A Design Flow for Mapping Spiking Neural Networks to Many-Core Neuromorphic Hardware

Shihao Song, M. Lakshmi Varshika, Anup Das, and Nagarajan Kandasamy
Electrical and Computer Engineering, Drexel University
Philadelphia, PA, USA
Email: {shihao.song,lm3486,anup.das,nk78}@drexel.edu

***Abstract*—The design of many-core neuromorphic hardware is becoming increasingly complex as these systems are now expected to execute large machine-learning models. A predictable design flow is needed to guarantee real-time performance such as latency and throughput without significantly increasing the buffer requirement of computing cores. Synchronous Data Flow Graphs (SDFGs) have been previously used for predictable mapping of streaming applications to multiprocessor systems. We propose an SDFG-based design flow to map spiking neural networks (SNNs) to many-core neuromorphic hardware with the objective of exploring the tradeoff between throughput and buffer-size requirements. The proposed design flow integrates an iterative partitioning approach based on Kernighan–Lin graph partitioning heuristic to create SNN clusters such that each cluster can be mapped to a core of the hardware. The partitioning approach minimizes inter-cluster spike communication, which improves latency on the shared interconnect of the hardware. Next, the design flow maps clusters to cores using Particle Swarm Optimization (PSO), an evolutionary algorithm, while exploring the design space of throughput and buffer size. Pareto-optimal mappings are retained from the design flow, allowing system designers to select a Pareto mapping that satisfies throughput and buffer-size requirements of the design. We evaluated the developed design flow using five large-scale convolutional neural network (CNN) models. Results demonstrate 63% higher maximum throughput and 10% lower buffer-size requirement compared to state-of-the-art dataflow-based mapping solutions.**

***Index Terms*—neuromorphic computing, spiking neural network (SNN), design-space exploration (DSE), oxide-based resistive random access memory (OxRRAM), dataflow**

I. Introduction

Neuromorphic computing systems are integrated circuits that implement the architecture of a central nervous system of primates [1]–[3]. These systems enable energy-efficient execution of Spiking Neural Networks (SNNs) [4] due to their event-driven execution, low-power design, and distributed in-place neural computing and synaptic storage architecture. Therefore, neuromorphic systems are suitable for implementing machine-learning inference tasks on embedded systems and edge devices of the Internet-of-Things.

A neuromorphic hardware is implemented as a many-core architecture, where a core is a processing element (PE) consisting of neuron circuitry and memory cells [5]. A common design practice is to build a PE as an analog crossbar [6], where memory cells are organized in a two-dimensional grid with horizontal wordlines and vertical bitlines connecting the neuron circuitry as illustrated in Figure 2b.

A crossbar can accommodate only a fixed number of pre-synaptic connections per post-synaptic neuron. Its dimension is typically constrained to reduce energy consumption and mitigate the negative impact of technology scaling. Therefore, neuromorphic system-software frameworks partition SNNs into smaller clusters such that each cluster can be mapped directly on to the crossbar of a neuromorphic PE [7]. We show that existing frameworks are either not scalable to large problem sizes or their exploration strategies do not encompass large portions of the hardware mapping design space, leaving behind a significant opportunity to improve performance.

Typically, inference hardware platforms are expected to perform streaming machine-learning tasks, i.e., to perform machine-learning inference continuously on streaming data collected from different sensors. A key performance metric for such tasks is throughput, defined as the inverse of the time it takes to perform an inference (i.e., the time between when an input is presented and when an outcome is returned by the hardware). A neuromorphic computing inference hardware enables parallel execution and pipelining of operations. Therefore, scheduling operations of an inference task onto this pipelined parallel computing environment is a grand challenge. Additionally, once a machine-learning task is partitioned into clusters, cyclic dependency may exist between these clusters, which can lead to performance degradation or in the worst-case, execution deadlock.

In this paper, we propose a design flow that maps SNN-based machine-learning applications onto the PEs of a many-core neuromorphic hardware to ensure predictable timing behavior. We make the following four key **contributions**.

