A Communication Data Layer for Distributed Neuromorphic Systems

András Veres, Péter Hága, András Rácz, Tamas Borsos, Zsolt Kenesi

A Communication Data Layer for Distributed Neuromorphic Systems

András Veres[0000−0003−0518−7452], Péter Hága[0000−0002−1952−343X], András Rácz[0000−0002−9466−3459], Tamás Borsos[0000−0001−9785−9839], and Zsolt Kenesi[0000−0002−0617−2992]

Ericsson Research, Budapest, Hungary
{andras.veres, peter.haga, andras.racz, tamas.borsos, zsolt.kenesi}@ericsson.com

Abstract. The proliferation of AI into everyday devices is a major trend today. This trend combined with the increasing amount of different AI hardware architectures and software frameworks imposes significant challenges when we want to interconnect such AI-based devices into single, large AI-driven distributed system. This paper addresses one key challenge which is around the problem of sharing AI encoded information among components of vastly heterogeneous nature. For that end we propose a new concept called Neuromorphic Data Layer, which can bridge various internal AI data representations in a communication channel-friendly way. The proposed methods are also stress tested in a distributed industrial robotic control & training use-case where all components are state-of-the-art devices, have some form of AI computation and they are interconnected over wireless technologies using the proposed Neuromorphic Data Layer.

Keywords: Neuromorphic computing · Distributed AI · IoT · Wireless networks · Vector Symbolic Architectures

1 Introduction

Distribution of AI systems into communicating, but physically separate components is going to be a major step in the evolution of AI. This trend is driven by the penetration of AI into edge devices, as well as data centers. Distribution will enable new types of applications, where distinct AI sub-tasks, including sensing, reasoning, planning and control will no longer need to be physically integrated to the same device. Another benefit of distribution is that computation can be scaled-out into large number of devices potentially.

Such distribution brings significant challenges to the interconnecting communication technologies. The most straightforward solution would be to encode neuron activations of an AI component's output layer into data packets and add a packet header pointing to the destination AI component's input neural layer. In case of a neuromorphic system, that means that the output spikes of a neuron population are transmitted with some addressing, like Address Event

Representation (AER). While such method offers high level of flexibility and has the obvious benefit of being transparent to the AI logic, in practice, it is more suitable as a chip-to-chip rather than a device-to-device protocol.

This paper introduces a concept for an AI-specific communication layer called the Neuromorphic Data Layer (NDL) in Section 2. This new layer is compatible with various AI frameworks and it can be implemented using neuromorphic computing and can be integrated with the application logic natively. NDL should allow various types of information representations, here we propose two interfaces or APIs. One is a raw event communication API (E-API) and the other is a so-called symbolic representation API (S-API). The symbolic representation API is based on the concepts of High-Dimensional Computing and Vector Symbolic Architectures [1][2]. We argue that the symbolic S-API has numerous benefits, from allowing to interconnect heterogeneous frameworks, as well as offering ways for the application to adapt to communication impairments.

A real-life, distributed prototype based on the proposed NDL concept is presented in Section 3. The demonstrated application is an industrial use-case, where a person trains a robot using hand gestures in-front of an event camera emitting spikes. The camera input is processed by an Intel Loihi neuromorphic chip, which also implements the S-API of the NDL to interface over a wireless connection with a neuromorphic component representing a central control logic built using Neural Engineering Framework (NEF) [3] . To show the flexibility of the NDL concept, we use the NDL S-API to control an industrial robotic arm using symbolic-encoded commands and we also integrate raw spike communication from a small neuromorphic IoT sensor. For all inter-device communication we used wireless technologies.

Practical considerations for implementing parts of the application and the NDL on Intel Loihi architecture [4] are presented in Section 4. We conclude the paper with empirical observations and numerical results about the performance of the system in Section 5.

2 Neuromorphic Data Layer

Besides the actual application logic and the communication layer there are numerous tasks a distributed neuromorphic system needs to implement. We can collect many of these tasks into a single layer called the Network / IoT Layer (Fig. 1), and should include security, device management, orchestration, data sharing, etc. In a neuromorphic system these should be implemented in neuromorphic computation friendly fashion.

