditional concepts of programming to embrace the uniquely physical and adaptive properties of neuromorphic systems so as to leverage these systems more effectively. Our contributions are twofold: first, we provide a detailed conceptual framework in Section III that redefines programming in the context of neuromorphic hardware, challenging traditional paradigms and encouraging the adoption of novel approaches. Second, we identify and discuss advanced programming models in Section IV that are currently underutilized in the field, proposing new directions for their evolution and integration into mainstream computing practices.

II. MOTIVATION

Neuromorphic hardware leverages the computational principles of the brain, which are vastly different from those exploited by conventional digital computers [4]. These fundamental differences pose challenges for traditional programming abstractions, and it is clear that we cannot apply conventional theoretical computer science to understand the computational processes in neuromorphic systems [15]. We highlight five fundamental differences underpinning this incompatibility.

a) Domain: Neurons are physical systems and operate in continuous time (see Figure 1). The merits of analog computation in terms of energy efficiency and inherent parallelism are well-known [16, 17]. Classical symbolic computation, in contrast, is decoupled from real physical time and time is only *simulated* through a discrete global clock signal.

b) Plasticity: When programming digital computers, one may neglect the physical properties of the underlying hardware, as the underlying hardware does not change during execution. In neuromorphic computers, such hardware-agnostic programming is not generally possible, as these devices are by definition *physical*. Any programming model of neuromorphic systems must, therefore, include the malleability of the system, since the physical system and the computational model are one and the same.

c) Stochasticity: Unlike deterministically switching transistors, neural systems are *stochastic*, which has lead to models of computation with probabilistic logic [18] and stochastic computing [19], where information is represented in probability distributions.

d) Decentralization: The *distributedness* of information representation and processing in neural networks stands in contrast to the localized information in binary transistor states and the sequential execution of elementary instructions in digital hardware. Such distributedness is leveraged in deep learning [5], in dynamic neural fields [20] where neurons are considered independent independently evolving dynamical systems, and in hyperdimensional computing [21] where high-dimensional random vectors are used for information representation and computation.

e) Unobservability: Neuromorphic hardware, especially analog and mixed-signal systems, often show limited observability, in that the system state can only be read out in parts. This is a key difficulty for plastic computations that change over time. It also relates to mismatches between platforms—a known challenge for analog devices that can also be observed between digital systems [22].

Figure 1 fits programming and computational concepts onto two axes: plasticity and domain. It is immediately clear that the upper right quadrant is largely left unexplored. To explain the vacuum, it is necessary to expand our concept of classical computation to include neuromorphic, physical systems.

III. THEORETICAL FRAMEWORK

The present section introduces a theoretical framework that abstracts "computation" to capture both classical and neuromorphic systems. We begin by conceptualizing how neuromorphic computing fits within the broader landscape of computational models, highlighting areas that remain underexplored and elucidating why these gaps exist.

A. Computing with physical systems

Horsman *et al.* [23] provide a general framework for computation with arbitrary physical systems. Therein, a computer is a physical machine Ψ which can be stimulated by an input signal u^Ψ and from which an output signal y^Ψ can be read out. The computation λ is specified by an abstract function from input u^λ to output y^λ. The machine Ψ then implements the computation λ if an encoding procedure E and decoding procedure D is known such that the machine Ψ will produce y^Ψ with $D(y^\Psi) \approx y^\lambda$ when stimulated with the input signal $E(u^\lambda) = u^\Psi$. This leads to the general form of the abstract computer model shown in Figure 2: the physical machine Ψ receives input u^Ψ and produces output y^Ψ, thereby implementing the abstract computation λ from input u^λ to output y^λ.

Any machine that is changing during program execution will, according to Horsman *et al.* require either that the machine is insensitive to the change such that the effective input-output function stays unaffected, or that the computational model includes the change, so the computation stays correct. Thus, to fit the framework of Horsman *et al.*, neuromorphic computing first requires a model that captures their (plastic) operation. This can be expressed in the formalism of (non-autonomous) dynamical systems, logic-based systems or other formalisms [10].

For digital computers, the abstract model of computation was developed first and only later physically realized. In contrast, neuromorphic hardware does not rely on any universally

accepted abstract model [24]. Abstract models of computation are co-developed with physical implementations [25].

B. Computer programs

While a computation $\mathcal{C}$ specifies *what* is being computed, a program $\mathcal{P}$ specifies *how* the computation is implemented. Many different programs $\mathcal{P}_1, \ldots, \mathcal{P}_n$ may implement the same computation $\mathcal{C}$. Note that the concept of a 'program' herein includes algorithms as Turing machines as well as learning algorithms and interactive programs—although the latter two cannot be implemented by Turing machines [26, 27].

