Mapping Very Large Scale Spiking Neuron Network to Neuromorphic Hardware

Ouwen Jin
Zhejiang University
Hangzhou, China
jinouwen@zju.edu.cn

Qinghui Xing
Zhejiang University
Hangzhou, China
xingqh@zju.edu.cn

Ying Li
Zhejiang University
Hangzhou, China
cnliying@zju.edu.cn

Shuiguang Deng*
Zhejiang University
Hangzhou, China
dengsg@zju.edu.cn

Shuibing He
Zhejiang University
Hangzhou, China
heshuibing@zju.edu.cn

Gang Pan*
Zhejiang University
Hangzhou, China
gpan@zju.edu.cn

ABSTRACT

Neuromorphic hardware is a multi-core computer system specifically designed to run Spiking Neuron Network (SNN) applications. As the scale of neuromorphic hardware increases, it becomes very challenging to efficiently map a large SNN to hardware. In this paper, we proposed an efficient approach to map very large scale SNN applications to neuromorphic hardware, aiming to reduce energy consumption, spike latency, and on-chip network communication congestion. The approach consists of two steps. Firstly, it solves the initial placement using the Hilbert curve, a space-filling curve with unique properties that are particularly suitable for mapping SNNs. Secondly, the Force Directed (FD) algorithm is developed to optimize the initial placement. The FD algorithm formulates the connections of clusters as tension forces, thus converts the local optimization of placement as a force analysis problem. The proposed approach is evaluated with the scale of 4 billion neurons, which is more than 200 times larger than previous research. The results show that our approach achieves state-of-the-art performance, significantly exceeding existing approaches.

CCS CONCEPTS

• **Computer systems organization** → *Neural networks*; • **Networks** → Network on chip; • **Hardware** → **Neural systems**.

KEYWORDS

Neuromorphic computing, Spiking Neural Networks (SNN), Network on chip (NOC), mapping

ACM Reference Format:
Ouwen Jin, Qinghui Xing, Ying Li, Shuiguang Deng, Shuibing He, and Gang Pan. 2023. Mapping Very Large Scale Spiking Neuron Network to Neuromorphic Hardware. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3582016.3582038

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9918-0/23/03…$15.00
https://doi.org/10.1145/3582016.3582038

1 INTRODUCTION

Neuromorphic computing promises to realize artificial intelligence while reducing energy requirements [30]. Spiking Neural Networks (SNNs) [25], usually regarded as the third generation of neural networks, play an important role in neuromorphic computing. Imitating the biological nervous system in terms of neuron and synaptic connection models, SNN features rich spatial-temporal information and high biological plausibility.

To exploit the advantage of the low power consumption of spike-based artificial intelligence, many neuromorphic hardware have been developed. For example, DYNAP-SE [28], TrueNorth [8], Neurogrid [5], SpiNNaker [12], Loihi [7], Tianji [35] and Darwin [24]. All of these neuromorphic hardwares share a common design rule to run SNN applications. That is: using a large number of specially designed neurosynaptic cores (for example, crossbars on Loihi; and ARM cores on SpiNNaker) to store synaptic weights and simulate neurons dynamics in parallel.

In this kind of multicore architecture, SNN applications are required to be mapped into hardware prior to their execution. A typical process of mapping an SNN application into hardware needs: 1) partitioning neurons in SNN into several clusters to meet the hardware constraints, and 2) placing these clusters in neuromorphic computing cores.

A few approaches, such as PACMAN [13], PyCARL [2], SpiNeMap [4] and DFSynthesizer [36], have been proposed to address mapping SNNs to neuromorphic hardware. The works show that mapping results can dramatically influence the performance of the running of SNN applications, including power consumption, latency, and throughput.

However, existing methods show limitations with the increase of hardware scale (As shown in table 1) and the development of larger-scale SNN applications [18, 22]. These limitations are due to two main reasons:

- Most previous works mainly focus on optimizing the partition of neurons, while few addressed the placement of clusters. However, with the increasing number of cores, the placement of partitioned clusters shows a more significant impact on performance than the partitioning of neurons.

Table 1: Capacity of several neuromorphic hardware platforms

	DYNAPs [28]	BrainScaleS [34]	Loihi [7]	SpiNNaker [12]	TrueNorth [8]
# Neurons/core	256	512	128	1000	256
# Synapses/core	16K	128K	500K	2K	262K
# cores/chip	1	1	1024	18	4096
# chips/system	4	8192	768	1M	64
	High-performance system				
# Neurons	1K	4M	100M	1B	64M
# Synapses	65K	1B	100B	200B	1T

- Existing methods are designed for relatively small scale (less than 5000 cores) neuromorphic hardware and lack scalability. These methods either fail to solve in a reasonable time or the solution is of inferior quality when the problem scale is enlarged.

To tackle the limitations, we propose an efficient mapping approach based on Hilbert Space-filling Curve (HSC) [16] and Force Directed (FD) algorithm. HSC is a space-filling curve that maps a 1D sequence into a 2D space. We found that HSC can provide the overall data flow layout and map connected neurons to nearby locations , which is desirable in mapping problems. FD algorithm is a novel optimization algorithm designed by us. By converting the connections between the cores into forces and analyzing the forces on the cores, the FD algorithm can efficiently optimize the location of the cores. Combining HSC with the FD algorithm, our approach is able to map very large scale SNN applications to neuromorphic hardware, aiming to reduce energy consumption, spike latency, and on-chip network communication congestion. The experimental results show that our method can solve and optimize the SNN mapping task of 4 billion neurons and millions of cores in 26 seconds, while the existing methods need more than 100 hours.

The main **contributions** of this paper are as follows:

- We are the first to apply Hilbert Space-filing Curve (HSC) to the SNN mapping problem. We explore why HSC has an advantage in mapping very large scale SNNs by concluding the unique properties of HSC.
- We propose the Force Directed (FD) algorithm to map very large scale SNN applications into neuromorphic hardware, reducing energy consumption, spike latency, and on-chip network communication congestion.
- Combining HSC with the FD algorithm completes our mapping approach. We evaluate our approach with a large scale of 4 billion neurons and millions of cores on a general neuromorphic hardware model. The results demonstrate the excellence of the proposed approach.

2 BACKGROUND AND RELATED WORK

2.1 Background of SNN Mapping Problem

Neuromorphic computing, an emerging computing paradigm using SNNs model, has the potential to drive the next wave of AI and is receiving increasing attention [30], for its characteristics of brain mimicking and energy efficiency.

An SNN application is an attempt to use SNN to implement machine learning approaches. SNNs can be trained to perform specific tasks by supervised or unsupervised methods like other traditional Artificial Neuron Networks (ANNs). While training refers to adjusting the synaptic weights between neurons. This survey [30] shows that SNN applications could provide competitive results in many machine learning tasks, including vision classification, reinforcement learning, and robotic autonomous control.

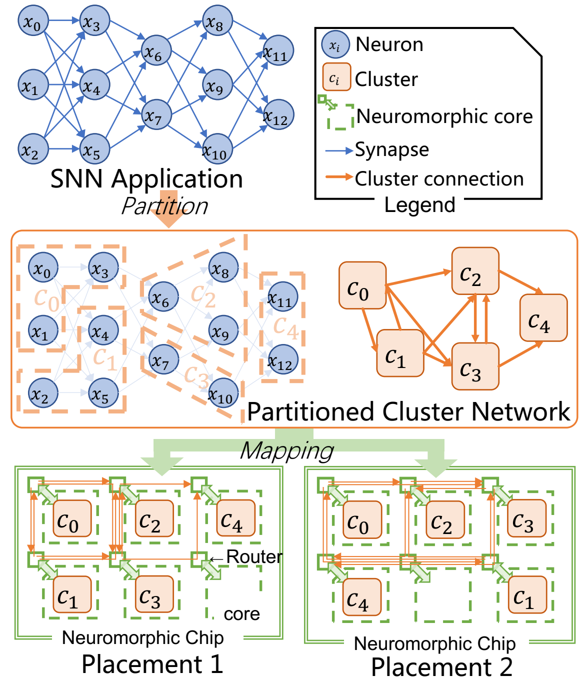


Figure 1: Mapping SNN into neuromorphic hardware

A neuromorphic hardware platform is a computer system specifically designed to implement SNN applications [27]. Table 1 lists several existing Neuromorphic hardware platforms. And many larger-scale neuromorphic hardware, e.g. SpiNNaker2 with 10 million core processors [26] is being designed and developed by companies and research institutions [11]. These hardware systems can be uniformly abstracted as multi-core computing systems that use many homogeneous neurosynaptic computing cores to simulate neuron dynamics in parallel. A typical implementation of the neurosynaptic computing core is the analog crossbar [21], while some choose to use digital processors [8, 12].

The most common design in the latest large-scale neuromorphic systems is Network-on-Chip (NOC) [23] with 2D mesh structures. The main advantage of this architecture is its high scalability to achieve massively parallel computing systems.

Based on this multicore architecture, SNN applications must be mapped into hardware. Mapping an SNN application into hardware means 1) partitioning neurons in SNN into several clusters to meet the hardware constraints and 2) placing these clusters in neuromorphic computing cores.