In this paper we focus on the problem of data sharing between components and introduce the Neuromorphic Data Layer (NDL). For the purpose of our distributed neuromorphic application, we defined two types of data APIs for the NDL: a raw event E-API and a symbolic representation S-API.

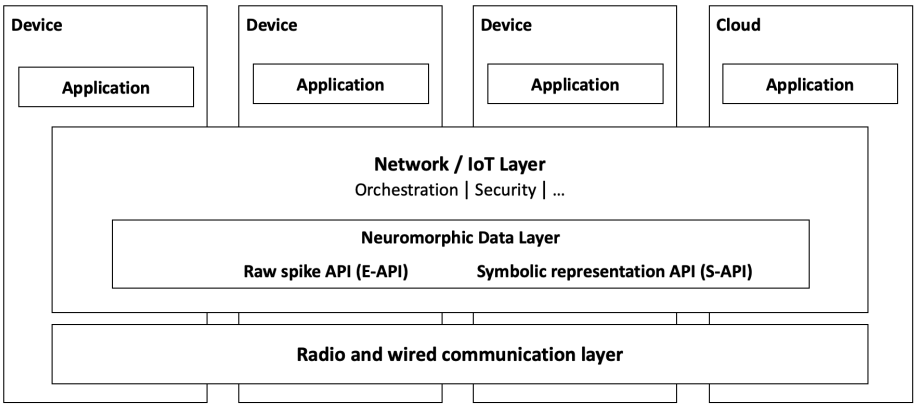


Fig. 1: Conceptual architecture of a wireless distributed neuromorphic system. The physically separated devices share information by using the two representation APIs offered by the Neuromorphic Data Layer.

2.1 Raw Spike API

The E-API interconnects two endpoints of communicating neural networks in a completely application transparent way. Neurons are identified by addresses, and the timing can be implicitly defined as the time of transmission, similar to Address Event Representation (AER). Spikes generated at the same time can be bundled together to reduce packetization overhead.

One drawback of raw spike transmission is that it requires both the sending and the receiving AI components to encode spike patterns exactly the same way. In reality, there are numerous ways how different AI frameworks, chips, sensors encode information, and they have good reasons to differ significantly. For example, a simple motion sensor may encode information as a rate of spikes, but an event camera encodes visual information in a spatio-temporal way. Other frameworks, for example the NEF [3], uses a specific way of encoding values as a combination of postsynaptic current activities of an entire neuron ensemble (via so-called tuning curves).

Another aspect of raw spike-based communication is the impact on the communication medium itself. A radio-based technology can operate best if the traffic is predictable and the application can tolerate some level of loss, jitter or latency [5]. In contrast, a real-time neuromorphic system can show high degree of burstiness due to its event driven nature and can be sensitive to either losses or delays. There may need to be some kind of adaptation to network conditions on the application-side similar to the one TCP/IP provides as well. The network may also have mechanisms to request the application's preferred way of treatment by the network.

2.2 Symbolic Representation API

The S-API represents information on a higher abstraction level than raw spike encoding over the E-API. The data representation in S-API is based on the theory of Vector Symbolic Architectures (VSA) [1][2]. In VSA symbols are represented as (usually random) vectors in a high-dimensional space. VSA defines a similarity metric, such that any two unrelated (randomly assigned) symbol vectors will be highly non-similar (perpendicular). VSA also defines a number of operations such as binding and bundling. Bundling two symbol vectors together creates a third vector, which is very similar to both of the original vectors. Binding the two vectors together results in a new vector that is dissimilar to both vectors, thereby basically, creating a new symbol. The information pieces in the new symbol can be queried by executing an unbinding operation on the new symbol. VSAs have been used in the context of robotics [6] and visual analysis [7].

In the context of communicating AI agents, we apply VSA as a means of a) representing information to be communicated as symbols, b) creating communication data structures as a single VSA symbol from various pieces of VSA encoded information (via binding). On the other end of the communication link the receiving AI agent reverses the previous steps by disassembling the VSA symbol into pieces (unbinding) and decodes the information from individual VSA encoded symbols.