- We propose an iterative approach to partition an SNN into smaller clusters such that each cluster can be implemented on a PE. Our approach integrates the Kernighan–Lin graph partitioning heuristic to find a set of minimum cuts of the directed graph representation of an SNN, minimizing the data (spike) communication between clusters (see Section V).
- We exploit the rich semantics and expressiveness of Synchronous Data Flow Graphs (SDFGs) to represent SNNs, allowing us to analyze key performance properties such as throughput and buffer space, thereby incorporating hardware resource constraints (See Section IV).
- We propose a framework to analyze consistency and deadlock when mapping machine-learning clusters to

hardware. This allows estimation of throughput degradation when 1) the buffer size within each PE is limited and 2) when PEs must be time-multiplexed between different clusters (see Section VI).
- We present a design flow to map SNN-based machine-learning applications on to state-of-the-art many-core neuromorphic computing systems using an instance of Particle Swarm Optimization (PSO). Solutions of the PSO heuristic explore the design space of performance and buffer-size requirements (see Section III).

We evaluate our design flow for a recent neuromorphic hardware using convolutional neural network (CNN)-based machine learning applications. Results show the scalability of our solution and a significant improvement in throughput.

II. Background and Related Works

Spiking Neural Networks (SNNs) enable powerful computations due to their spatio-temporal information encoding capabilities [4]. Figure 1a shows the operation of a leaky integrate-and-fire (LIF) post-synaptic neuron with N pre-synaptic connections. The neuron is described by the state variable $v(t)$ which represents the membrane potential of the neuron. Figure 1b shows a simple implementation of the neuron using membrane resistance R_m and capacitance C_m.

Figure 1c shows the state diagram of the neuron. The dynamics of the neuron is described by [8] as

$$C_m \frac{dv(t)}{dt} = I_{\text{leak}}(t) + I_s(t) + I_{\text{inj}}(t), \tag{1}$$

where $I_{\text{leak}}(t) = -\frac{C_m}{\tau_m}[v(t) - v_{\text{rest}}]$ is the leakage current in the membrane, $\tau_m = C_m R_m$ is the time constant of the membrane, v_{rest} is the resting potential, $I_s(t)$ is the current due to the synaptic input to the neuron, and $I_{\text{inj}}(t)$ is the current injected into the neuron by an intercellular electrode.

We consider current-based (CUBA) synapses, where the synaptic current of the post-synaptic neuron is given by

$$I_s(t) = \sum_{i=1}^{N} S_i W_i, \tag{2}$$

where $S_i = \sum_{\tau_k} \delta(t - \tau_k)$ is the spike train of i^{th} pre-synaptic neuron and w_i is the synaptic strength of the connection of this neuron to the post-synaptic neuron.

In the firing state, the post-synaptic neuron fires a spike when its membrane voltage $v(t)$ crosses the threshold voltage V_{th}. The output spiking current is defined as

$$I_{\text{spike}}(t) = C_m \left[\frac{dv(t)}{dt}\right]^{-1}_{v=V_{\text{th}}} (V_{\text{rest}} - V_{\text{th}})\delta(v(t) - V_{\text{th}}) \tag{3}$$

SNNs can implement many machine-learning approaches. For a supervised machine-learning application, an SNN is trained with representative data, where training refers to adjusting the synaptic weight of connections between pre- and post-synaptic neurons of the SNN [9]. Machine-learning **inference** refers to feeding live data points to a trained SNN and generating the corresponding output.

Neuromorphic hardware platforms are used to implement SNN-based machine learning applications [1]. Table I shows some recently demonstrated neuromorphic hardware platforms along with their capacity in terms of number of neurons and synapses. These platforms are implemented as a many-core hardware [5] (see Figure 2a), where the cores are interconnected via a shared interconnect such as Network-on-Chip [10] and Segmented Bus [11]. A neuromorphic core consists of a PE, which implements the neuron circuitry and synaptic cells. A common design practice is to build a PE as an analog crossbar [6] (see Figure 2b). In a crossbar, pre-synaptic neuron circuitry acts as current drivers and are placed along each wordline, while post-synaptic neuron circuitry acts as current sinks and are placed along each bitline. Memory cells are placed at the crosspoint of a wordline and bitline, and they store the synaptic weights of an SNN.