Figure 1 illustrates an example of mapping SNN applications into neuromorphic hardware. The input of the mapping problem is an SNN application, which is represented as a graph consisting

of neurons and synapses. Firstly, Neurons in the SNN application need to be partitioned into clusters due to the neurosynaptic core's capacity limitation. In Figure 1, the maximum number of neurons per core limited by hardware capacity is 3, so 13 neurons θ_{0-12} are partitioned into 5 clusters c_{0-4}. Based on the partition result, we can obtain the Partitioned Cluster Network (PCN), a graph made of clusters. The connections between clusters preserve the information of synapses that connect neurons partitioned in separate clusters.

The result of the mapping algorithm is a placement. Figure 1 shows two possible results of mapping given PCN into a 2×3 neuromorphic hardware. The green dotted box represents a neurosynaptic core, and the square in the upper left corner represents its router. One obvious conclusion is that placement 1 is a higher-quality solution than placement 2 because the path length of all connections in placement 1 is less than or equal to that in placement 2. The following section will give a more systematic and mathematical description of the problem.

2.2 Related Work

Mapping and scheduling problem is a well-known NP-hard problem [14], and this paper studies one instance of this type of problem. As a result, obtaining an optimized solution is extremely challenging. Based on this fact, existing mapping methods are often based on iterative optimization or greedy approaches to obtain approximate optimal solutions.

PACMAN [13] is the standard mapping method for SpiNNaker [12]. It employs a simple first come, first serve technique while considering user-specified constraints. TrueNorth [33] proposed a heuristic algorithm to place clusters to hardware cores layer-by-layer. Clusters of the input layer are placed in predefined positions. Then clusters in the following layers would choose a core with the minimum sum of distances from all adjacent inward clusters in preceding layers. DFSynthesizer [36] is initialized by randomly allocating clusters to neuromorphic cores. From this, it searches for a better solution by swapping clusters' positions iteratively. It evaluates throughput and energy cost every time clusters are moved and retains new mapping if the metric improves. SDFSNN [3] uses Synchronous Data-Flow Graphs (SDFGs) for mapping exploration. PSOPART [6] uses an instance of PSO (particle swarm optimization) to directly map neurons of SNN to neuromorphic cores. Based on PSOPART, SpiNeMap [4] adds a partition phase before using the PSO algorithm. Since a hardware core can contain no more than one cluster, SpiNeMap binarizes the position and velocity of particles. PyCARL [2] extends SpiNeMap's method to support the high-level programming interface PyNN. Song et al. [37] proposed a framework that also employs PSO to generate the SNN placement. Song integrated SDF^3 into PSO as its fitness function to calculate the throughput for a given mapping. In the framework, a scheduler is constructed to avoid placement conflict.

Previous research mainly focuses on heuristic methods to map neuron clusters to the neuromorphic hardware. A common problem with these methods is that they may be too slow to solve when the hardware scale is enlarged. These methods tested on a scale of fewer than 5000 cores mapping problems. Therefore, a more efficient mapping method is needed with the emergence of neuromorphic hardware such as SpiNNaker2 and Darwin3, which has a capacity of tens of billions of neurons and millions of parallel computing cores.

3 PROBLEM FORMULATION

3.1 Neuromorphic Hardware Model

In this paper, as shown in Figure 2, neuromorphic hardware is modeled as a many-core system made of a set of homogeneous processing cores connected by routers. Each router binds to a core and connects its neighbors in four directions with bidirectional links to form an interconnection network with a 2D mesh topology.

Each core, together with its binding routers, has coordinates of (x, y). The system size is defined by (N, M), which means a number of $N \times M$ cores are available in the system. The core at the top-left corner is indexed as $(0, 0)$, while the core at the bottom-right corner is indexed as $(N-1, M-1)$. We denoted the hardware system as a set

$$S = \{(x, y) \in \mathbf{N}^2 | 0 \le x < N, 0 \le y < M\}. \quad (1)$$

The capacity of processing cores in different neuromorphic hardware varies greatly. Two constants, CON_{npc} and CON_{spc}, are used to address this difference in hardware constraints. CON_{npc} defines the maximum number of neurons that can be configured per core, while CON_{spc} defines the maximum number of synapses that can be configured per core.

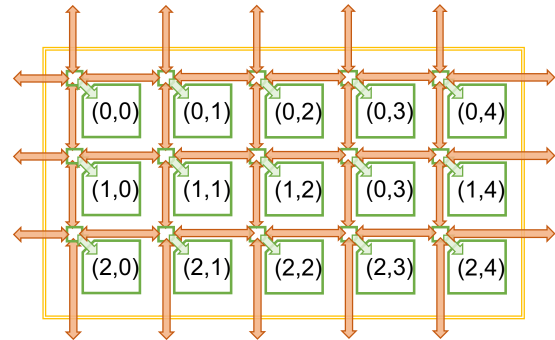


Figure 2: Hardware model

3.2 SNN Application and Partitioned Cluster Network (PCN) Model

An SNN application is naturally modeled as a directed Graph

$$G_{SNN} = (V_S, E_S, w_S). \quad (2)$$

Each node $x \in V_s$ presents a minimum computational unit: neurons. Each edge $e_{i,j} \in E_s$ presents a synapse connecting neuron x_i and x_j. $w_s : E_s \to \mathbf{R}$ is the edge weight function defined on the edges of the graph. It should be noted that the value of weights $w_s(e)$ does not represent the synapse weights in the SNN but the density of the spiking emitted by synapse e, or in other words, the communication traffic volume.

Neurons in SNN need to be partitioned into clusters to meet the hardware constraints before mapping algorithms can operate on

Algorithm 1: partition algorithm

```
   Input: G_SNN = (V_S, E_S, w_S)
   Output: G_PCN
1  c_latest = {};
2  foreach x_i ∈ V_S do
3  |  if Neuron x_i can not be partitioned into c_latest due to the
   |    hardware limitation then
4  |  |  V_P = V_P + c_latest;
5  |  |  c_latest = new{};
6  |  end
7  |  c_latest = c_latest + x_i;
8  end
9  build E_p and w_P based on G_SNN;
10 G_PCN = (V_P, E_P, w_P);
```

them. In this paper, we use algorithm 1 to do partition. The output of the partition algorithm is the Partitioned Cluster Network (PCN)

$$G_{PCN} = (V_P, E_P, w_P), \tag{3}$$

which is defined by a directed Graph. Each node

$$c_i = \{x | x \text{ is partitioned in } c_i\} \in V_p, \tag{4}$$

presents a cluster of neurons, whose self is a set made of the neurons it contains. A partitioned cluster can be mapped on an arbitrary core in neuromorphic hardware, which means it is the smallest unit that the mapping algorithm can schedule. Each edge $e_{i,j} \in E_P$ presents a communication connection between clusters c_i and c_j. The weight function w_P is given by the following equation:

$$w_P(e_{i,j}) = \sum_{e_{u,v} \in B_{i,j}} w_S(e_{u,v}), \tag{5}$$

where

$$B_{i,j} = \left\{ e_{u,v} \in E_S \,\middle|\, x_u \in c_i \text{ and } x_v \in c_j \right\}, \tag{6}$$

which presents the communication traffic volume between clusters, and it is proportional to the total number of spikes pass through this connection.

3.3 Problem Definition

Based on the system and application models, the mapping problem can be modeled as finding a placement $P : V_P \longrightarrow S$, an injective function that maps the clusters into cores, which is noted as follow:

$$F(G_{PCN}, S) = P \tag{7}$$

$$P(c_i) = (x_i, y_i), \tag{8}$$

where F is a given mapping algorithm.

To quantify the output placement, we use five metrics to optimize the mapping task, that is, energy consumption, latency, and network congestion. These metrics are computed as follow:

(1) *Energy Consumption:* The total energy consumed by all spikes on interconnect given $G_{PCN} = (V_P, E_P, w_P)$ and Placement P is computed as follow

$$\begin{aligned} M_{ec} = \sum_{e_{i,j} \in E_p} & (w_P(e_{i,j}) \left(\left\|P(c_i) - P(c_j)\right\| + 1\right) EN_r \\ & + w_P(e_{i,j}) \left\|P(c_i) - P(c_j)\right\| EN_w), \end{aligned} \tag{9}$$

where $\| \cdot \|$ is the $L1$ norm, which gives the Manhattan distance between two cores, EN_r is the energy consumption for a router route one spike message, and EN_w is the energy consumption for one spike message transmitted through a wire between routers.

(2) *Average Latency:* The average time a spike message spent on transmission in interconnect network, given $G_{PCN} = (V_P, E_P, w_P)$ and Placement P, is computed as follow

$$\begin{aligned} M_{al} = \sum_{e_{i,j} \in E_p} & (w_P(e_{i,j})(\|P(c_i) - P(c_j)\| + 1)L_r \\ & + w_P(e_{i,j})\|P(c_i) - P(c_j)\|L_w) \\ & / \sum_{e_{i,j} \in E_p} w_P(e_{i,j}), \end{aligned} \tag{10}$$

where L_r is the delay for a router route one spike message, and L_w is the delay for one spike message transmitted through a wire between routers.