A computation $\mathcal{C}$ is described by a formal specification that formalizes the intention of the computation (Figure 3). The specification of a computation is expressed in some mathematical formalism. In digital computing, this can be done using formalisms from logic. In analog computing, there are various formalisms that describe the computation, for example qualitative geometrical constructs like attractors and bifurcations [10].

A program $\mathcal{P}$ is described in another formalism. In digital computing, programs are expressed in some programming language. In analog computing, one typically uses differential equations to describe the program. When programs interact with another, one also speaks of each individual program as a *process* and the ensemble of all processes as the program, whose behavior emerges from the interaction of the processes.

Operationally, a program is defined by the data flow and control flow. The data flow specifies how signals that carry computationally relevant information are propagated through the machine. The control flow specifies what operations or transformations are done on these signals. For example, in a field-programmable gate array (FPGA) the data flow is configured through its routing elements while the control flow is defined by the function implemented in each logic block. In a neuromorphic chip, the data flow is defined by the connectivity of the neural network while the control flow is defined by the synapse and neuron models, learning rules, synaptic weights, time constants, thresholds, and more.

C. Programming process

Programming, in the context of the theoretical framework above, is the process of designing *how* a certain computation should be implemented, illustrated in Figure 3. Programming begins with some informal intention of what computation to implement. This intention can be formalized into a specification (right path in Figure 3), or the programmer may directly come up with an informal idea for a program that implements the intended computation, and then code this idea in some formal language (left path in Figure 3). This program is then communicated to the physical computer through a pre-defined programming interface. Finally, the system executing this program can be controlled or instructed to remain within the program's specification.

Programming need not be done by a human programmer. As shown by the green arrows in the diagram, programming can also be automated: a computer program can take in a formal specification of a program to then create a program that satisfies these specifications. In *program synthesis*, the specification is given in some algebra or formal model and a search algorithm then finds a program that meets this specification. In *supervised machine learning*, the specification is given by a dataset of input-output examples and an error function, which is then minimized for some machine learning model until it meets a given error threshold.
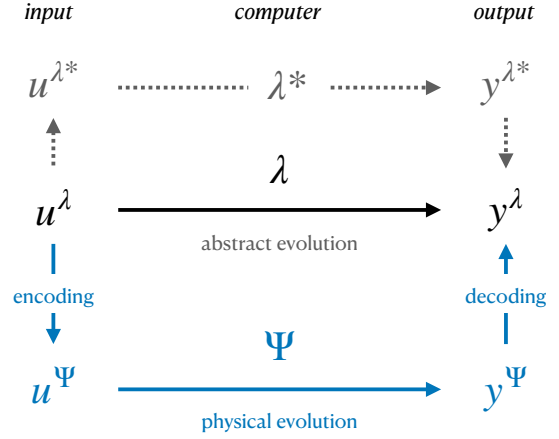

Fig. 2. The relationship between an abstract computation $\lambda : u^\lambda \mapsto y^\lambda$ (middle), the same computation expressed in a different computational model λ^* such as a higher-level programming language (top), and the physical computer Ψ that realizes this computation (bottom), from [23].

D. Languages and Paradigms

Conventionally, programming amounts to writing source code, *coding*, in some formal language. Here, 'programming language' is used in an unconventionally wide sense to include any formal language that can be communicated to a physical system. This includes programming languages like Python but also extends to other formalisms like differential equations describing dynamical systems, or block diagrams describing signal processing systems. In any case, the 'programming language' must be compatible with the elementary instructions that the computer's programming interface provides. Given this compatibility, the programmer is free to explore the space of all possible programs. Work on elementary instruction sets for non-digital computers goes back at least to the 1940s and continues to the present day [28, 29] but there is still no universally accepted model [10]. Consequently, it is not clear what a neuromorphic programming language may look like [30]; will it require new syntax such as visual representations, or will a program be represented by a string of symbols in some formal language?

Since the goal is to improve the way we think about programming neuromorphic computers in general, Floyd [31] argued that it is more effective to turn to *programming paradigms* rather than programming languages. A programming paradigm is an approach to programming "based on a mathematical theory or a coherent set of principles" [32] and

a programming language implements one or more programming paradigms. Centering the discussion on programming paradigms shifts the focus away from syntactical issues to the way programs are conceived and designed.
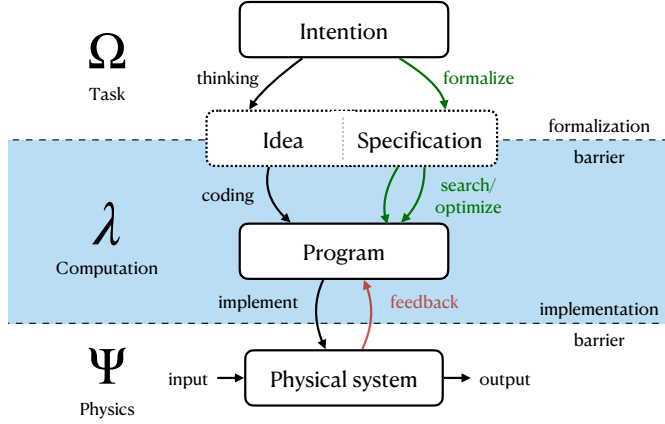

Fig. 3. The process of computer programming, inspired by [33]. The abstract intention lives in the informal space Ω of all possible computational tasks. This intention is formalized into a program $p \in \lambda$ that is expressed in some formal language, which is then implemented through some physical system $s \in \Psi$.