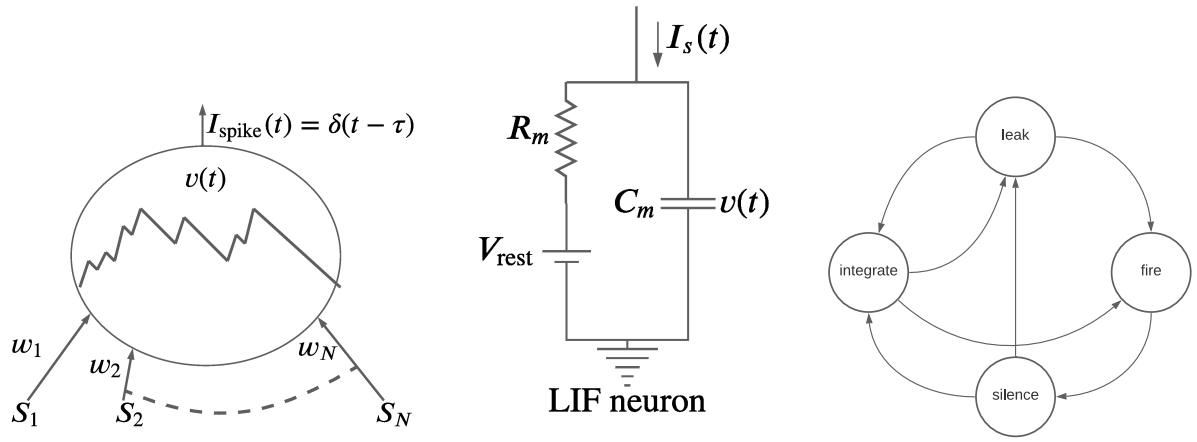
TABLE I
Capacity of recent neuromorphic hardware platforms.

	ODIN [12]	**μBrain** [13]	**DYNAPs** [14]	**BrainScaleS** [15]	**SpiNNaker** [16]	**Neurogrid** [17]	**Loihi** [18]	**TrueNorth** [19]
# Neurons/core	256	336	256	512	36K	65K	130K	1M
# Synapses/core	64K	38K	16K	128K	2.8M	8M	130M	256M
# Cores/chip	1	1	1	1	144	128	128	4096
# Chips/board	1	1	4	352	56	16	768	4096
	High-performance neuromorphic system							
# Neurons	256	336	1K	4M	2.5B	1M	100M	4B
# Synapses	256	336	65K	1B	200B	16B	100B	1T

A neuromorphic hardware enables distributed and pipelined processing of SNN operations. Additionally, each crossbar in the hardware can implement a maximum of N pre-synaptic neurons per post-synaptic neuron. Therefore, system-software frameworks such as NEUTRAMS [20], NeuroXplorer [21], Corelet [22], and PACMAN [23] consist of 1) a compiler, which partitions an SNN model into clusters such that the neurons and synapses of each cluster can be mapped to a crossbar of the hardware, and 2) a run-time manager, which maps the clusters of an SNN to the cores of a many-core hardware. To this end, several mapping strategies have been proposed, including optimizing for energy [7], [24]–[26], throughput [27]–[30], resource utilization [20], [31]–[33], circuit aging [34]–[38], inference lifetime [39], and endurance [40]–[42]. These mapping techniques all use some variant of the SNN-partitioning approach proposed in SpiNeMap [24].

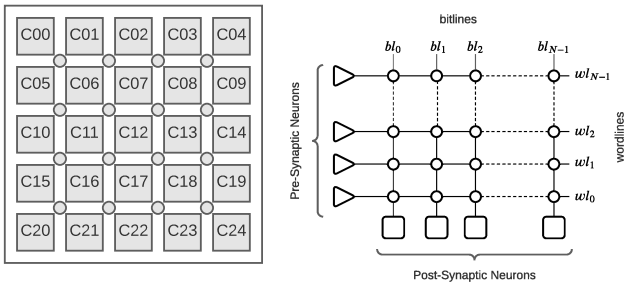
Recently, dataflow models have been used to analyze performance of SNNs implemented on neuromorphic hardware. There are two strategies proposed in literature – the **SDFSNN** [27] and its extended version [28], which uses dataflow graphs to model an SNN, performing partitioning and mapping explorations with neurons and synapses directly, and the **DFSynthesizer** [29] and its extended version [30], which uses dataflow graphs to only model the clustered SNN, allowing mapping and scheduling of the clusters (a collection of neurons and synapses) to the PEs of a neuromorphic hardware.