(3) *Maximum Latency:* The maximum time spike messages spent on transmission in interconnect network among all connection routes, given $G_{PCN} = (V_P, E_P, w_P)$ and Placement P, is computed as follow

$$\begin{aligned} M_{ml} = \max_{e_{i,j} \in E_p} & ((\|P(c_i) - P(c_j)\| + 1)L_r \\ & + \|P(c_i) - P(c_j)\|L_w). \end{aligned} \tag{11}$$

(4) *Average Congestion:* The average congestion in the interconnection network, given the $G_{PCN} = (V_P, E_P, w_P)$ and Placement P, is computed as follow

$$M_{ac} = \sum_{(x,y) \in S} Con(x, y)/(N * M), \tag{12}$$

where $Con(x, y)$ compute the congestion in router whose coordinates is (x, y).It is compute as follow

$$\begin{aligned} Con(x, y) = \sum_{e_{i,j} \in E_P} & (w_P\left(e_{i,j}\right) \\ & * Expe\left(x, y, P(c_i), P(c_j)\right)), \end{aligned} \tag{13}$$

where $Expe(x, y, (x_s, y_s), (x_t, y_t))$ is the function that compute the expected value of number of spike pass through coordinate (x,y) when one spike is transmitted from coordinate (x_s, y_s) to (x_t, y_t). The complete calculation of this function is presented in algorithm 4 in the appendix.

(5) *Maximum Congestion:* The maximum congestion in the interconnection network, given the $G_{PCN} = (V_P, E_P, w_P)$ and Placement P, is computed as follow

$$M_{mc} = \max_{(x,y) \in S} Con(x, y). \tag{14}$$

4 THE PROPOSED APPROACH

4.1 Overview

Figure 3 illustrates the main work flow of our proposed approach. The process flow consists of two steps: step 1) using Hilbert Space-filling Curve (HSC) to obtain an initial placement, and step 2) perform Force directed (FD) algorithm to optimize the placement.

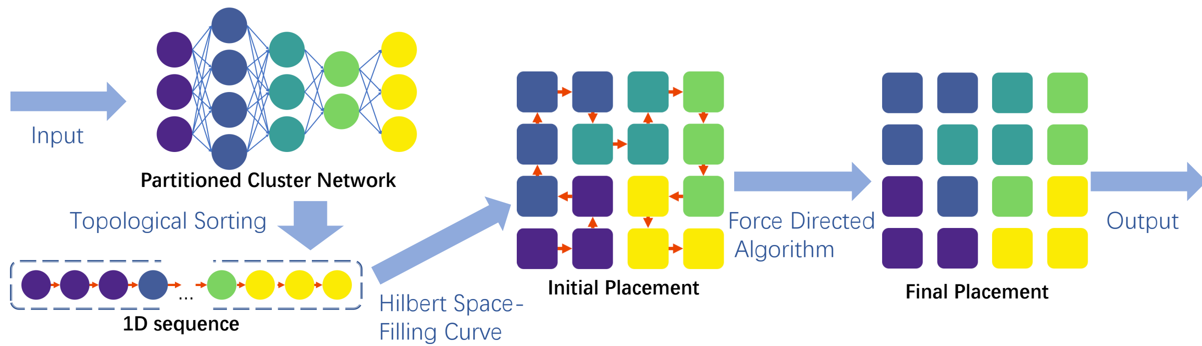


Figure 3: Diagram of the proposed approach

4.2 Initial Placement Based on Hilbert Space-filling Curve

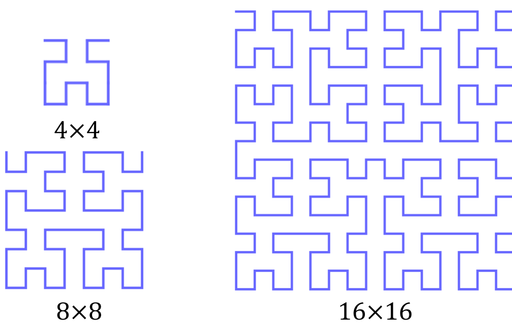


Figure 4: Some instances of Hilbert space-filling curve

4.2.1 Motivation. In our analysis, the most critical problem that existing heuristic search methods cannot map large-scale networks is that the solution space grows too fast with the size. The number of possible solutions for a mapping task with N kernels is up to $N!$. It is too difficult for existing methods to approach optimal solutions from a random starting point in solution space. So we wondered if we could start from a better initial placement rather than a completely random one.

Figure 4 shows some instances of HSC. HSC is a well-known fractal graph that provides a mapping relationship between 1D and 2D space. We find that HSC has very special properties, which can help to obtain a good initial placement.

4.2.2 Properties of HSC. We use the HSC as a guide to achieving an initial placement because it has the following properties.

(1) *Locality:* The HSC is widely used in computer science mainly because it gives a mapping between 1D and 2D space that preserves locality well. The locality here means that two points closing to each other in one-dimensional space are also close to each other after being mapped in 2D space [16]. SNN applications also show another form of locality property: neurons are only connected to a few other neurons locally instead of being widely connected in the whole network. This locality can be found both in deep machine learning applications and biological nervous systems in nature [39]. Combining these two locality properties is that the neurons with connections will be mapped closely in 2D space based on HSC, which is favorable for the mapping problem.
(2) *Infinity:* HSC can give a mapping between 1D and 2D space in infinity order, which means there are no scalability issues when mapping very large-scale SNN applications to hardware.
(3) *Provide data flow layout:* Most SNN applications show directionality: data flow from input to output like a stream. As shown in the figure 5, the HSC maintains this directionality when mapping, providing a U-shaped overall layout and fully using all space. Better yet, an HSC is a fractal graph. That is, it has this property in any of its sub-graphs. Therefore, not only on a global scale but on any small local area, this property can help to obtain a better Placement. We believe this feature may be the most critical reason HSC can provide superior initial solutions for further optimization.

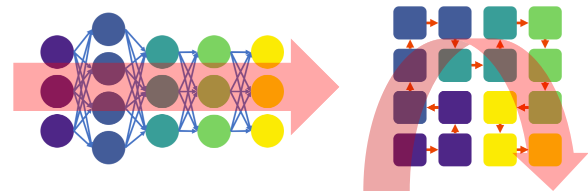


Figure 5: Data flow layout

4.2.3 Initial Placement with HSC. We achieve the initial placement through the following two steps: step 1) A sequence of clusters is obtained by a variety of topological ordering of the PCN graph. And step 2) a discrete approximation of Hilbert Space-filling Curve is used to map the 1D sequence into the 2D mesh.

In step 1), the topological order of the PCN graph is generated by algorithm 2, which is the same as the classical topological sorting algorithm, except for a slight modification to enable it to handle non-Directed-acyclic-graphs (non-DAG). The algorithm's output is

Algorithm 2: Topological sorting

```
Input: G_PCN = (V_P, E_P, w_P)
Output: Seq : V_p → N
1  p = 0;
2  S = all clusters with no incoming edge;
3  Seq(c_i) = −1 for all c_i in V_P;
4  while p < |V_p| do
5      if S is not empty then
6          n = c_i with smallest index i in S;
7          remove n from S;
8      else
9          n = c_i with smallest index i that Seq(c_i) == −1
10     end
11     Seq(n) = p;
12     p = p + 1;
13     foreach c_j with an edge e_{i,j} ∈ E_P do
14         E_P = E_P − e_{i,j};
15         if c_j has no other incoming edge and Seq(c_j) == −1;
16          then
17             S = S + c_j
18         end
19     end
20 end
```

a function $Seq : V_p \rightarrow \mathbf{N}$.

$$Seq(c_i) = j, \tag{15}$$

presents that the cluster c_i is the jth item in the sequence of topological order.

In step 2), the sequence is mapped into 2D space using a discrete approximation of the Hilbert Space-filling Curve, which is noted as a function $Hilbert : \mathbf{N} \rightarrow (\mathbf{N}, \mathbf{N})$.

$$Hilbert(i) = (x_i, y_i), \tag{16}$$

presents that the ith item in the sequence is mapped to the position (x_i, y_i).

Combining steps 1 and 2 we obtain the function of the initial placement $P_{init} : V_p \rightarrow (\mathbf{N}, \mathbf{N})$.

$$P_{init} = Hilbert \circ Seq, \tag{17}$$

which essentially transforms the network, represented as a graph, into a 1D sequence, and then further maps it to 2D space based on the HSC.

4.3 A Statistical Analysis of HSC

Figure 6 provides a statistical analysis of the advantages of HSC curves applied to mapping problems.

Figure 6.a shows three examples of different space-filling curves on an 8*8 mesh, including HSC, Zigzag, and Circle [32]. The arrows in the figure indicate the arrangement of the original 1D sequence in 2D space based on the corresponding space-filling curves.

Figure 6.b shows the generated distance heatmaps based on curves in Figure 6.a. In the heatmap, the value at any point (x, y) is equal to the distance of the point pair (P_x, P_y), where P_i represents the position where the ith point in the 1D sequence is mapped to 2D space, based on the corresponding curve. The closer the point pair is, the smaller the value and the darker the color in the heatmap, and vice versa. The heatmap shows some features of the space-filling curve from a visual perspective: compared with other space-filling curves, the HSC's heatmap appears darker near the diagonal and has fewer sharp spikes of brightness. This phenomenon images the locality property of the HSC, i.e., points that are adjacent in 1D sequence will also be adjacent in 2D space.

Figure 6.c shows three connection images of SNN. In the connection image, the value of the point (x, y) indicates whether there is a connection between neuron x and neuron y in the neural network. If there is, the corresponding point is colored. Otherwise, it is blank.