Beyond languages and paradigms, there is a set of well-developed tools for computer programming without which most modern software systems would not exist. We mention the integrated development environment (IDE), the keyboard-and-mouse interface, version control systems, agile development only in passing, as a more detailed treatment of these is outside the scope of this paper.

IV. Programming Paradigms

In the present section, existing programming paradigms are explored and then related to Neuromorphic Programming in Section V. Conventional computer programming paradigms can be differentiated by the program's representation. One commonly distinguishes *imperative programming*, where a program consists of a sequence of instructions, from *declarative programming*, where a program consists of a sequence of logical assignments. We use the term *decentralized programming* for parallel, concurrent, and distributed programming, spanning both imperative and declarative styles. We distinguish these paradigms from *automated programming*, in which the human programmer passes a (potentially incomplete) specification into an optimization or learning algorithm that creates the final program. Finally, we also give a brief overview of *non-digital programming* methods that are applicable to analog and physical computers.

A. Imperative programming

The most common way of writing sequential, instruction-based programs uses the *imperative* paradigm, as implemented in C. Imperative programming was augmented with objects, which can contain instructions as well as data, to yield the *object-oriented* paradigm, as implemented in C++ or Java.

Imperative programming is the dominant paradigm for large-scale software systems in conventional computing.

B. Declarative

Instead of describing the control flow of a program in imperative programming, declarative programs describe the logic of the program. A declarative program describes *what* the program does rather than *how* it does it. Declarative programming is done in database query languages like SQL, functional programming languages like Haskell, or logic programming languages like Prolog. In *dataflow programming*, a program is modeled as a graph of data flowing between operations. A functional cousin of dataflows is the *functional reactive programming* (FRP) that couples hybrid systems of *behaviors* and *events* with continuous-time flow [34]. Dataflow programming is well-suited for neuromorphic computers, where data flows between neurons.

While classical programs are deterministic, the execution of a *probabilistic program* depends on random numbers, for example by calling a (pseudo) random number generator. Such a program can be viewed as sampling from a probability distribution. In *probabilistic programming*, the program itself is considered a distribution, and the programmer can analyze this distribution and condition the distribution on observations [35]. Indeed, the goal of probabilistic programming is not simply the execution of a program, but also the analysis thereof.

C. Decentralized programming

A single processor core may use time-sharing to allow multiple programs to run concurrently. In so-called *concurrent programming*, the lifetime of multiple computing processes overlap and may interact with another [36]. Concurrency introduces issues of synchronization, such as deadlocks and race conditions. With the advent of multicore microprocessors came the need to use resources on *different* cores simultaneously. This led to the development of *parallel programming* techniques, in which multiple processes are carried out simultaneously on different cores. *Distributed programming* deals with programs that are executed on multiple networked computers, which interact to achieve a common goal. The first approach to formalizing this appeared by Milner et al. as π-calculus [37]. Methods from distributed programming are used for multi-chip systems, and frequently employed in training large-scale neural networks.

D. Automated programming

In *meta-programming*, it is possible for a program to write or modify programs, by simply treating the program as data. In *reflective programming*, a program modifies its own behavior whereas in *automatic programming*, a program generates another program. If a formal specification of the desired program is given, *program synthesis* can be used to generate a program that provably satisfies this specification [38]. If exact adherence to a formal specification is not required, but only the satisfaction of given constraints, *constraint programming*

may be used [39]. If an incomplete specification is available, such as input-output examples, then *inductive programming* can be used to generate a suitable candidate program [40].

In classical programming, a human programmer defines the program that specifies how input data is processed. *Machine learning* constructs programs that learn from the input data, in ways that may not have been anticipated by any human. Machine learning has deep roots in probability theory and overlaps significantly with probabilistic programming [41]. In supervised machine learning, a mapping from inputs to outputs is learned from a set of examples. In reinforcement learning, a policy of how to act in some environment is learned from rewards and punishments. Both the learned mapping in supervised learning and the learned policy in reinforcement learning can be used as programs. This makes machine learning a paradigm for automated programming. *Deep learning* uses multi-layered artificial neural networks (ANNs) for machine learning. The connectivity in ANNs is usually fixed, and the weights are learned, typically in a supervised fashion using gradient descent, to minimize the error on given input-output examples. The techniques of deep learning have also been adapted for spiking neural networks [42]. In *differentiable programming*, programs are written in a way that they are fully differentiable with respect to some loss function, thereby allowing the use of gradient-based optimization methods to find better-performing programs. Deep learning is a special case of this, where programs are artificial neural networks that are differentiated using backpropagation.