We show that SDFSNN is not scalable to large SNN models. DFSynthesizer, on the other hand, uses clusters of an SNN-model as its input, and is therefore scalable to large problem sizes. However, we show that DFSynthesizer is not able to explore a significant portion of the design space. Figure 3



(a) An LIF neuron with N pre-synaptic connections. (b) Implementation of an LIF neuron. (c) State diagram of an LIF neuron.

Fig. 1. Implementation and operation of an LIF neuron.



(a) Neuromorphic hardware. (b) Analog crossbar-based PE.

Fig. 2. Distributed computing architecture in neuromorphic hardware.

shows at a high-level, how the proposed design flow differs from these existing works. The proposed flow uses an iterative approach involving graph partitioning into clusters followed by mapping these clusters to hardware. We describe this flow in greater detail in Section III.

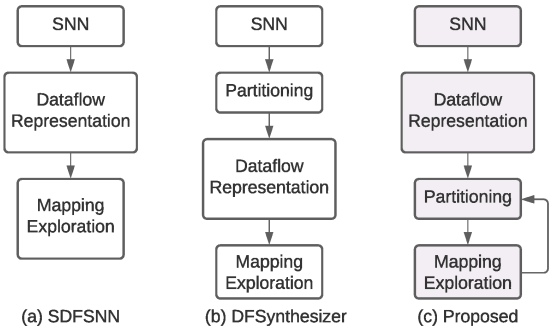


Fig. 3. Comparing the proposed approach with SDFSNN [27], [28] as well as DFSynthesizer [29], [30].

III. Proposed Design Flow

Figure 4 shows the five steps of our design flow. These steps are enumerated below.

1) An SNN model is represented using a dataflow graph.
2) The SNN graph is partitioned into clusters using an iterative solution.
3) Clusters and their connections are analyzed for consistency and freedom from deadlock.
4) The sub-graph representing each cluster is substituted as nodes into the original dataflow graph to generate a dataflow representation of the clustered SNN.
5) The clustered SNN graph is mapped to the hardware, where mapping involves allocating a cluster to a core of the hardware.
6) If multiple clusters are mapped to a core, a list scheduler is used to schedule (order) the execution of these clusters on the core.
7) A decision is made on the buffer size on each channel. To do so, we make a trade-off between buffer size and throughput of the application.
8) The design flow explores a new partition and repeats the above-described exploration steps.

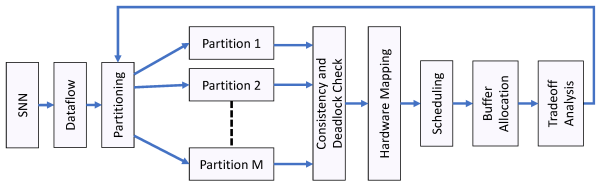


Fig. 4. Steps of the proposed design flow.

IV. Dataflow Representation of SNNs

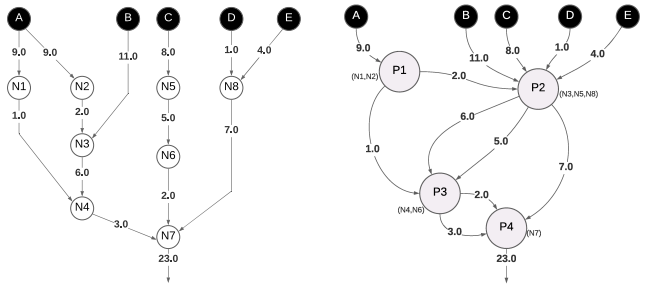
We model an SNN as a Synchronous Data Flow Graph (SDFG) for predictable performance analysis [43]. SDFGs are commonly used to model streaming applications that are implemented on a multi-core system [44]–[46]. Both pipelined streaming and cyclic dependencies between tasks can be easily modeled in SDFGs. These graphs are used to analyze a system in terms of key performance properties such as throughput, execution time, communication bandwidth, and

buffer requirement [47]–[49]. Nodes of an SDFG are called *actors*, which are computed by reading *tokens* from their input ports and writing the results of the computation as tokens on output ports. Port rates are visualized as annotations on edges. Actor execution is also called *firing*, and it requires a fixed amount of time to execute. Edges in the graph are called *channels* and they represent dependencies among actors. An actor is said to be *ready* when it has sufficient input tokens on all its input channels and sufficient buffer space on all its output channels; an actor can only fire when it is ready.