Figure 6.d shows a Cost we designed to measure the performance of different curves. The specific calculation process is as follows: the connection image of SNN is covered on the heatmap of the space-filling curve like a mask, and then the desired cost is obtained by summing up the values of all the covered points in the heatmap. In fact, all the points to be counted are precise all the connections in the neural network, and their connection distances in 2D space are the values on the heatmap. These values can obtain the sum of all the connection distances in the mapped space, which is an important metric to measure the mapping performance.

In order to further obtain the average performance of these space-filling curves when mapping an arbitrary unknown SNN, we made the probability cloud image. The probability cloud is composed of many connection images of different SNNs. In the same way, we can obtain the cost of different curves on this point cloud map, and the results are shown in Figure 6.e.

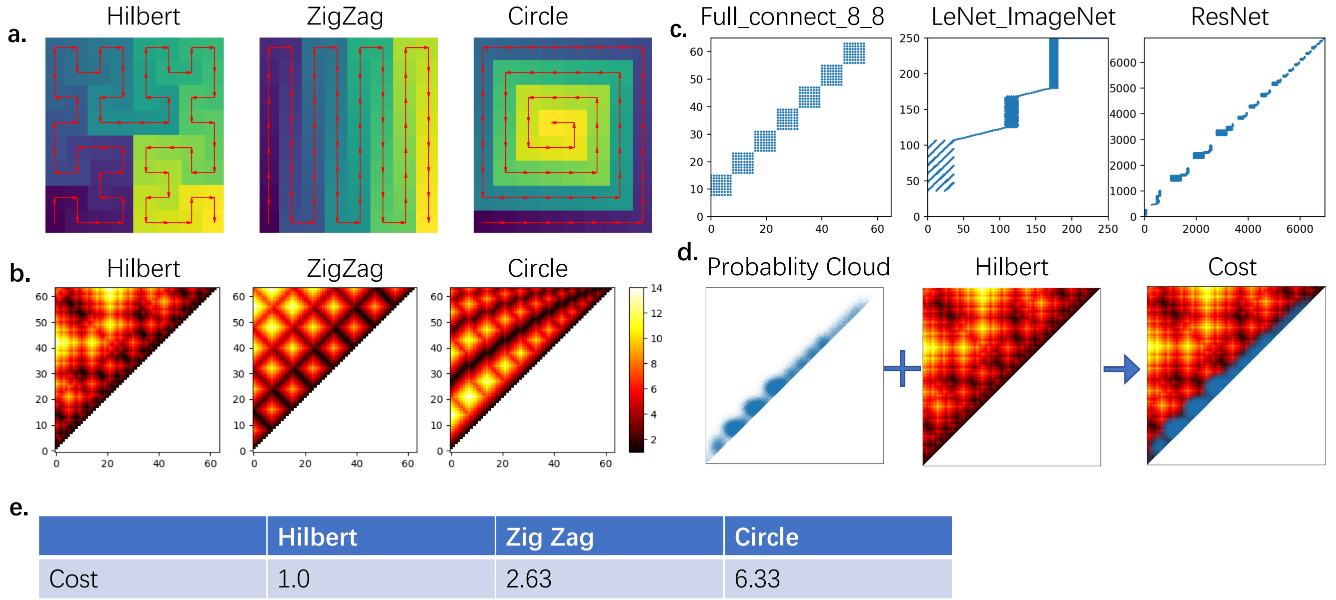
It can be seen from the results that HSC performs several times better than ZigZag and Circle on mapping neural networks in a statistical sense.

4.4 Force Directed Algorithm for Finetuning

4.4.1 Overview. The HSC provides a placement that only maps clusters at a macro level, so there is a large room for local optimization. Therefore, we propose the FD algorithm to finetune the placement provided by HSC.

The main idea of the FD algorithm is to regard clusters as particles on a 2D plane and the connection between clusters as the tension. In this way, in the dynamic evolution of the physical model, particles subjected to greater tension will get closer, achieving the desired result: clusters with greater connections will be mapped to closer positions.

The design of the FD algorithm also takes advantage of the locality of the SNN applications. It is embodied in two points: 1) The force of each cluster is only related to clusters connected to it. Therefore, when the algorithm updates the position of a cluster in the placement, only clusters connected to it need to be maintained. The locality of SNN ensures that the number of affected clusters is not too large, thus ensuring the high efficiency of computation. 2) Based on the placement given by HSC, the locality of the SNN applications ensures that clusters with connections are mapped in close positions. This property means that each cluster is not too far from the optimal position, ensuring that the FD algorithm's strategy based on local tuning does not need to be executed too many times.



	Hilbert	Zig Zag	Circle
Cost	1.0	2.63	6.33

Figure 6: Why HSC space filling curve. a) Space-filling Curve. b) Distance-Heatmap. c) Network. d) Cost Computation. e) Cost of different space-filling curves

4.4.2 Details of FD Algorithm. To perform the FD algorithm, we first build an abstract physical model. Although we borrow many physics concepts, this model does not have any real physics meaning but is entirely intended to serve the mapping algorithm.

We first define the *potential energy*. Given placement P and PCN graph $G_{PCN} = (V_P, E_P, w_P)$, the potential energy is defined as follow:

$$U_{c_i}(c_j, P(c_i), P(c_j)) = u(P(c_j) - P(c_i)) * w_p(e_{i,j}). \tag{18}$$

The equation (18) means that based on a given placement, the clusters c_i will generate a potential field. Any other cluster c_j in the system will therefore have a potential energy $U_{c_i}(c_j)$. The value of potential energy is determined by

(1) The position of c_j in the potential field generated by c_i, i.e., the relative position of c_i to c_j.The further away c_j is from c_i, the more potential energy c_j has.
(2) The connection weight between $c_i$and c_j. If there is no connection between c_i and c_j, c_i will not gain potential energy from c_j's potential field.
(3) The shape of the potential energy field.

Figure 7 visualized three potential energy fields, whose shape is determined by potential function $u(p)$. In case a), the potential function is designed most straightforwardly:

$$u_a(p) = |x_p| + |y_p|, \tag{19}$$

where $p = (x_p, y_p)$ is the position of the cluster relative to the origin of the potential field. This potential function defines a uniform potential field. The potential energy at position p is linearly proportional to the $L1$ norm of the position vector p, i.e., the Manhattan distance from p to origin.

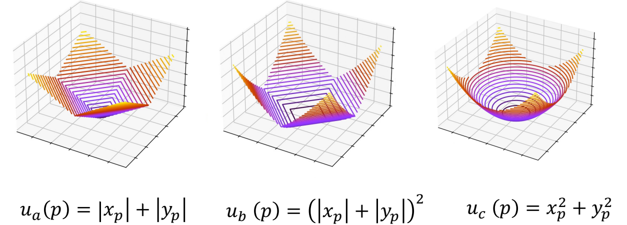


Figure 7: Different potential energy fields

The energy fields are not uniform in case b) and c) but is denser when away from the origin. This design gives clusters relatively more potential energy when away from the field's origin. In the subsequent optimization process of reducing the system's total energy, these pairs of clusters farther away will be preferentially pulled closer. Such property helps to reduce long-distance connections and therefore reduce latency. The potential energy at position p is proportional to the squared $L1$ norm and the squared $L2$ norm of p, respectively. The potential functions are designed as follows:

$$u_b(p) = (|x_p| + |y_p|)^2, \tag{20}$$

$$u_c(p) = x_p^2 + y_p^2. \tag{21}$$

There are many other possible potential field designs. The above three potential energy fields are chosen in this paper because of their computational efficiency and performance. And by selecting different potential field functions, our algorithm can flexibly tradeoff between solving speed and solution quality.

The total potential energy of a cluster E_{c_i} and system's total energy E_s can be computed as follow:

$$E_{c_i} = \sum_{c_j | e_{j,i} \in E_p} U_{c_j}(c_i, P(c_j), P(c_i)), \tag{22}$$

$$\begin{aligned} E_s &= \sum_{c_i \in V_P} E_{c_i} \\ &= \sum_{c_i \in V_P} \sum_{c_j | e_{j,i} \in E_p} U_{c_j}(c_i, P(c_j), P(c_i)) \\ &= \sum_{e_{i,j} \in E_p} U_{c_i}(c_j, P(c_i), P(c_j)). \end{aligned} \tag{23}$$

The target of the FD algorithm is to minimize the system's total energy by optimizing the placement:

$$\underset{P}{\operatorname{argmin}}\, E_s. \tag{24}$$

Equation (26) shows that minimizing the system's total energy is equivalent to minimizing the system's total energy consumption when using a specially designed potential function (25).

$$u(p) = (||p|| + 1)EN_r + ||p||EN_w \tag{25}$$

$$\begin{aligned} E_s &= \sum_{e_{i,j} \in E_p} U_{c_i}(c_j, P(c_i), P(c_j)) \\ &= \sum_{e_{i,j} \in E_p} u(P(c_j) - P(c_i)) * w_p(e_{i,j}) \\ &= \sum_{e_{i,j} \in E_p} ((||P(c_j) - P(c_i)|| + 1)EN_r + \\ &\quad ||P(c_j) - P(c_i)||EN_w) * w_p(e_{i,j}) \\ &= M_{ec}. \end{aligned} \tag{26}$$

In our model, *force* is defined as the reduction in potential energy of a cluster as it moves towards its neighbor position. A cluster will be subjected to different forces in four directions: up, down, left, and right. The force is computed as follows:

$$\begin{aligned} &Force_{c_i,d} \\ &= E_{c_i} - E'_{c_i} \\ &= E_{c_i} - \sum_{c_j | e_{j,i} \in E_p} U_{c_j}(c_i, P(c_j), P(c_i) + \nabla p_d), \end{aligned} \tag{27}$$

where

$$d \in \{UP, DOWN, LEFT, RIGHT\} \tag{28}$$

$$\nabla p_d = \begin{cases} (-1, 0) & d = UP \\ (1, 0) & d = DOWN \\ (0, -1) & d = LEFT \\ (0, 1) & d = RIGHT \end{cases}. \tag{29}$$

The equation (27) shows that when the cluster moves in one direction, the force in that direction is positive if the potential energy is reduced and negative otherwise. This fact indicates that in order to reduce the system's total energy, clusters tend to move in the direction of greater force.