In machine learning, a core goal is good generalization to unseen examples. If generalization is not needed, one may use *optimization* itself as a programming paradigm in which the result of the optimization is the desired program or the optimization process itself. For example, *evolutionary programming* uses population-based evolutionary optimization algorithms to find programs by encoding the program's specification in a fitness function that is optimized. Evolutionary algorithms have been used to generate rules for a cellular automaton to solve computational problems that are difficult to solve by manually designing a learning rule [43].

E. Non-digital programming

Building on evolutionary optimization, *evolution in materio* [44] was proposed to harness material properties for computation. Natural evolution excels in exploiting the physical properties of materials, and artificial evolution aims to emulate this ability. *Physical reservoir computing* can be used to harness the dynamics of physical systems for computation by modeling the physical system as a high-dimensional reservoir for which a linear readout is trained [45]. It is also possible to create a *surrogate model* of the physical device, then optimize the surrogate model in simulation with deep learning methods and transfer the optimized model back to the device [46]. Although analog computers have been around for at least as long as their digital counterparts, *analog programming* methods are not at the same level of maturity as those for digital computers. Ulmann [47] argues that the development

of reconfigurable analog computers will advance the state of analog computer programming, and efforts to develop such hardware already exist [48]. Currently, analog programming often draws on methods from control engineering, signal processing and cybernetics. For analog neuromorphic computers, *signal processing* provides a rich framework for computing with temporal signals [49]. Moreover, *control theory* has developed a rich repertoire of methods to drive a dynamical system into a mode of operation that is robust, stable, and implements some desired dynamics [50]. These methods can be used to keep analog computers within a desired regime of operation to implement a desired computation. However, the expressiveness of behaviors in control theory lags far behind that of digital programming languages. The field of *autonomic computing* aims to design systems that adapt to stay within a high-level description of desired behavior [51]. The field takes inspiration from the autonomic nervous system, which can stay within a stable 'dynamic equilibrium' without global top-down control.

V. NEUROMORPHIC PROGRAMMING

The preceding section already alluded to some methods that have been adopted for neuromorphic hardware, such as reservoir computing, deep learning, and evolutionary optimization. In the present section, we give a more in-depth review of existing programming approaches for neuromorphic hardware.

a) Neuromorphic co-design: As neuromorphic computers exploit physical phenomena of their underlying hardware, manually designed neuromorphic programs will necessarily be close to physics. Therefore, although not strictly a 'programming' paradigm, it is instructive to consider *neuromorphic co-design* as a paradigm for designing neuromorphic systems. The field is rooted in the original vision of neuromorphic computing [52] and designs application-specific and reconfigurable mixed-signal neuromorphic chips in sub-threshold CMOS technology. This approach uses tools from signal processing and computational neuroscience to implement a desired behavior in networks of silicon neurons [8].

b) Machine learning methods: Given the success of deep learning in applying machine learning techniques to neural networks, this is a natural starting point for neuromorphic computers. However, it is unrealistic to expect deep learning methods to work for SNNs as well as they do for ANNs since these methods were optimized for ANNs [53]. *ANN-to-SNN conversion* is possible, but typically not optimal because the resulting SNNs do not leverage the computational power of spiking neurons. Instead, they limit the richer dynamics of SNNs to the less expressive domain of ANNs [11]. Offline training methods like *backpropagation*, the workhorse of deep learning, can be implemented directly in SNNs using surrogate gradients [42]. Given a neural network, it is necessary to communicate this network to the hardware. *Neuromorphic compilation* [54] was proposed as a general framework to (approximately) compile neural networks into different hardware systems, automatically adapting to physical constraints. *Reservoir computing* can simplify the training of SNNs on

hardware, but it still requires the reservoir states to be read out and stored on a digital computer, which may not be possible given limited observability of some devices or limited data storage.

c) Online learning: The machine learning methods presented above all operate offline and often off-device. Frequent re-training creates a large overhead, limiting the performance and applicability of neuromorphic computers. As a result, *on-device learning* methods are an active topic of research [55]. *Plasticity* is a popular paradigm for on-device learning, where local learning rules are used to modify the connectivity (structural plasticity) and connection strengths (synaptic plasticity) of a SNN. Parallels to emergent programming may be drawn here, as the resulting behavior of the SNN emerges from the interaction of local rules. It is not clear what local rules will yield a particular network-level behavior, but evolutionary search [56] and meta-learning [57] have been used to (re-)discover desirable plasticity rules.

d) Evolutionary methods: A key advantage of evolutionary approaches is that they can jointly optimize the network's architecture and weights, thus simultaneously designing and training the network without requiring the network to be differentiable. However, evolutionary approaches can be slower to converge than other training methods and the resulting architectures are not easily interpretable or reusable for different tasks [11].