One important property of an SDFG is *throughput*, which is defined as the inverse of its long-term period. A period is the average time needed for one iteration of the SDFG. An iteration is defined as the minimum non-zero execution such that the original state of the SDFG is obtained. This is the performance parameter used in this paper.

To model a trained SNN as an SDFG, we consider the average number of spikes per frame on each synaptic connection of the SNN. For image-based applications, which are the primary focus of this work, a frame corresponds to an individual image. For time-series applications such as natural language and biosignal processing, a frame corresponds to the data collected within a fixed-length timing window. Spike count on synapses of an SNN can be obtained by simulating the trained SNN in a simulator such as Brian [50] and PyCARL [51] using representative training data. Figure 5a shows an example SNN with 8 neurons (N1-N8) connected to 5 inputs (A-E). Formally,

Definition 1: (SNN GRAPH) *An SNN* $\mathbf{G_{SNN}} = (\mathbf{N}, \mathbf{S})$ *is a directed graph consisting of a finite set* $\mathbf{N}$ *of nodes, representing neurons and a finite set* $\mathbf{S}$ *of edges, representing synapses.*



(a) SDFG representation of an SNN. (b) Partitioned SDFG.

Fig. 5. Modeling SNN as a Synchronous Dataflow Graph (SDFG).

Table II shows the one-to-one mapping of an SNN to SDFG properties. In representing an SNN as an SDFG, we discard the inter-spike interval on synapses, retaining only the spike count. For instance, the neuron N3 (in Fig. 5a) in our model fires 6 spikes (tokens) at once when it receives 2 spikes from N2 and 11 spikes from input B. In practice, however, the 6 output spikes from N3 are generated and transmitted at different times.

Table III reports the average input/output degree and the maximum diameter of the SDFG obtained from the five evaluated machine learning applications (see Section VIII).

TABLE II
ONE -TO-ONE MAPPING OF SNN TO SDFG TERMINOLOGY.

SDFG Terminology	SNN Terminology
actor	neuron
channel	synapse
token	spike

TABLE III
PROPERTIES OF SDFG REPRESENTATION OF EVALUATED SNN.

Application	In Degree		Out Degree		Diameter
	Max	Average	Max	Average	
LeNet	144	73	144	73	4
AlexNet	102	119	204	119	7
ResNet	288	133	576	134	8
DenseNet	288	104	576	104	10
VGG	288	89	576	89	11

V. ITERATIVE SNN PARTITIONING

Each core in a neuromorphic hardware can accommodate only a certain number of pre- and post-synaptic neurons. So, a single core may not be sufficient to map all neurons and synapses of an SNN. In such scenarios where more than one cores are needed, an SNN needs to be partitioned into clusters, where each cluster consists of a subset of neurons and synapses of the original SNN. The partitioning step ensures that a cluster can fit onto a core of the many-core neuromorphic hardware. Spike communication constitutes a significant fraction of the total energy consumption in a neuromorphic hardware [25]. Therefore, SNN partitioning algorithms minimize the spike communication between clusters. To this end, we propose a novel iterative approach to partitioning an SNN into clusters. Our approach is tightly integrated with cluster mapping explorations to generate better throughput-buffer size trade-off.

Graph partitioning is an NP-hard problem and has been studied extensively in the context of workload distribution for the efficient use of a distributed memory parallel computer. Several heuristic solutions have been proposed to solve this problem with the objective of minimizing the communication cost between computers and balancing the workload on each computer. A thorough review of these methods and the extensive literature associated with them is beyond the scope of this paper. We chose Kernighan-Lin (KL) recursive graph partitioning approach [52]. In the following, we describe how the KL approach is tuned for neuromorphic computing and is integrated inside the proposed iterative solution.