When the sender agent wants to send a piece of data, it uses an internal representation E. The internal representation may depend on the hardware and software frameworks used. For example, an internal state variable may be a represented as the spiking rate of a neuron, delay between spikes or even a collective spiking pattern of a population of neurons.

Regardless of the type of internal representation, there needs to be a function G, which translates the internal representations E_i to a distinct symbolic representations S_i in the symbolic space:

$$S_i = G(E_i). \tag{1}$$

The sender packs symbolic encoded data pieces into a single VSA vector: $V = H(S_1, ..., S_n)$, which is then used for transmission. If both G and H are invertible, the receiver is able to decode each internal state. H can be implemented in various ways, it may for example, use a unique symbol T for each data type. For example T_{temp} to represent temperature, T_{pos} to represent positions. Then H may take the following form:

$$V = H(S_1, ..., S_n) = \sum (T_i \otimes S_i), \tag{2}$$

where $\otimes$ represents the VSA binding operator, and $\sum$ represents the repetitive use of the VSA bundling operator $\oplus$.

The destination agent receives V', which may include errors introduced by the communication channel. Such errors may manifest as bit erasures or additions or in case of non-binary VSA representations, it may add real or complex valued

noise on the VSA vector dimensions, e.g., when using HRR or F-HRR [1]. If the communication channel errors can be modeled as a random error symbol Z bundled (e.g., dimension-based add, or phase shift) on top of the original symbol, then the received symbol will still be VSA-similar to the original due to the nature of the bundle operation:

$$V' = V \oplus Z \approx V \tag{3}$$

This way the receiver can unbind individual T_i data types and recover the sent symbol S_i:

$$R_i = T_i \oslash V' \quad \text{and} \quad R_i \approx S_i, \tag{4}$$

where $\oslash$ is the VSA unbinding operator and "$\approx$" means that the decoded vector R_i is similar to the encoded S_i when compared using the VSA similarity metric (e.g., Hamming-distance or cosine similarity). At this step the receiver can use an associative clean-up memory [8] to remove noise.

In practice, there are limitations of how much communication error can be tolerated without symbol misinterpretation. We present an empirical study in Section 5 showing the impact of radio channel errors in a testbed.

In the final step, the receiver decodes the information stored in S_i into its own format of representation. With notation:

$$E_i^{rec} = G^{rec}(S_i). \tag{5}$$

Even though the information encoded in E_i and E_i^{rec} are the same, their way of representation may be different: G^{rec} may be a completely different function than G. This last step is another important feature of the S-API, as it allows devices using various computational architectures to use the same S-API for communication, they only need to implement their respective G and G^{rec} functions.

2.3 Encoding of Integer Valued State Variables

Probably the most typical data type that applications exchange are integer valued variables, or data structures constructed from integer values. If using the S-API, the application needs to encode its internal integer representation to S-API compatible VSA symbols. Considering that applications may use different internal frameworks, such encoding and decoding may have very different implementation complexities.

From a theoretical point of view there are multiple ways an x integer value may be mapped to a VSA vector. We consider three different assignment methods: Spatial Semantic Pointers (SSPs) [9], VSA permutation and random assignment. In case of SSPs, S_x symbols can be derived from a single basis vector B by self-binding it $x-1$ times:

$$S_x = B \otimes B \otimes ... \otimes B \qquad (B \text{ appears x times}) \tag{6}$$

The above calculation can be extended to encode fractional values as well [10].

The permutation based transformation also requires a single basis vector B, on which a VSA binding operation is performed multiple times.

$$S_x = p^x B. \tag{7}$$

Since permutation can take the form of circular shifting of B, this method is fairly simple to implement on neuromorphic hardware. From a communication point of view both SSP and permutation methods have the advantage that only the B basis vector has to be shared between the sender and the receiver.

The third way is probably the simplest which involves defining a random vector for each integer value. This method provides greater flexibility in customizing the used symbol vectors according to the requirements of the used communication channel. For example, we can restrict the vectors to have exactly n number of 1s randomly placed ($n << L$):

$$S_x \in \{0,1\}^L, \quad \text{where} \quad S_x^T \times S_x = n. \tag{8}$$

The downside of this method is that all the symbols need to be shared (instead of a single basis vector) between the sender and receiver before starting their communication.