Tension is defined on the pair of adjacent clusters. It is computed as follows:

$$\begin{aligned} &Tension_{c_i,c_j} = Force_{c_i,d_{ij}} + Force_{c_j,d_{ji}}, \\ &\text{where} \quad ||P(c_i) - P(c_j)|| = 1, \end{aligned} \tag{30}$$

where d_{ij} representing direction from $P(c_i)$ to $P(c_j)$. The tension's value equals the sum of the forces of the two clusters pointing toward each other. According to the definition of force (27), the tension's value is equal to the sum of the potential energy reduced by switching the positions of the two clusters.

Based on the above model definition, we propose the FD algorithm. Algorithm 3 illustrates the complete workflow of the FD algorithm. First, the input placement P_{init} is set as the baseline of the optimization (line 2). Based on initial placement, the $Force$ array is built using equation (27) (lines 3-5). The $Tension$ array is built using equation (30) (lines 7-8). Meanwhile, a list L of all pairs with positive tension is constructed (lines 9-11). The list L is then sorted into a queue where the higher the tension, the more advanced the pair (line 13). The FD algorithm then optimizes placement by swapping pairs in the queue (lines 14-41). In each iteration, a new queue will be generated for the next iteration. The algorithm recursively iterates until the new queue is empty (line 14). In each iteration, a part of the pairs in the queue will be swapped. The proportion of this part is controlled by hyperparameter λ (line 17). Before being swapped, a pair has to be checked to ensure its tension is still positive. If not, the pair is discarded (lines 18-19). In the swapping process, the clusters mapped to these two positions are swapped, and their $Force$ values are rebuilt (lines 20-21). The $Force$ of all other clusters connected to these two clusters is maintained (line 24). Finally, all clusters affected by the swapping process are added to a list $L_{affected}$ (line 25). When all the swaps are complete, the algorithm starts building the queue for the next iteration L_{next} (lines 30-40). L_{next} is first initialized to contain all pairs in the queue for this iteration (line 15). All pairs containing the affected clusters are then appended to L_{next}. After rebuilding the tensions of all pairs in L_{next}, all pairs with non-positive tensions are discarded (lines 36-38). Finally, the queue required for the next iteration is obtained by sorting L_{next} in order of tension (line 40).

4.5 Design Choices of FD Algorithm

In this section, we motivate the FD algorithm's design choices.

(1) *Check before the swapping process.* Pairs in the queue are assured of positive tension when the queue is constructed. However, during a series of swaps in one iteration, a previous swap may cause a change in the tension of the subsequent swaps. Before being swapped, a pair of clusters has to be checked to ensure its tension is still positive. This checking procedure is used to ensure the convergence of the algorithm. According to equations (30) and (27), the change of the system's total energy is equal to the tension of the pair before the swap. The positive tension ensures the system's total energy decreases monotonically.

$$E_s^* = E_s - Tension_{c_i,c_j} < E_s, \tag{31}$$

where E_s^* is the new system's total energy after the swap. Another possible method is to dynamically maintain the queue, i.e., updating all pairs' tension after each swap. However, this method will bring high algorithm complexity. We choose to use a static queue and check in terms of algorithm efficiency.

(2) *Hyperparameter λ.* The main idea of introducing the hyperparameter λ is to mimic the dynamic evolution of a physical

```
Algorithm 3: Force Directed
   Input: P_init, G_PCN = (V_P, E_P, w_P), S, λ
   Output: P_final
 1 Define: UP, DOWN, RIGHT, LEFT = 0, 1, 2, 3;
 2 P_now = P_init;
   /* Build array Force                                    */
 3 foreach p = (x, y) in S do
 4 |  build the Force[p][0..3];
 5 end
   /* Build array Tension and list L                       */
 6 L = empty List;
 7 foreach pair = (p_u, p_v) in 2D mesh that p_u and p_v is
     adjacent do
 8 |  build the Tension[pair];
 9 |  if Tension[pair] > 0 then
10 |  |  L.append(pair)
11 |  end
12 end
13 sort L s.t. ∀i < j, Tension[L[i]] ≥ Tension[L[j]];
   /* Iterating and optimizing                             */
14 while L is not empty do
15 |  L_next = L;
16 |  L_affected = empty List;
   |  /* switch the pair                                    */
17 |  foreach pair_i = (p_u, p_v) in L and i < λ|L| do
18 |  |  recompute tension[pair_i];
19 |  |  if Tension[pair_i] > 0 then
20 |  |  |  swap the placement of clusters currently placed
   |  |  |    at p_u and p_v in P_now;
21 |  |  |  rebuild Force[p_u][0..3] and Force[p_v][0..3];
22 |  |  |  foreach c_j in (c_u, c_v) do
23 |  |  |  |  foreach c_k with an edge e_j,k or e_k,j in E_P do
24 |  |  |  |  |  maintain Force[P_now(c_k)][0..3];
25 |  |  |  |  |  L_affected.append(c_k);
26 |  |  |  |  end
27 |  |  |  end
28 |  |  end
29 |  end
   |  /* clean up                                           */
30 |  remove duplicates from L_affected;
31 |  foreach c in L_affected do
32 |  |  append all pairs contains P_now(c) to L_next
33 |  end
34 |  foreach pair = (p_u, p_v) in L_next do
35 |  |  rebuild Tension[pair];
36 |  |  if Tension[pair] <= 0 then
37 |  |  |  remove pair form L_next
38 |  |  end
39 |  end
40 |  L = L_next; sort L;
41 end
42 P_final = P_now;
```

model, namely, a particle with a larger force will move faster. The hyperparameter λ determines a certain percentage of pairs in the front of the queue that could participate in the swapping process in an iteration. This property indicates that the algorithm will preferentially swap pairs with large tension. The value of λ cannot be too large or too small. A large λ indicates that almost all pairs can participate in the swapping process, making the algorithm less physically interpretable and reducing the placement quality. A low λ causes only a small number of pairs to be swapped in each iteration, increasing the total number of iterations required to achieve convergence, resulting in the low efficiency of the algorithm. Through a series of experiments, we give a practical value of λ: 30%, aiming to balance the efficiency and quality.

(3) *Introducing of $L_{affected}$.* We use a list $L_{affected}$ to record all affected clusters in an iteration. This method ensures that we get all pairs with positive tension after each iteration while performing as less tension computation as possible. This method is especially effective when the algorithm is close to convergence, where only a small number of clusters in the system have not reached the convergence position. The algorithm tracks these clusters and their related pairs through this list, thus avoiding the overhead of maintaining all pairs' tension, resulting in significant efficiency improvement.

5 EXPERIMENTS AND EVALUATIONS

5.1 Experimental Setting

Table 2: Parameters of target neuromorphic hardware

Parameter	Value
CON_{npc}	4096
CON_{spc}	64K
EN_r	1
EN_w	0.1
L_r	1
L_w	0.01

5.1.1 Experimental Environment. We use software to simulate an abstract target neuromorphic hardware platform to measure our proposed approach. The parameters for the target hardware platform are listed in Table 2.

The experiments are conducted on an Ubuntu 20.04.2 LTS (GNU Linux 5.8.0-59-generic x86 64) workstation with 40 CPU cores (Intel(R) Xeon(R) Silver 4210R CPU@2.40GHz), 256G memory and 4 GPU cards (GeForce RTX 3080).

We use GPUs to train and transform SNN applications. All mapping algorithms are implemented by C++ without GPU computing power or multicore parallel computing feature.

5.1.2 SNN Applications for Evaluations. Table 3 lists 13 SNN applications used for evaluation. Columns 2 and 3 of the table report the number of neurons and synapses of the SNN applications. Columns 4 and 5 report the number of clusters and connections of the corresponding PCN graph. Column 6 reports the scale of the target neuromorphic hardware.