To formulate our partitioning problem, we consider the example of an $M \times M$ analog crossbar, which can accommodate a maximum of M pre-synaptic and M post-synaptic neurons. We represent a partition of the SNN using the binary mapping matrix $\mathbb{P} \in \mathbb{R}^{|N| \times |C|}$, where

$$p_{i,j} = \begin{cases} 1 & \text{if neuron } n_i \text{ is mapped to cluster } c_j \\ 0 & \text{otherwise} \end{cases}$$

Algorithm 1 illustrates the pseudo-code for the proposed iterative SNN partitioning. The algorithm runs for η iterations, which is a user-defined parameter that controls the design space exploration for throughput-buffer size trade-off. First,

the algorithm partitions an SNN graph G_{SNN} by randomly allocating neurons to different clusters (line 2). The total communication cost (measured as the total number of spikes communicated between clusters) is evaluated (line 3). By minimizing the total communication cost as formulated, the partitioning algorithm minimizes 1) communication energy, thereby lowering the total energy consumption and 2) congestion on the shared interconnect, thereby reducing the spike latency. Starting from this initial partitioning, the KL approach recursively swaps neurons between clusters, such that the cost is minimized (lines 4-18). To this end, a variable δ is used to track the reduction of the cost function. δ is initialized to a very large number (line 4). The algorithm iterates through lines 6 to 17 as long as the improvement in cost is greater than a user-defined minimum δ_{min}. At each iteration, the algorithm performs the following. For each neuron pair (line 6), clusters to which these neurons are mapped are obtained from the partition matrix (lines 7-8). If the two clusters are different (line 9), the partition is changed by swapping the two neurons (line 10). If this new change is valid (i.e., both the clusters satisfy the hardware constraint), then the new cost is evaluated (lines 11-12). If the cost is lower than the initial cost (line 13), the neuron swap is made permanent and the reduction in the cost function is evaluated (lines 14-16). The KL partitioning terminates by generating a clustered SNN graph G_{CSNN} from the partition matrix by replacing the subgraph of each cluster as a node (line 19). Figure 5b shows the clusters generated from the original SNN of Figure 5a. A clustered SNN graph is formally defined as

Algorithm 1: Partitioning SDFG graph G_{SNN}.

```
   Input: G_SNN = (N, S)
1  for r = 0; r < η; r++ do                    /* Run for η iterations */
2      P^init = InitPartition();               /* Initial partition */
3      Evaluate Cost^init;                     /* Evaluate comm. cost of this
         initial partition */
       ; /* KL Partitioning begins here                                  */
4      δ = ∞ ;                                 /* Set a large value to δ */
5      while δ > δ_min do                      /* Repeat until the improvement in
         cost is not significant */
6          for n_i, n_j ∈ N do                 /* For each pair of nodes in the
             SDFG G_SNN */
7              k = argmax P^init(i, :) ;       /* Find the cluster
                 where neuron n_i is mapped */
8              l = argmax P^init(j, :) ;       /* Find the cluster
                 where neuron n_j is mapped */
9              if k ≠ l then                   /* If the cluster of n_i and n_j
                 are different */
10                 P^new = P^init | p_{i,k} = 0, p_{j,l} = 0, p_{i,l} = 1, p_{j,k} =
                     1 ;                       /* Swap the neurons n_i and n_j */
11                 if Σ_u P^new(u, k) ≤ M and Σ_v P^new(v, l) ≤ M
                     then                      /* If the neuron swap is valid */
12                     Evaluate Cost^new ;     /* Evaluate comm.
                         cost of this new partition */
13                     if cost^new < cost^init then /* If the cost
                         reduces */
14                         P^init = P^new;     /* Retain the swap
                             */
15                         δ = cost^init − cost^new;   /* Retain
                             the improvement in cost */
16                         cost^init = cost^new; /* Set new cost
                             */
17         end
18     end
19     Generate G_CSNN = (C, E) using P^init ;  /* Generate the
         clustered graph from the mapping */
       ; /* KL Partitioning ends here                                    */
20     Mapping(G_CSNN);                        /* Perform hardware mapping */
21 end
```

Definition 2: (CLUSTERED SNN GRAPH) *A clustered SNN graph* $\mathbf{G_{CSNN}} = (\mathbf{C}, \mathbf{E})$ *is a directed graph consisting of a finite set* $\mathbf{C}$ *of clusters and a finite set* $\mathbf{E}$ *of edges between these clusters.*

The partitioning algorithm uses the clustered SNN graph to perform hardware mapping (line 20) for throughput-buffer size trade-off (line 21). Finally, the algorithm is repeated to explore a new design space, starting from another initial partitioning (line 2). We next describe this hardware mapping.