2.4 Practical Integer Encoding and Decoding on Neuromorphic Hardware

In case of our Loihi-based spiking neural network (SNN) implementation we apply the following method to encode and decode x to and from S_x. First we encode the x number into a one-hot encoded X vector (i.e., in a form of $X = [0, 0, \ldots, 0, 1, 0, \ldots, 0]$, where all the values are 0 except the 1 standing in the x^{th} position of the vector). In this case the x number can be encoded into S_x simply by:

$$V = M \times X = S_x, \quad \text{where} \quad M = [S_1, S_2, \ldots, S_n]. \tag{9}$$

The one-hot encoded X selects the appropriate S_x symbol from the M encoding matrix or, in other words, selects the x^{th} column of the M matrix that represents the x number. In our Loihi-based implementation x is one-hot encoded by a neuron layer, so the M encode matrix can be directly mapped to the neural connectivity matrix between the neuron layer representing x and an output neuron layer representing the encoded value V. The V vector can be transmitted directly or as part of a complex symbol, see Eq. (2).

As the destination agent receives V' (instead of V) due to transmission errors the decoding process has to select the best candidate out of the possible S_i symbols. By applying the M^T transpose of the encode matrix to the received V' vector we get

$$D = M^T \times V', \tag{10}$$

where D vector contains the complementary Hamming-distance values of the received V' and all the possible S_i symbols. Elements of D indicate a weight of

how many 1s are matching between the V' received vector and the S_i symbols. The best candidate for the transmitted x value is the index of the maximum value of elements of D according to our X definition. By selecting the most similar symbol

$$x = \arg\max(D) \tag{11}$$

we receive the transmitted x value.

The presented decoding method can be easily implemented on the Loihi hardware by using a single neural layer connected to the neurons representing the received vector V' elements where the connection matrix is defined by the M^T matrix and a winner take all algorithm is used to select the neuron with highest activity.

3 Distributed Neuromorphic Prototype System

Using the previously defined Data Layer APIs we built an end-to-end distributed neuromorphic industrial use-case prototype. Our motivation was to prove that E-API and S-API can be used in a practical real-life scenario, and also to be able to perform empirical studies and gain experience. For that purpose, our prototype system was designed to contain highly heterogeneous components: sensors, compute resources, actuators. We also selected various AI platforms and also non-AI components to show that the Data Layer can solve the interconnection of such components efficiently. The overall architecture of the system and photos showing parts of the end-to-end system are shown in Fig. 2.

3.1 End-to-end Use-case

In the implemented use-case a human operator trains a robotic arm by using hand gestures. First, the operator shows the robot what position to take (for simplicity limited to a 2D plane), and the robot mirrors the operator's hand in real-time. Rotation of the palm is translated into the robot gripper's similar orientation. When the desired position and pose are achieved, the operator can command the system (using a gesture) to store the exact position and pose in its memory. The operator can switch among real-time tracking, storing and replaying at any time.

The implementation is distributed into several, wirelessly interconnected components. The operator holds a small IoT device containing an LED active marker emitting sequences of high-rate (several kHz) light impulses and an inertial measurement unit (IMU). The IMU feeds a small neuromorphic network on board and transmits spike-encoded orientation data over the air using E-API.

The LED marker impulse trains are picked up by an iniVation DVXplorer DVS camera [11], which feeds a Kapoho Bay neuromorphic compute device containing 2 Intel Loihi chips [4][12]. The Loihi logic identifies the LED marker impulse trains, determines the operator's hand position, encodes the 2D coordinates as a bundle of two-coordinate VSA symbols and transmits them using