Table 3: Benchmarks

Applications	G_{SNN} Neurons	Synapses	G_{PCN} Clusters	Connections	Target Hardware
DNN_65K	65536	805M	16	48	4×4
DNN_16M	16.7M	4T	4096	258048	64×64
DNN_268M	268M	70T	65536	4M	256×256
DNN_4B	4B	1125T	1M	67M	1024×1024
CNN_65K	65536	2M	16	48	4×4
CNN_16M	16.7M	528M	4096	16384	64×64
CNN_268M	268M	8B	65536	262K	256×256
LeNet-MNIST	9118	0.4M	9	19	3×3
LeNet-ImageNet	1.0M	188M	251	2151	16×16
AlexNet	0.9M	1.0B	229	4289	16×16
MobileNet	6.9M	0.5B	1688	37418	42×42
InceptionV3	14.6M	5.4B	3570	117597	60×60
ResNet	28.5M	11.6B	6956	478602	84×84

The first seven SNN applications starting with DNN or CNN are randomly synthetic. DNN stands for Deep Neural Network, which contains multiple layers of neurons, and each adjacent layer of neurons is fully connected. CNN stands for Convolutional Neural Network. The connections between neurons follow the classical convolutional network structure [19].

The following seven SNN applications are realistic Artificial Neural networks (ANN). They are: LeNet [20] trained in MNIST [10] and ImageNet [9] datasets, AlexNet [19], MobileNet [17], InceptionV3 [38], and ResNet [15]. We trained these networks using Tensorflow [1] and converted them into SNN form using the ANN-to-SNN conversion tool SNNToolBox [31].

5.1.3 Comparison Approaches. We mainly evaluate the following five approaches.

1. *The baseline:* Randomly mapping, clusters are randomly mapped into the neuromorphic hardware.
2. *Truenorth:* A method proposed in [33]. It is used in Truenorth [8], a large-scale neuromorphic computing system.
3. *DFSynthesizer:* A greedy mapping algorithm proposed in [36].
4. *PSO:* Particle Swarm Optimization is a classic optimization algorithm used by many mapping approaches [4, 36, 37]. We take the PSO configuration from this SOTA work [37].
5. *Proposed approach:* In the first part of the experiment, we show our method's performance under different configurations in detail. In the second part of the experiment, we compare our approach against other methods.

5.1.4 Evaluation Metrics. All the metrics for measuring the quality of the placement are detailed in section 3.3, including:

- Energy consumption (9)
- Average latency (10)
- Maximum latency (11)
- Average congestion (12)
- Maximum congestion (14)

Another metric we use to measure the efficiency of the technique is:

- Algorithm execution time

which measures how long it takes for a given technique to get a solution.

5.2 Performance of the Proposed Approach

The experiment consists of two parts. In the first part, we show the advantages of HSC compared with other space-filling curves and how HSC can cooperate with the FD algorithm to obtain optimal performance. In addition, the influence of different potential functions on the FD algorithm is also shown. In the second part, we use the benchmark to compare our approach against other existing methods.

Figure 8 shows the results of the first part of the experiment. The experiment is based on ResNet, a representative SNN application from the benchmark. The experiment involves ten mapping methods, marked a) to j). Where method a) is the baseline method, i.e., random mapping. Methods b), c), and d) directly obtain placement based on space-filling curves. e) to j) are six methods that contain the optimization phase of the FD algorithm, where methods e) and f) use the potential function described by equation 19 , while g) and h) use 20, i) and j) use 21. Methods e), g), and i) run the FD algorithm beginning with a randomly initialized placement. Methods f), h), and j) use the FD algorithm on an initial placement given by HSC. The left part of the figure shows the solution quality given by the ten methods through the designed metrics. All metrics are normalized to the baseline method. In the bar for latency and congestion, light and dark colors represent this metric's maximum and average values, respectively. The right part shows the solving time required by some methods. Since there is no iterative optimization process for methods a) to e), their time consumption is less than 0.001 seconds.

Based on the experimental results shown in Figure 8, we have the following observations:

1. HSC has significantly better performance than other space-filling curves.
   Other space-filling curves, ZigZag and Circle, will wander in a wide range of 2D space, which may lead to the adjacent and connected neurons being assigned to locations far away from each other, resulting in the solution's performance being even worse than the baseline method. While HSC, by its nature, is well suited for mapping SNN, which has been discussed before.
   Compared with the baseline, HSC's energy consumption is reduced by 77.3%; average and maximum delay is reduced by 64.2% and 62.4%, respectively; average congestion is reduced by 77.4%, while maximum congestion is increased by 12.6%. As for the increment of maximum congestion, we find that HSC places many densely connected neurons in a centralized area, resulting in local routing hotspots, which can be solved by subsequent optimization of the FD algorithm.
2. Neither HSC nor FD algorithm can complete the mapping task well alone.
   On the one hand, the HSC provides a placement that only maps clusters at a macro level, so there is a large room for local optimization. Based on the placement given by HSC, using the FD algorithm can further reduce energy consumption by 23.3%, average and maximum latency by 26.5% and

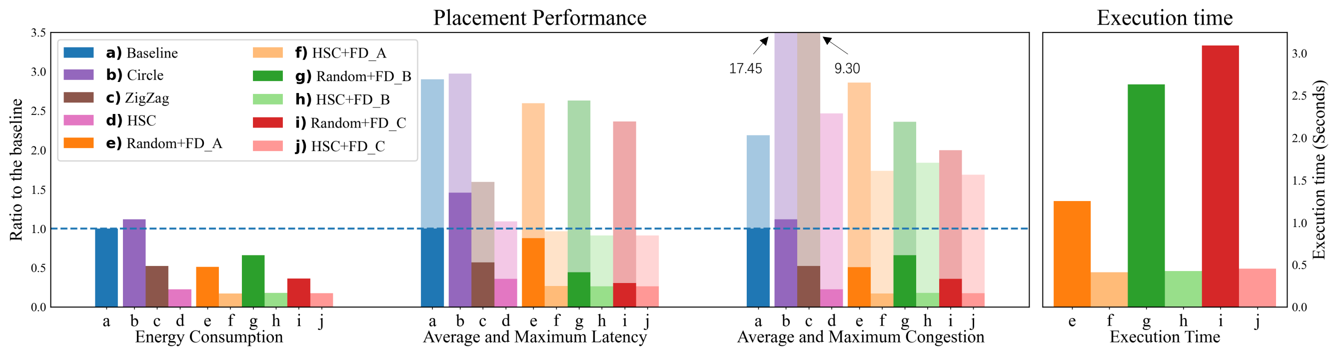


Figure 8: Performance of the space-filling curve and FD algorithm at ResNet

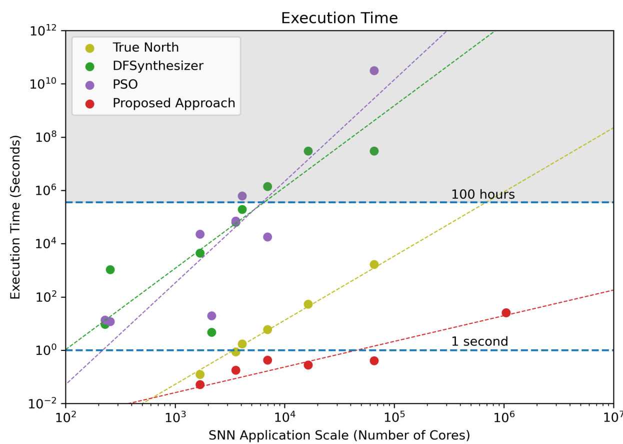


Figure 9: Results on execution time

16.4%, and average and maximum congestion by 23.5% and 31.6%. Reducing maximum congestion also solves the router hotspots problem involved by HSC.

On the other hand, the results show that the FD algorithm heavily depends on the quality of the initial solution. Without the initial solution provided by HSC, the metrics of the solution given by the FD algorithm become significantly worse, and the solution time becomes very long. According to our analysis, the huge performance reduction of the FD algorithm is due to the lack of macro guidance and dataflow layout provided by HSC. Each iteration of the FD algorithm only makes local optimization and adjustment, so it isn't easy to affect the overall layout. And without a proper initial placement, clusters will be far from the ideal converge position, increasing the time cost of convergence.

(3) Among the three methods f), h), and j), methods j) can achieve better solution quality.

These three methods represent the complete method proposed by us, i.e., HSC combined with FD. Method j) uses the potential energy function described by formula A, which can better punish the long-distance connection relation than other potential energy functions to obtain better quality solutions. Although the computing complexity of the formula is higher, this disadvantage is eliminated when using HSC. We analyze that the total number of iterations needed to converge is reduced due to the better optimization and adjustment strategy of method j).

Based on the above experimental observations, we let method j) represent our proposed approach to participate in all subsequent experiments.

5.3 Comparison with Other Approaches

In the second part, we present the major experimental results: the performance of the existing mapping method and our approach under the benchmark.

Figure 9 shows the time required by each method to solve SNN mapping problems of different scales. The X-axis indicates the size of the SNN application, which is represented by the number of neuron clusters. The Y-axis represents the time required for the method to execute. Notice that we are taking logarithms of both axes.

Since existing algorithms may not solve the very large-scale SNN mapping task in a reasonable time, some tasks in the dataset have no data points, and some data points are estimated. The data points in Figure 9 that took more than 100 hours (where painted gray) are our estimates, not actual data. It is estimated by forcing the algorithm to stop at 100 hours and then dividing the current time by the percentage of the number of iterations completed. Since the total number of iterations in PSO and DFSynthesizer is known, we can obtain the estimated time using this approach. TrueNorth was not able to estimate time this way.