VI. HARDWARE MAPPING EXPLORATIONS

In order to perform the hardware mapping exploration of a clustered SNN graph, we represent a many-core neuromorphic hardware using the hardware graph defined as

Definition 3: (NEUROMORPHIC HARDWARE GRAPH) *A neuromorphic hardware graph* $\mathbf{H} = (\mathbf{T}, \mathbf{L})$ *is a directed graph consisting of a finite set* $\mathbf{T}$ *of cores and a finite set* $\mathbf{L}$ *of links between these cores.*

Definition 4: (CORE AND LINK) *A core* $\mathbf{t}_i$ *is a tuple* $\langle I_i, O_i, \tau_i, inC(i), outC(i), inB(i), outB(i)\rangle$ *consisting of a set* I_i $(\subseteq Ports)$ *of input ports, a set* O_i $(\subseteq Ports)$ *of output ports with* $I_i \cap O_i = \emptyset$, τ_i *is the execution time of* $\mathbf{t}_i$, $(inC(i), outC(i))$ *is the maximum number of incoming and outgoing connections supported by* $\mathbf{t}_i$, *and* $(inC(i), outC(i))$ *is its maximum incoming and outgoing bandwidth. Each link* $l_{i,j} \in \mathbf{L}$ *connecting cores* $\mathbf{t}_i$ *and* $\mathbf{t}_j$ *is associated with a latency* $t_{i,j}$, *which is the time it takes to communicate a spike packet on this link.*

Figure 6 shows our design-space exploration framework for mapping an SNN to a many-core neuromorphic hardware. The flow starts with refining resource requirements of a clustered SNN graph. An application graph specifies only the resource requirement of its clusters. Estimating resource requirements of its edges (i.e., buffer size and bandwidth) is performed in this first step of the flow. In the next step, the flow maps each cluster to a core. For this mapping, a static-order schedule is constructed for each core that maps more than one clusters. Next, the throughput is computed and the exploration is continued starting with a different cluster-to-core mapping. Finally, the flow iterates back to step 1 and increases the buffer size assigned to edges in order to explore a new design space.

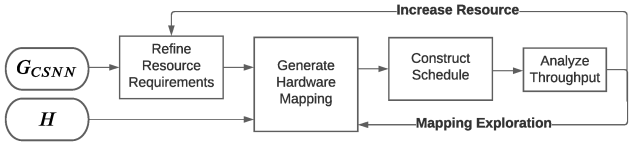


Fig. 6. Design space exploration to perform throughput-buffer size trade-off.

Figure 7 illustrates the selection of Pareto points using our design space exploration approach. There are 9 Pareto points (A-I) obtained from 6 design spaces (big circles), which

correspond to 6 distinct partitionings of an SNN. The design space using DFSynthesizer is shown in the figure with a different color circle. We observe that Pareto points C, D, and E are common to both DFSynthesizer and the proposed approach. However, Pareto points X and Y of DFSynthesizer are discarded in favor of better solutions (Pareto points F, G, and H) obtained using the proposed approach. We conclude that the proposed approach can generate better trade-off between throughput and buffer size.

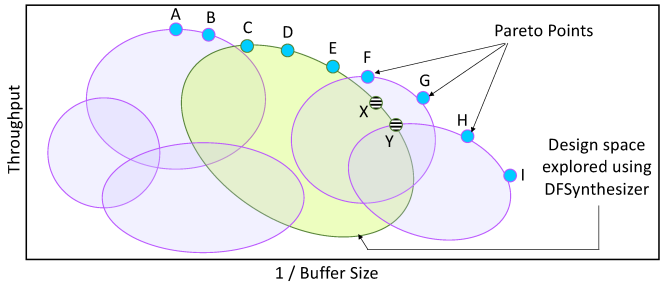


Fig. 7. Demonstration of design space exploration of throughput and buffer size, and the selection of Pareto points.