e) Spiking neuromorphic algorithms: With the increased availability of neuromorphic hardware, a number of hand-crafted spiking neuromorphic algorithms (SNA) have been proposed. SNAs implement computations using temporal information processing with spikes, often to implement well-defined computations such as functions on sets of numbers [58], functions on graphs [59], solving constraint satisfaction problems or solving a steady-state partial differential equation using random walks [60].

f) Neurocomputational primitives: Various neurocomputational primitives have been proposed in the neuromorphic community. Such primitives can be useful for simple tasks and can be combined into complex neuromorphic systems [61]. The winner-take-all (WTA) network is a common circuit motive in the neocortex that has been used extensively for neuromorphic systems [62]. The more general *dynamic neural fields* (DNFs) are a modern framework for neural attractor networks [20]. The *neural state machine* (NSM) [63] also builds on WTA networks to implement finite state machines in SNNs. The *temporal difference encoder* (TDE) [64] is a circuit primitive that has been used for motion estimation and obstacle avoidance. *Neural oscillators* generate rhythmic activity that can be used for feature binding and motor coordination, for example as a central pattern generator [65]. Other primitives are scattered around the literature, and shared libraries of neurocomputational primitives are only starting to be assembled [61].

g) Higher abstractions: The *neural engineering framework* [66] raises the level of abstraction beyond networks of neurons—it allows dynamical systems to be automatically distilled into networks of spiking neurons using the Nengo programming environment [67]. The framework of *vector symbolic architectures* (VSA) is suitable for neuromorphic systems, enabling knowledge representation and reasoning in high-dimensional spaces. VSAs interfaced with deep networks and generalizations of the optimization and search algorithms described in this survey could provide a path to enabling fast, efficient, and scalable next-generation AI capabilities on neuromorphic hardware. *Lava* is an open-source neuromorphic programming framework that includes libraries of neuromorphic algorithms for optimization, attractor networks, deep learning methods for SNNs, VSAs, and plans to include more paradigms. *Fugu* [68] is a hardware-independent mechanism for composing SNAs. In Fugu, a program is specified as a computational graph, reminiscent of dataflow programming, where nodes represent SNAs and connections represent dataflow between the SNAs.

VI. FUTURE APPROACHES TO PROGRAMMING BRAIN-INSPIRED HARDWARE

For neuromorphic systems to scale to large heterogeneous computing systems, commonly agreed-upon computational models are required, similar to how programming abstractions catalyzed the development of the digital computer in the 1950s and 60s. We have argued that it is difficult to program computers that harness their underlying physical dynamics for computation without a guiding theory that unites physics with computation. But, despite the rich history of programming methods and languages, our analyses in Sections II and III show that the direct translation of classical methods to neuromorphic systems is not a reasonable pursuit: neuromorphic computing (dually analog/digital, plastic, stochastic, decentralized, and unobservable) is incompatible with the assumptions of conventional programming efforts (digital, immutable, noiseless, centralized, observable, see Figure 1). None of the methods and approaches we reviewed meet the criteria for a universally agreed-upon neuromorphic abstraction [6, 11].

However, decades of research in neuromorphic computing and engineering, provide important insights towards initial features of neuromorphic programming methods. Revisiting the requirements from Section II, *modern approaches must accommodate the simultaneously analog and digital nature of neural computation* to supplement digital models with differential equations in some shape or form. This does not exclude digital instructions, but implies approximately continuous-time primitives, such as linearized time-stepped or variable-time models. Second, *the computational models must be able to capture the inherent change in the underlying substrate*. This further challenges digital instructions because arbitrary bit-flips are detrimental to digital computation. Online learning methods may help by continuously adapting to changes in the environment as well as changes in the underlying substrate. Third, *novel approaches should model noise on the signal-level and remain robust to small and unpredictable perturbations*. This is challenging, but possible, to achieve in classical programs, and much more applied in machine learning and

automated programming schemes, and even harnessed in probabilistic programming and stochastic computing. Fourth, *any model should allow for event-based information processing.* Synchronicity in nature is extremely costly, but used almost everywhere in conventional programming paradigms except some reactive and concurrent languages. Finally, *the computation should depend solely on locally available information* rather than on global system states, as done by local plasticity rules used in neuromorphic hardware. Some machine learning and optimization methods, like reservoir computing or evolutionary optimization, also do not require a complete description of the computer's state.