Results show that the efficiency of our approach is significantly higher than the existing algorithms. Our approach is also the only method to solve the SNN mapping problem of 4 billion neurons and 1 million cores in a reasonable time (26 seconds). While all other methods failed to provide a solution under 100 hours.

Figures 10, 11, and 12 respectively illustrate the performance of the placements obtained by approaches in terms of three metrics: energy consumption, latency, and congestion. In figure 11 and 12, the light and dark parts represent the maximum and average values of the corresponding indexes. The test SNN applications in

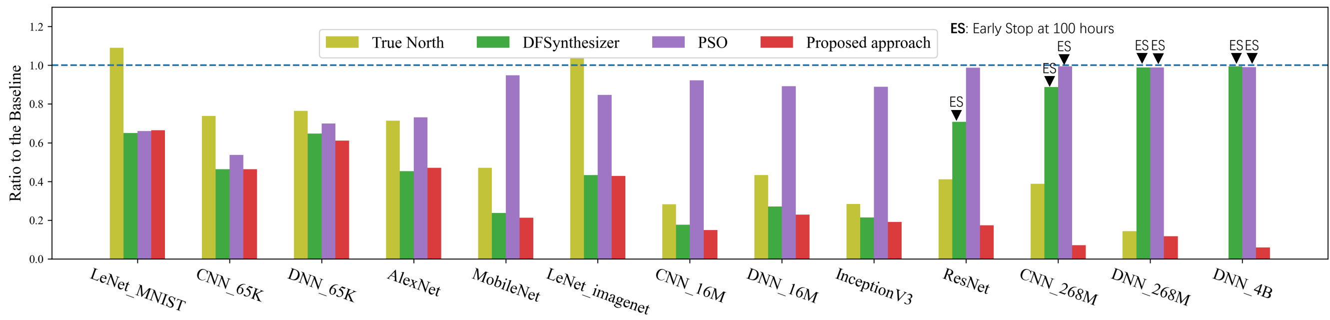


Figure 10: Results on energy consumption

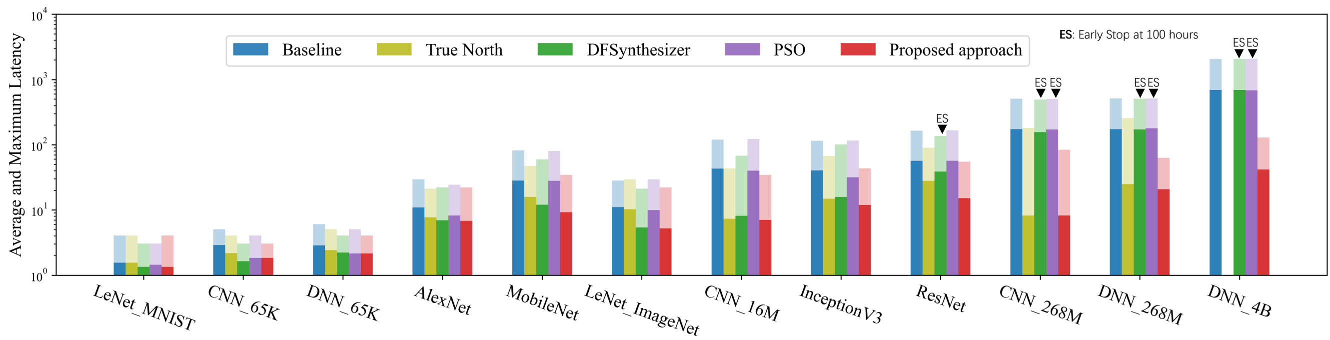


Figure 11: Results on average and maximum latency

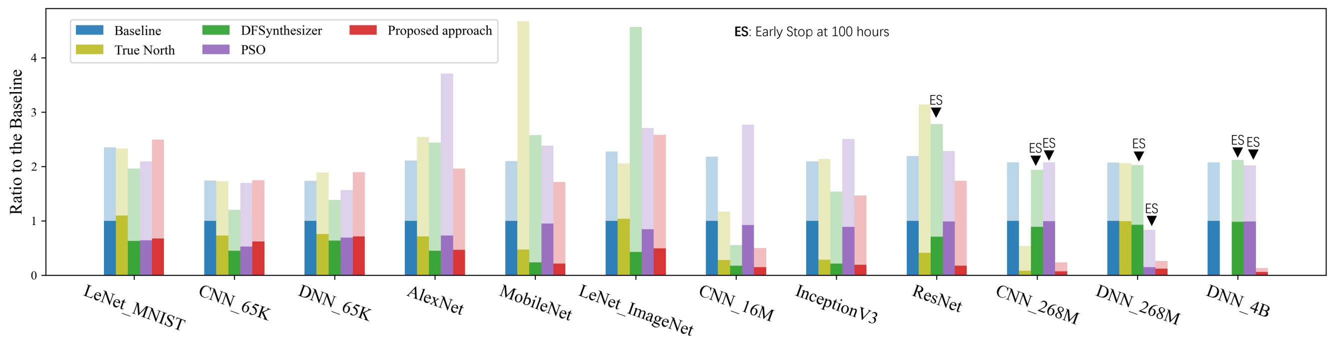


Figure 12: Results on average and maximum congestion

all figures are listed from left to right in order of scale. The Y-axis in figure 11 is logarithmic.

Just like in the case of Figure 9, some existing algorithms cannot complete the solution on some data sets due to the too-slow solution time. Therefore, some test points have no data, and others have incorrect data. For practical reasons, we put an upper bound on the execution time of 100 hours for all methods. If the algorithm exceeds 100 hours, it will be forced to stop and take its current optimization result as its solution to be measured. These particular incorrect data points will be marked "early stop" in the figures. Since both PSO and DFSynthesizer algorithms are iteratively optimized, it makes sense to retire early. While TrueNorth cannot use this technique.

The results show that the quality of the solution obtained by our algorithm is significantly better than the existing algorithms in each metric. In average, compared with the best results of the other three methods, our algorithm reduces energy consumption by 47.8%, average latency by 31.7%, maximum latency by 36.5%, average congestion by 42.9%, and maximum congestion by 36.7%. For the largest test application, the corresponding performance improvement is 93.9%, 92.7%, 93.8%, 93.9%, and 93.8%.

One observation is that when the SNN application scale is small, the solutions of approaches are of similar quality. As the problem size increases, the performance of existing methods sharply decreases compared to the baseline while our method improves. Our analysis is that when the size of the problem is small, the problem's solution space is small, and each method can obtain results close to the optimal solution. When the size of the problem increases, the solution space grows exponentially, and the existing methods are difficult to approach optimal solutions from a random starting point. On the other hand, Our algorithm can start from a relatively good starting point provided by the HSC and benefit from the larger optimization space to achieve more improvement compared to the baseline.

6 CONCLUSION

This paper proposes a novel approach to map SNNs to neuromorphic hardware, motivated by our findings that existing approaches are inadequate for the mapping requirements of new large-scale hardware. The main idea of our approach is to use the Hilbert space-filling curve to obtain an initial placement and generate an optimal overall layout of the data flow for the SNN, and then use the FD algorithm to optimize the placement further. We discuss how the unique properties of the space-filling curve and SNN fit together to obtain a high-quality placement. We present the FD algorithm, which we have made much effort to make efficient when mapping large-scale SNN. Experimental results demonstrate that our approach shows significantly better performance in both solution quality and solving speed compare to existing methods.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China (No. 2022YFB4500100), Natural Science Foundation of China (No. 61925603), Zhejiang Lab (No. 2021KC0AC01, No. 2020KC0AC01) and The Key Research and Development Program of Zhejiang Province in China (No. 2020C03004).

A HSC IN ARBITRARY RECTANGLE

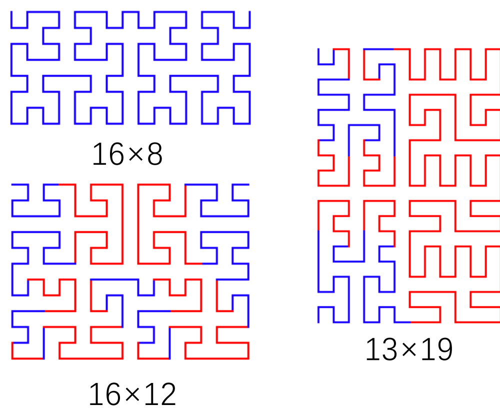


Figure 13: Some instances of modified Hilbert curve in arbitrary rectangle

One issue that arises when mapping sequence using HSC is that discrete Hilbert curve is defined only on squares with sides length being the powers 2, i.e., 2^n, while the hardware system size is usually a rectangle of arbitrary size. To address this issue, we use a modified Hilbert curve based on Rong's work [29] to generate the mapping function. This modification aims to extend the domain of the Hilbert curve into a rectangle of arbitrary size while preserving the locality property as much as possible. Figure 13 shows some instances of the modified Hilbert curve.