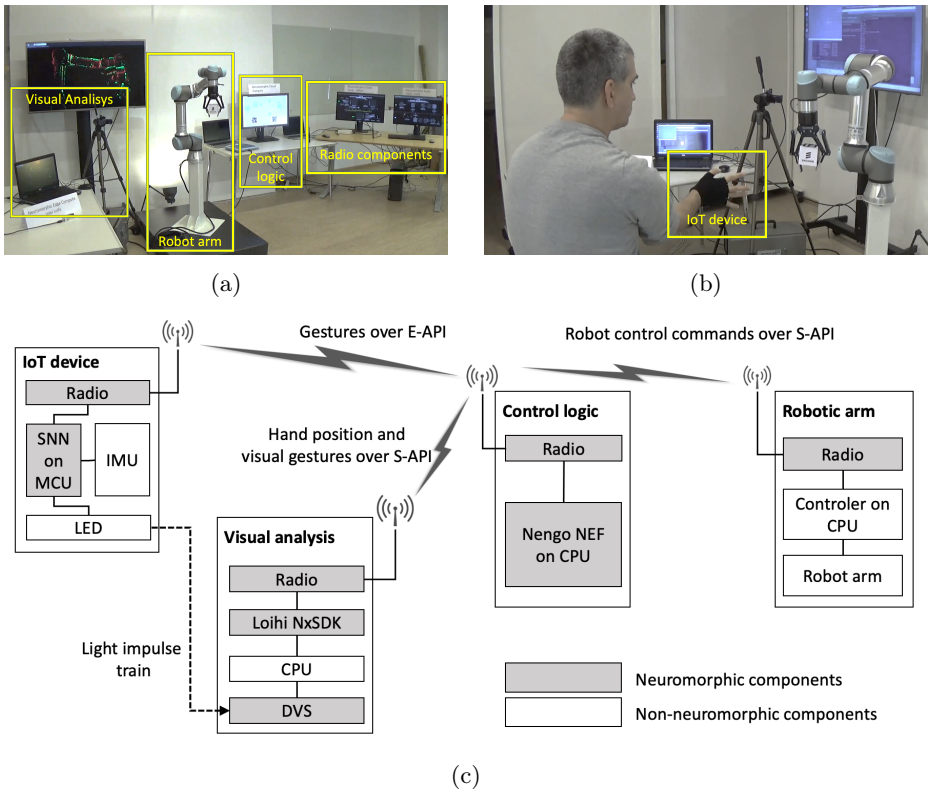


Fig. 2: The architecture of the end-to-end demonstration system and its physical realization in our lab: (a) the system overview with the main components highlighted, (b) a snapshot when the robot is controlled by the operator. The IoT device is embedded into the operator's glove, and (c) the system architecture is which the components running neuromorphic code are denoted with grey color. The content and the representation API of the radio messages are presented.

the symbolic S-API. The Loihi logic integrates all these functions into a single neural network. The S-API part was based on the description we explained in Section 2.2.

A central control logic receives both the IMU data as well as the position data in real-time. This central control logic runs a complex Nengo [13] based code. The Nengo logic architecture is shown in Fig. 3. The principle of operation takes ideas from Spaun, a functional large-scale model of the brain [14]. The position, IMU and other inputs received over S-API are placed on a state ensemble ("data bus") first. Action selection is performed by weighing the utility of possible actions and selecting one using the basal-ganglia-thalamus circuit model [15]. The actions are gated on the "bus" and buffered on the output using gated associative memory circuits. The output control parameters (arm position and

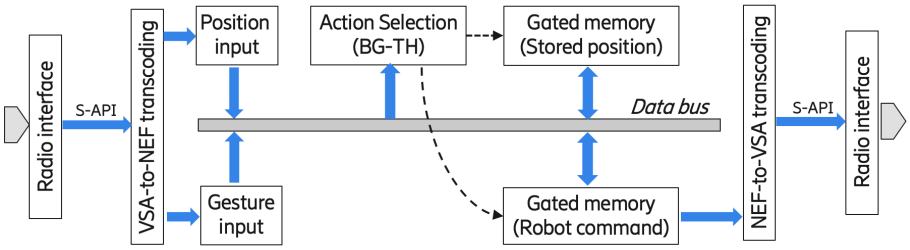


Fig. 3: Control logic in Nengo.

manipulator orientation) are then encoded symbolically and transmitted towards the robotic arm using the S-API.

The robotic arm is an industrial grade Universal Robots UR5 robot arm [16], which decodes the parameters and issues the commands directly to the arm hardware controller. The arm does not contain AI components (except an S-API capable radio and decoder), it exemplifies a traditional device communicating with neuromorphic counterparts.