To unite the physical dynamics and computational principles, fully neuromorphic abstractions are preferable, but probably not tractable before the "hardware lottery" has been won [13] and significant gains have been achieved with neuromorphic hardware compared against other computational substrates. A more realistic scenario is that we can play on the strengths of multiple current approaches, and integrate them into one cohesive abstraction. The neuromorphic system hierarchy [54] provides a common abstraction that can represent ANNs and SNNs, but is not designed for neurocomputational primitives, spiking neuromorphic architectures, or other neuromorphic abstractions. The Neuromorphic Intermediate Representation [22] provides an abstraction for graph-structured continuous-time computations, which can support a wider variety of neuromorphic programming paradigms. A key innovation in NIR is the integration of heterogeneous representations, such the Neural Engineering framework, Lava, Fugu, PyNN, and NeuroML. The outcome is that multiple representations and paradigms can cooperate through a shared representation, flexible enough to cover multiple approaches, but unambiguous enough to only provide limits for discrepancies in the execution. However, a complete/full neuromorphic abstraction must also offer support for computational graphs that change over time, which NIR presently does not offer.

A longstanding goal in computer science has been to program physical devices that mimic the efficiency and functionality of the brain [4, 7]. While significant headway has been made towards instrumenting digital as well as non-digital systems, more work is needed to find robust programming methods for neuromorphics. We hope that neuromorphic programmers can leverage the work outlined in this paper to build large-scale neuromorphic programs to tackle real-world tasks, and to further develop guiding principles and paradigms for neuromorphic programming.

ACKNOWLEDGMENT

REFERENCES

[1] M. M. Waldrop, "The chips are down for moore's law," *Nature*, vol. 530, no. 7589, pp. 144–147, 2016.

[2] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[3] C. Edwards, "Moore's law," *Communications of the ACM*, vol. 64, no. 2, pp. 12–14, 2021.

[4] J. von Neumann, *The computer and the brain*. New Haven, CT: Yale University Press, 1958.

[5] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[6] C. Mead, "Neuromorphic engineering: In memory of Misha Mahowald," *Neural Computation*, vol. 35, no. 3, p. 343–383, Feb. 2023.

[7] H. Jaeger, B. Noheda, and W. G. v. d. Wiel, "Toward a formal theory for computing machines made out of whatever physics offers," *Nature Communications*, vol. 14, no. 1, 2023.

[8] G. Indiveri, B. Linares-Barranco, T. J. Hamilton, A. van Schaik, R. Etienne-Cummings, T. Delbruck, S.-C. Liu, P. Dudek, P. Häfliger, S. Renaud, J. Schemmel, G. Cauwenberghs, J. Arthur, K. Hynna, F. Folowosele, S. Saighi, T. Serrano-Gotarredona, J. Wijekoon, Y. Wang, and K. Boahen, "Neuromorphic silicon neuron circuits," *Frontiers in Neuroscience*, vol. 5, 2011.

[9] C. Frenkel, D. Bol, and G. Indiveri, "Bottom-up and top-down neural processing systems design: Neuromorphic intelligence as the convergence of natural and artificial intelligence," 2021.

[10] H. Jaeger, "Towards a generalized theory comprising digital, neuromorphic and unconventional computing," *Neuromorphic Computing and Engineering*, 2021.

[11] C. D. Schuman, S. R. Kulkarni, M. Parsa, J. P. Mitchell, P. Date, and B. Kay, "Opportunities for neuromorphic computing algorithms and applications," *Nature Computational Science*, vol. 2, no. 1, pp. 10–19, 2022.

[12] J. B. Aimone, "Neural algorithms and computing beyond moore's law," *Communications of the ACM*, vol. 62, no. 4, pp. 110–110, 2019.

[13] S. Hooker, "The hardware lottery," *Commun. ACM*, vol. 64, no. 12, p. 58–65, nov 2021.

[14] J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, Eds., *Unconventional Programming Paradigms*. Springer Berlin Heidelberg, 2005.

[15] J. B. Aimone and O. Parekh, "The brain's unique take on algorithms," *Nature Communications*, vol. 14, no. 11, p. 4910, Aug. 2023.

[16] K. Boahen, "A neuromorph's prospectus," *Computing in Science & Engineering*, vol. 19, no. 2, pp. 14–28, 2017.

[17] R. Sarpeshkar, "Analog versus digital: Extrapolating from electronics to neurobiology," *Neural Computation*, vol. 10, no. 7, pp. 1601–1638, 1998.

[18] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata Studies*, pp. 43–98, 1956.

[19] A. Alaghi and J. P. Hayes, "Survey of stochastic computing," *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 2s, pp. 1–19, 2013.

[20] M. A. Giese, "Dynamic neural field theory for motion perception." Boston, MA: Springer US, 1999, dOI: 10.1007/978-1-4615-5581-0.

[21] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive Computation*, vol. 1, no. 2, pp. 139–159, 2009.

[22] J. E. Pedersen, S. Abreu, M. Jobst, G. Lenz, V. Fra, F. C. Bauer, D. R. Muir, P. Zhou, B. Vogginger, K. Heckel, G. Urgese, S. Shankar, T. C. Stewart, J. K. Eshraghian, and S. Sheik, "Neuromorphic Intermediate Representation: A Unified Instruction Set for Interoperable Brain-Inspired Computing," Nov. 2023, arXiv:2311.14641 [cs].