B EXPECTATION FUNCTION

Algorithm 4: function $Expe()$

Input: $x, y, (x_s, y_s), (x_t, y_t)$

1. $P_s, P_t = (x_s, y_s), (x_t, y_t)$;
2. **if** (x, y) *is outside the area enclosed by* P_s *and* P_t **then**
3.   return 0;
4. **end**
5. L = [all grid points within the area enclosed by P_s and P_t];
6. sort L by the Manhattan distance from the P_s;
7. Build array $E[]$;
8. **foreach** *points* P_i *within the area enclosed by* P_s *and* P_t **do**
9. $E[P_i] = 0$;
10. **end**
11. E$[P_s] = 1$;
12. **foreach** $P_{now} = (x_{now}, y_{now})$ *in* L **do**
13.   **if** $x_{now} == x_t$ **then**
14. $E[(x_{now}, y_{now} + 1)] += E[P_{now}]$;
15.   **end**
16.   **else if** $y_{now} == y_t$ **then**
17. $E[(x_{now} + 1, y_{now})] += E[P_{now}]$;
18.   **end**
19.   **else**
20. $E[(x_{now}, y_{now} + 1)] += E[P_{now}]/2$; $E[(x_{now} + 1, y_{now})] += E[P_{now}]/2$;
21.   **end**
22. **end**
23. return $E[(x, y)]$

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[2] Adarsha Balaji, Prathyusha Adiraju, Hirak J Kashyap, Anup Das, Jeffrey L Krichmar, Nikil D Dutt, and Francky Catthoor. 2020. PyCARL: A PyNN interface for hardware-software co-simulation of spiking neural network. *arXiv preprint arXiv:2003.09696* (2020).

[3] Adarsha Balaji and Anup Das. 2019. A framework for the analysis of throughput-constraints of SNNs on neuromorphic hardware. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 193–196.

[4] Adarsha Balaji, Anup Das, Yuefeng Wu, Khanh Huynh, Francesco G. Dell'Anna, Giacomo Indiveri, Jeffrey L. Krichmar, Nikil D. Dutt, Siebren Schaafsma, and Francky Catthoor. 2020. Mapping Spiking Neural Networks to Neuromorphic Hardware. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 1 (2020), 76–86. https://doi.org/10.1109/tvlsi.2019.2951493

[5] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V

Arthur, Paul A Merolla, and Kwabena Boahen. 2014. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proc. IEEE* 102, 5 (2014), 699–716.

[6] Anup Das, Yuefeng Wu, Khanh Huynh, Francesco Dell'Anna, Francky Catthoor, and Siebren Schaafsma. 2018. Mapping of local and global synapses on spiking neuromorphic hardware. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1217–1222.

[7] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. 2018. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 1 (2018), 82–99.

[8] Michael V DeBole, Brian Taba, Arnon Amir, Filipp Akopyan, Alexander Andreopoulos, William P Risk, Jeff Kusnitz, Carlos Ortega Otero, Tapan K Nayak, Rathinakumar Appuswamy, et al. 2019. TrueNorth: Accelerating from zero to 64 million neurons in 10 years. *Computer* 52, 5 (2019), 20–29.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. IEEE, 248–255.

[10] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[11] Shuiguang Deng, Pan Lv, Ouwen Jin, Schahram Dustdar, Ying Li, De Ma, Zhaohui Wu, and Gang Pan. 2022. Darwin-S: A Reference Software Architecture for Brain-Inspired Computers. *Computer* 55, 5 (2022), 51–63. https://doi.org/10.1109/MC.2022.3144397

[12] Steve B Furber, Francesco Galluppi, Steve Temple, and Luis A Plana. 2014. The spinnaker project. *Proc. IEEE* 102, 5 (2014), 652–665.

[13] Francesco Galluppi, Sergio Davies, Alexander Rast, Thomas Sharp, Luis A Plana, and Steve Furber. 2012. A hierachical configuration system for a massively parallel neural hardware platform. In *Proceedings of the 9th conference on Computing Frontiers*. 183–192.

[14] Michael R Gary and David S Johnson. 1979. Computers and Intractability: A Guide to the Theory of NP-completeness.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[16] David Hilbert. 1935. Dritter Band: Analysis · Grundlagen der Mathematik · Physik Verschiedenes, Nebst Einer Lebensgeschichte. (1935), 1–2. https://doi.org/10.1007/978-3-662-38452-7_1

[17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[18] Yangfan Hu, Huajin Tang, and Gang Pan. 2021. Spiking Deep Residual Networks. *IEEE Transactions on Neural Networks and Learning Systems* (2021), 1–6. https://doi.org/10.1109/TNNLS.2021.3119238

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

[20] Yann LeCun et al. 2015. LeNet-5, convolutional neural networks. *URL: http://yann.lecun. com/exdb/lenet* 20, 5 (2015), 14.

[21] Chenchen Liu, Bonan Yan, Chaofei Yang, Linghao Song, Zheng Li, Beiye Liu, Yiran Chen, Hai Li, Qing Wu, and Hao Jiang. 2015. A spiking neuromorphic design with resistive crossbar. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[22] Qianhui Liu, Gang Pan, Haibo Ruan, Dong Xing, Qi Xu, and Huajin Tang. 2020. Unsupervised AER Object Recognition Based on Multiscale Spatio-Temporal Features and Spiking Neurons. *IEEE Transactions on Neural Networks and Learning Systems* 31, 12 (2020), 5300–5311. https://doi.org/10.1109/TNNLS.2020.2966058

[23] Xiaoxiao Liu, Wei Wen, Xuehai Qian, Hai Li, and Yiran Chen. 2018. Neu-NoC: A high-efficient interconnection network for accelerated neuromorphic systems. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 141–146.

[24] De Ma, Juncheng Shen, Zonghua Gu, Ming Zhang, Xiaolei Zhu, Xiaoqiang Xu, Qi Xu, Yangjing Shen, and Gang Pan. 2017. Darwin: A neuromorphic hardware co-processor based on spiking neural networks. *Journal of Systems Architecture* 77 (2017), 43–51.

[25] Wolfgang Maass. 1997. Networks of spiking neurons: The third generation of neural network models. *Neural Networks* 10, 9 (1997), 1659–1671. https://doi.org/10.1016/s0893-6080(97)00011-7

[26] Christian Mayr, Sebastian Hoeppner, and Steve Furber. 2019. Spinnaker 2: A 10 million core processor system for brain simulation and machine learning. *arXiv preprint arXiv:1911.02385* (2019).

[27] Carver Mead. 1990. Neuromorphic electronic systems. *Proc. IEEE* 78, 10 (1990), 1629–1636.

[28] Saber Moradi, Ning Qiao, Fabio Stefanini, and Giacomo Indiveri. 2017. A Scalable Multicore Architecture With Heterogeneous Memory Structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs). *IEEE Transactions on Biomedical Circuits and Systems* 12, 1 (11 2017), 106–122. https://doi.org/10.1109/TBCAS.2017.2759700

[29] Yibiao Rong, Xia Zhang, and Jianyu Lin. 2021. Modified Hilbert Curve for Rectangles and Cuboids and Its Application in Entropy Coding for Image and Video Compression. *Entropy* 23, 7 (2021), 836. https://doi.org/10.3390/e23070836

[30] Kaushik Roy, Akhilesh Jaiswal, and Priyadarshini Panda. 2019. Towards spike-based machine intelligence with neuromorphic computing. *Nature* 575, 7784 (2019), 607–617. https://doi.org/10.1038/s41586-019-1677-2

[31] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. 2017. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in neuroscience* 11 (2017), 682.

[32] Pradip Kumar Sahu and Santanu Chattopadhyay. 2013. A survey on application mapping strategies for Network-on-Chip design. *Journal of Systems Architecture* 59, 1 (2013), 60–76. https://doi.org/10.1016/j.sysarc.2012.10.004

[33] Jun Sawada, Filipp Akopyan, Andrew S Cassidy, Brian Taba, Michael V Debole, Pallab Datta, Rodrigo Alvarez-Icaza, Arnon Amir, John V Arthur, Alexander Andreopoulos, et al. 2016. Truenorth ecosystem for brain-inspired computing: scalable systems, software, and applications. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 130–141.

[34] Johannes Schemmel. 2021. The BrainScaleS accelerated analogue neuromorphic architecture. *Brain-inspired Computing* (2021), 14.

[35] Luping Shi, Jing Pei, Ning Deng, Dong Wang, Lei Deng, Yu Wang, Youhui Zhang, Feng Chen, Mingguo Zhao, Sen Song, et al. 2015. Development of a neuromorphic computing system. In *2015 IEEE international electron devices meeting (IEDM)*. IEEE, 4–3.

[36] Shihao Song, Harry Chong, Adarsha Balaji, Anup Das, James Shackleford, and Nagarajan Kandasamy. 2022. DFSynthesizer: Dataflow-based synthesis of spiking neural networks to neuromorphic hardware. *ACM Transactions on Embedded Computing Systems (TECS)* 21, 3 (2022), 1–35.

[37] Shihao Song, M Lakshmi Varshika, Anup Das, and Nagarajan Kandasamy. 2021. A design flow for mapping spiking neural networks to many-core neuromorphic hardware. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[38] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2015. Rethinking the Inception Architecture for Computer Vision. *CoRR* abs/1512.00567 (2015). arXiv:1512.00567 http://arxiv.org/abs/1512.00567

[39] Giulio Tononi, Olaf Sporns, and Gerald M Edelman. 1994. A measure for brain complexity: relating functional segregation and integration in the nervous system. *Proceedings of the National Academy of Sciences* 91, 11 (1994), 5033–5037.

Received 2022-10-20; accepted 2023-01-19