3.2 Heterogeneity Aspects of the Prototype System

The system contains several devices and functional components (Table 1). These components show high degree of heterogeneity regarding their way of operation, complexity, communication requirements and internal architecture as well.

The first column lists the physically separate devices. Each contains several sub-components, but from a Data Layer point of view, they can be seen as one (albeit complex) device.

The second column shows the principal way of operation of the devices. The IoT device and the robot are regular non-AI components, even though they may send and receive AI encoded data. On the other hand, the visual analysis device contains two neuromorphic components: DVS and Loihi. The control logic is based on the Nengo Neuromorphic Framework [13] running on a PC. The robotic arm contains a regular PC-based controller and the physical UR5 robotic arm. The third column shows the component's way of operation, in particular whether the device's primary computational mode is neuromorphic or not. The fourth column shows that information is encoded in different ways in all components.

Finally, the last column shows how the component communicates with its peers. The control logic is the most complex in this regard, as it uses both the E and S-APIs, and it sends and also receives data over these APIs.

Table 1: Components used in the demonstration.

Device	operation	neuromorphic	encoding	Data Layer API
IoT device	MCU + LED + IMU	mixed	delay based spike	E-API
Visual analysis	DVS + Loihi NxSDK	yes	raw spike, one-hot	S-API
Control logic	Nengo NEF	yes	NEF ensembles	E/S-API
Robotic arm	CPU + UR5	no	Cartesian coord. system	S-API

4 Visual Analysis and Data Layer on Loihi

The compute part of the visual analysis component presented in the previous section is implemented on the Intel Loihi neuromorphic chip. Our implementation ranges from the injection of the event camera spikes into the Loihi, through the computing of the position of the IoT device, to the transforming of the position data into the radio conform VSA vector. The resulting VSA vector is then transmitted to the control logic component by a radio unit.

Implementing a concrete algorithm in the Loihi chip needs practical considerations to fit the problem with the chip's resource constraints like the total number of neurons assigned to a neurocore or the total number of synaptic states mapped to the cores [4].

The details of the implementation logic is presented in Fig. 4. The input data and the processing logic are distributed into separated neurocores. To fit the processing logic into a Loihi chip the full camera resolution (640×480) is re-scaled and cropped to 120×120 pixels. The input data is then split into 36 sub-regions containing 20×20 pixels each and their processing is mapped to separated neurocores. Each sub-region determines the estimated position of the LED by integrating the incoming spikes per pixel. The integrated pixels are connected into a regional R_x and R_y flattening neuron layer (residing on the same neurocores). These regional R_x and R_y layers are connected to a global X and Y flattening layer (located on a dedicated neurocore). The global flattening layer is responsible to compute the final numerical result (the x and y coordinates of the LED) needs to be communicated to the other devices. In these flattening layers (containing 120 neurons each) we use lateral inhibition between the neurons to shape the resulting information into a one-hot encoded representation of the coordinates. By the construction of the neural layers we ensure that neurons representing the x, y coordinate values belonging together are spiking simultaneously.

The neurons of the x and y global flattening layers are mapped to the single S-API neural layer consisting 1000 neurons matching the dimension number of the VSA space we use in the communication. This way we create a multi-dimensional S-API interface in which we simultaneously encode and transmit both coordinate values. We generate M_x and M_y encoding matrices for both

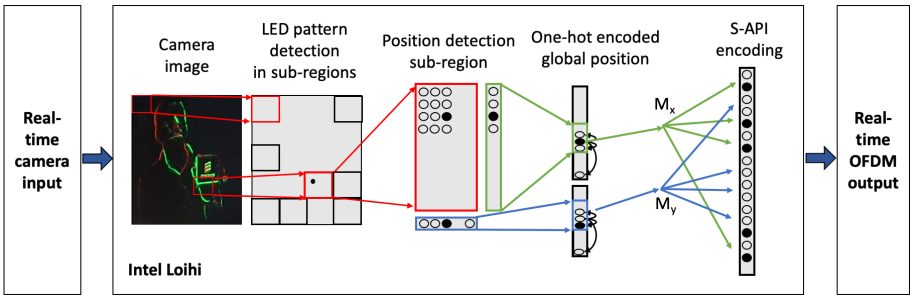


Fig. 4: Processing flow from the real-time camera input through the neuromorphic compute steps running on the Loihi chip to the real-time radio output.