[23] C. Horsman, S. Stepney, R. C. Wagner, and V. Kendon, "When does a physical system compute?" *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 470, no. 2169, p. 20140182, 2014.

[24] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware," *arXiv preprint*, 2017.

[25] S. Abreu, I. Boikov, M. Goldmann, T. Jonuzi, A. Lupo, S. Masaad, L. Nguyen, E. Picco, G. Pourcel, A. Skalli, L. Talandier, B. Vettelschoss, E. A. Vlieg, A. Argyris, P. Bienstman, D. Brunner, J. Dambre, L. Daudet, J. D. Domenech, I. Fischer, F. Horst, S. Massar, C. R. Mirasso, B. J. Offrein, A. Rossi, M. C. Soriano, S. Sygletos, and S. K. Turitsyn, "A photonics perspective on computing with physical substrates," *Reviews in Physics*, vol. 12, p. 100093, Dec. 2024.

[26] L. Valiant, *Probably Approximately Correct: Nature's Algorithms for Learning and Prospering in a Complex World*. Basic Books, Inc., 2013.

[27] P. Wegner, "Why interaction is more powerful than algorithms," *Communications of the ACM*, vol. 40, no. 5, pp. 80–91, 1997.

[28] C. E. Shannon, "Mathematical theory of the differential analyzer," *Journal of Mathematics and Physics*, vol. 20, no. 1-4, pp. 337–354, 1941.

[29] J. Hasler, "Defining analog standard cell libraries for mixed-signal computing enabled through educational directions," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020.

[30] O. Michel, J.-P. Banâtre, P. Fradet, and J.-L. Giavitto, "Challenging Questions for the Rationale of Non-Classical Programming Languages," *International Journal of Unconventional Computing*, vol. 2, pp. 337–347, 2006.

[31] R. W. Floyd, "The paradigms of programming," *Communications of the ACM*, vol. 22, no. 8, pp. 455–460, 1979.

[32] P. V. Roy, "Programming paradigms for dummies: What every programmer should know," in *New computational paradigms for computer music*, G. Assayag and A. Gerzso, Eds. Éditions Delatour France (Le vallier), 2009, pp. 9–47.

[33] G. Grünert, "Unconventional programming: non-programmable systems," Ph.D. dissertation, Friedrich-Schiller-Universität Jena, 2017.

[34] Z. Wan and P. Hudak, "Functional reactive programming from first principles," ser. PLDI '00. New York, NY, USA: Association for Computing Machinery, May 2000, p. 242–252.

[35] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, "Probabilistic programming," in *Future of Software Engineering Proceedings*. ACM, 2014.

[36] R. Milner, "Elements of interaction," *Communications of the ACM*, vol. 36, no. 1, pp. 78–89, 1993.

[37] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, i," *Information and Computation*, vol. 100, no. 1, p. 1–40, Sep. 1992.

[38] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[39] F. Rossi, P. van Beek, and T. Walsh, "Constraint programming," in *Handbook of Knowledge Representation*. Elsevier, 2008, ch. 4, pp. 181–211.

[40] S. Gulwani, J. Hernández-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. Zorn, "Inductive programming meets the real world," *Communications of the ACM*, vol. 58, no. 11, pp. 90–99, 2015.

[41] K. P. Murphy, *Machine Learning*. MIT Press Ltd, 2012.

[42] J. K. Eshraghian, M. Ward, E. O. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun, D. S. Jeong, and W. D. Lu, "Training spiking neural networks using lessons from deep learning," *Proceedings of the IEEE*, vol. 111, no. 9, pp. 1016–1054, 2023.

[43] M. Mitchell, J. P. Crutchfield, and P. T. Hraber, "Evolving cellular automata to perform computations: mechanisms and impediments," *Physica D: Nonlinear Phenomena*, vol. 75, no. 1-3, pp. 361–391, 1994.

[44] J. Miller and K. Downing, "Evolution in materio: looking beyond the silicon box," in *Proceedings 2002 NASA/DoD Conference on Evolvable Hardware*. IEEE, 2002.

[45] G. Tanaka, T. Yamane, J. B. Héroux, R. Nakane, N. Kanazawa, S. Takeda, H. Numata, D. Nakano, and A. Hirose, "Recent advances in physical reservoir computing: A review," *Neural Networks*, vol. 115, pp. 100–123, 2019.

[46] L. G. Wright, T. Onodera, M. M. Stein, T. Wang, D. T. Schachter, Z. Hu, and P. L. McMahon, "Deep physical neural networks trained with backpropagation," *Nature*, vol. 601, no. 7894, pp. 549–555, 2022, publisher: Springer Science and Business Media LLC.