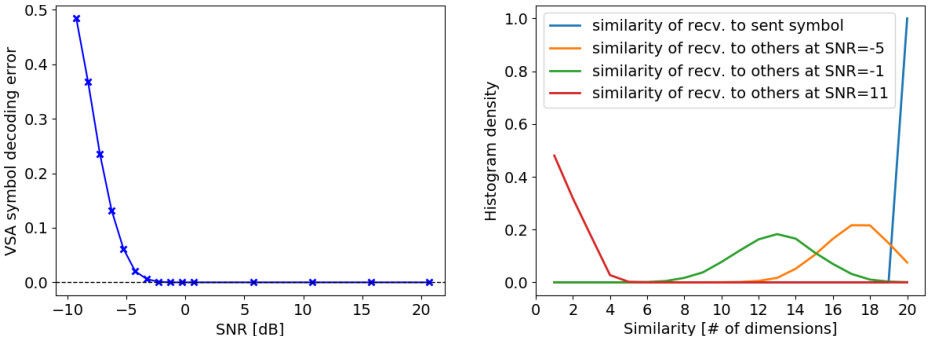
the x and y coordinates separately in which each one-hot encoded coordinate is mapped to a vector containing altogether 20 pieces of 1s at randomly selected positions as we discussed in Section 2.4. The one-hot encoded coordinate values correspond to the single neurons of the x or y flattening layer, the joint VSA space corresponds to the S-API neural layer. The M_x and M_y encode matrices, as the connection matrices, map the x and y flattening neural layers to the same S-API neural layer.

Finally the activity of the S-API neurons are collected and translated into a binary representation to be transmitted in a communication channel. The collection of the neural activities can rely on a timer that starts counting from the first spike after the last message transmission or on counting the spiking neurons and use this as a trigger to start the transmission and to reset the spike counters to zero. The parameters of the neurons of the S-API layer are set in a way that the information is carried by only a single spike firing per neuron and the spikes fired by different neurons are firing at the same time (in the same time-step) synchronously. Thus the communication component will be able to transmit these matching spikes simultaneously.

On the receiver side, the radio module decodes the radio signal as a received joint VSA symbol and injects that over the S-API to a Loihi input layer containing 1000 neurons (the received VSA vector). The receiver neurons are configured to have fast decay to ensure that the neurons are spiking simultaneously on each received VSA vector and consecutive transmissions do not mix. The receiving S-API neural layer is connected to two application level neural layers (one for processing x and one for y coordinate values) through their pre-shared M_x^T and M_y^T matrices used as connection matrices between the neural layers. This way the x and y values are retrieved separately after a winner take all algorithm selects the most similar symbols to the received ones.

5 Empirical Results

Next, two performance related aspects are investigated of the end-to-end prototype. One aspect was the symbol decoding error over various radio physical



(a) Symbol error vs. SNR

(b) Similarity distribution of received symbols

Fig. 5: VSA symbol decoding performance

layer implementations. We custom built two physical layers specifically for the transmission of E-API and S-API data: an Orthogonal Frequency-Division Multiplexing (OFDM) and an Ultra-Wideband (UWB) radio. The VSA symbol decoding error at different measured Signal to Noise Ratio (SNR) levels are shown in Fig. 5a for the case of binary encoded VSA over OFDM. This communication link is used to transfer the human hand's coordinates as detected by the neuromorphic camera and sent towards the central control logic. If a symbol decoding error happens over this link, it can result in a wrong position, impacting the complete robot control application. As we see in Fig. 5a the detection is completely error free in a large part of the SNR range and starts to degrade only when operating well below the noise level.

The reason for the decoding degradation can be observed in Fig. 5b, which shows that the similarity of the received symbol to all other symbols increases as the SNR decreases making it more difficult to decide on the correct symbol.

The other investigated aspect was the latency budget in the system. The visual analysis in the Loihi implementation can be subdivided into several latency contributors: the LED light train integration took $10ms$, the position decision and VSA encoding was done in $5ms$, altogether the Loihi component took approximately $15ms$. The OFDM radio transmission took around $10ms$, while the UWB was slower, around $50ms$. For the Nengo based code we could not use HW accelerator (we only had one Loihi), so we had to trade-off between position noise and latency, due to the relatively large size of the Nengo network.