[47] B. Ulmann, *Analog and Hybrid Computer Programming*. De Gruyter, 2020.

[48] J. Hasler, "Large-scale field-programmable analog arrays," *Proceedings of the IEEE*, vol. 108, no. 8, pp. 1283–1302, 2020.

[49] E. Donati, M. Payvand, N. Risi, R. Krause, K. Burelo, G. Indiveri, T. Dalgaty, and E. Vianello, "Processing EMG signals using reservoir computing on an event-based neuromorphic system," in *2018 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. IEEE, 2018.

[50] D. Kirk, "Optimal control theory: an introduction," 1970.

[51] M. Parashar and S. Hariri, "Autonomic computing: An overview," in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 257–269.

[52] C. Mead, "Neuromorphic electronic systems," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1629–1636, 1990.

[53] M. Davies, A. Wild, G. Orchard, Y. Sandamirskaya, G. A. F. Guerra, P. Joshi, P. Plank, and S. R. Risbud, "Advancing neuromorphic computing with Loihi: A survey of results and outlook," *Proceedings of the IEEE*, vol. 109, no. 5, pp. 911–934, 2021.

[54] Y. Zhang, P. Qu, Y. Ji, W. Zhang, G. Gao, G. Wang, S. Song, G. Li, W. Chen, W. Zheng, F. Chen, J. Pei, R. Zhao, M. Zhao, and L. Shi, "A system hierarchy for brain-inspired computing," *Nature*, vol. 586, no. 7829, pp. 378–384, 2020.

[55] A. Basu, J. Acharya, T. Karnik, H. Liu, H. Li, J.-S. Seo, and C. Song, "Low-power, adaptive neuromorphic systems: Recent progress and future directions," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 1, pp. 6–27, 2018.

[56] J. Jordan, M. Schmidt, W. Senn, and M. A. Petrovici, "Evolving interpretable plasticity for spiking networks," *eLife*, vol. 10, p. e66273, oct 2021.

[57] B. Confavreux, E. J. Agnes, F. Zenke, T. Lillicrap, and T. P. Vogels, "A meta-learning approach to (re)discover plasticity rules that carve a desired function into a neural network," *34th Conference on Neural Information Processing Systems (NeurIPS 2020)*, 2020.

[58] S. J. Verzi, F. Rothganger, O. D. Parekh, T.-T. Quach, N. E. Miner, C. M. Vineyard, C. D. James, and J. B. Aimone, "Computing with spikes: The advantage of fine-grained timing," *Neural Computation*, vol. 30, no. 10, pp. 2660–2690, 2018.

[59] K. E. Hamilton, T. M. Mintz, and C. D. Schuman, "Spike-based primitives for graph algorithms," *arXiv preprint*, 2019.

[60] J. D. Smith, W. Severa, A. J. Hill, L. Reeder, B. Franke, R. B. Lehoucq, O. D. Parekh, and J. B. Aimone, "Solving a steady-state PDE using spiking networks and neuromorphic hardware," in *International Conference on Neuromorphic Systems 2020*. ACM, 2020.

[61] C. Bartolozzi, G. Indiveri, and E. Donati, "Embodied neuromorphic intelligence," *Nature Communications*, vol. 13, no. 1, 2022.

[62] R. J. Douglas and K. A. Martin, "Recurrent neuronal circuits in the neocortex," *Current Biology*, vol. 17, no. 13, pp. R496–R500, 2007.

[63] E. Neftci, J. Binas, U. Rutishauser, E. Chicca, G. Indiveri, and R. J. Douglas, "Synthesizing cognition in neuromorphic electronic systems," *Proceedings of the National Academy of Sciences*, vol. 110, no. 37, pp. E3468–E3476, 2013.

[64] D. Gutierrez-Galan, T. Schoepe, J. P. Dominguez-Morales, A. Jimenez-Fernandez, E. Chicca, and A. Linares-Barranco, "An event-based digital time difference encoder model implementation for neuromorphic systems," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2021.

[65] R. Krause, J. J. A. van Bavel, C. Wu, M. A. Vos, A. Nogaret, and G. Indiveri, "Robust neuromorphic coupled oscillators for adaptive pacemakers," *Scientific Reports*, vol. 11, no. 1, 2021.

[66] C. Eliasmith and C. H. Anderson, *Neural Engineering (Computational Neuroscience Series): Computational, Representation, and Dynamics in Neurobiological Systems*. Cambridge, MA, USA: MIT Press, 2002.

[67] T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. C. Stewart, D. Rasmussen, X. Choo, A. R. Voelker, and C. Eliasmith, "Nengo: a python tool for building large-scale functional brain models," *Frontiers in Neuroinformatics*, vol. 7, 2014.

[68] J. B. Aimone, W. Severa, and C. M. Vineyard, "Composing neural algorithms with fugu," in *Proceedings of the International Conference on Neuromorphic Systems*. ACM, 2019.