Overall, our observation was that the system was fairly robust. Symbol decoding errors were rare, and the control-loop felt fairly responsive even though we did not have the resources and time to optimize for neither in this early prototype. We argue that NDL is flexible enough to build a wide range of practical distributed neuromorphic systems.

6 Conclusion

The Neuromorphic Data Layer serves as a native communication layer for distributed and heterogeneous neuromorphic applications by defining two APIs E and S-API. Devices based on heterogeneous architectures can use these APIs to exchange information regardless of their internal data representation. We presented a practical Loihi-based implementation of the S-API for integer variables. Our end-to-end industrial prototype demonstrates the feasibility of the proposed concepts. We believe that such a common data layer among the vastly heterogeneous neuromorphic and AI applications will be essential in the future in order to open the way to many new and interesting neuromorphic applications.

Acknowledgements The authors would like to thank the INRC [17] making us the Loihi cloud and the Kapoho Bay device available.

References

1. Plate, T.A. "Holographic reduced representations", IEEE Transactions on Neural Networks 1995;6(3):623-41.
2. Kenny Schlegel, Peer Neubert, and Peter Protzel, "A Comparison of Vector Symbolic Architectures", Artificial Intelligence Review (15/12/2021)
3. C. Eliasmith, C. H. Anderson, Neural engineering: Computation, representation, and dynamics in neurobiological systems, MIT press, 2004.
4. Davies et al. "Loihi: A neuromorphic manycore processor with on-chip learning." IEEE Micro 38.1 (2018): 82-99.
5. T. Borsos, M. Condoluci, M. Daoutis, P. Haga and A. Veres, "Resilience Analysis of Distributed Wireless Spiking Neural Networks", IEEE Wireless Communications and Networking Conference 2022
6. Neubert, Peer, Stefan Schubert, and Peter Protzel, "Learning vector symbolic architectures for reactive robot behaviours" Workshop on Machine Learning Methods for High-Level Cognitive Capabilities in Robotics held in conjunction with the International Conference on Intelligent Robots and Systems (IROS), 2016.
7. Neubert, P., Schubert, S., Schlegel, K. & Protzel, P. "Vector Semantic Representations as Descriptors for Visual Place Recognition", In Proc. of Robotics: Science and Systems (RSS). 2021
8. Stewart, T. and Tang, Y. and Eliasmith, C., "A biologically realistic cleanup memory: Autoassociation in spiking neurons", Cognitive Systems Research 2011
9. Brent Komer and Chris Eliasmith, "Efficient navigation using a scalable, biologically inspired spatial representation", In the Proceedings of 42nd Annual Meeting of the Cognitive Science Society 2020
10. Komer, Brent, et al. "A neural representation of continuous space using fractional binding." CogSci. 2019.
11. iniVation DVXplorer DVS Camera, https://inivation.com/wp-content/uploads/2021/08/2021-08-iniVation-devices-Specifications.pdf
12. Davies et al. "Advancing neuromorphic computing with Loihi: A survey of results and outlook." Proceedings of the IEEE 109.5 (2021): 911-934.

13. Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T.C., Rasmussen, D., Choo, X., Voelker, A.R., Eliasmith, C.: "Nengo: a python tool for building large-scale functional brain models", Frontiers in neuroinformatics 7 (2013)
14. C. Eliasmith, T. C. Stewart, X. Choo, T. Bekolay, T. DeWolf, Y. Tang, D. Rasmussen, "A large-scale model of the functioning brain", Science 338 (2012) 1202–1205.
15. Stewart, Terrence & Choo, Xuan & Eliasmith, Chris. "Dynamic Behaviour of a Spiking Model of Action Selection in the Basal Ganglia" 10th International Conference on Cognitive Modeling, 2010
16. Universal Robots UR5, https://www.universal-robots.com/products/ur5-robot/
17. Intel Neuromorphic Research Community https://newsroom.intel.com/news/intel-announces-neuromorphic-computing-research-collaborators/