A. Refining Resource Requirements

Spikes that are communicated on edges of a clustered SNN graph must be stored in a buffer. The amount of buffer that is allocated to these edges has a large impact on the achieved throughput of an application. Allocating more buffer to an edge might increase the throughput because it may increase pipelining opportunities. Typically, buffer size is chosen such that the throughput requirement is met [53]–[55]. However, the throughput requirement is not known beforehand. Therefore, a trade-off must be made between the realizable throughput and the buffer allocated to the edges of the clustered SNN graph.

We use the SDF^3 tool [56] to perform this throughput-buffer size trade-offs, i.e., generating different *Pareto points*. SDF^3 uses a fast technique involving the construction of abstract dependency graph from the clustered SNN graph to estimate the maximum throughput for a given buffer size by considering its mapping to a single-core neuromorphic hardware [57]. This simplifies the analysis in the absence of hardware mapping information, which is obtained in the subsequent steps. However, to make the analysis relevant for multi-core neuromorphic hardware, we consider separate buffer on each edge, with the total buffer size obtained by adding the buffer sizes allocated to different edges [57].

Figure 8 reports the Pareto points for the five evaluated applications. We observe that throughput of these applications increases with an increase in the allocated buffer size. This is because with more buffers on edges, clusters can be executed earlier whenever tokens are ready, which increases throughput.

B. Generating Hardware Mapping

For each Pareto point, a hardware mapping exploration is performed, where mapping involves placing each cluster of the clustered SNN graph on to a core of the hardware. To this

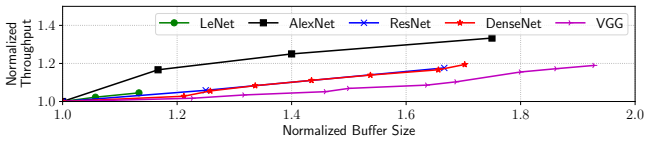


Fig. 8. Throughput buffer size tradeoffs.

end, we use an instance of the Particle Swarm Optimization (PSO) [58], a meta-heuristic algorithm used to search for the optimum solution of an optimization problem. The mapping problem is indicated using the matrix $\mathcal{M} \in \mathbb{R}^{|C|\times|T|}$, where $m_{i,j} = \begin{cases} 1 & \text{if cluster } c_i \text{ is mapped to core } t_j \\ 0 & \text{otherwise} \end{cases}$.

The mapping constraint is the following:

- A cluster can be mapped to only one crossbar, i.e.,

$$\sum_j m_{ij} = 1 \quad \forall i \tag{4}$$

The optimization problem is to maximize the throughput of an SNN represented as an SDFG. For computing throughput, we use the SDF^3 tool, which estimates throughput of an SDFG based on its self-timed execution [59]. To do so, we integrate the tool inside our PSO formulation, allowing to estimate the throughput for a given allocation of clusters to cores. Therefore, the fitness function is represented as $F = \text{SDF}^3(\mathcal{M})$.

For PSO, we instantiate n_p swarm particles. The position of these particles are solutions to the fitness function, and they represent different cluster-to-core mappings. Each particle also has a velocity with which it moves in the search space to find the optimum solution. During the movement, a particle updates its position and velocity according to its own experience (closeness to the optimum) and also experience of its neighbors. We introduce the following notations.

$$\begin{aligned} D &= |C| \times |T| = \text{dimensions of the search space} \\ \boldsymbol{\Theta} &= \{\theta_l \in \mathbb{R}^D\}_{l=0}^{n_p-1} = \text{positions of particles in the swarm} \\ \mathbf{V} &= \{\mathbf{v}_l \in \mathbb{R}^D\}_{l=0}^{n_p-1} = \text{velocity of particles in the swarm} \end{aligned} \tag{5}$$

Position and velocity of swarm particles are updated, and the fitness function is computed as

$$\begin{aligned} \boldsymbol{\Theta}(t+1) &= \boldsymbol{\Theta}(t) + \mathbf{V}(t+1) \\ \mathbf{V}(t+1) &= \mathbf{V}(t) + \varphi_1 \cdot \Big(P_{\text{best}} - \boldsymbol{\Theta}(t)\Big) + \varphi_2 \cdot \Big(G_{\text{best}} - \boldsymbol{\Theta}(t)\Big) \\ F(\theta_l) &= \text{SDF}^3(M_l) \end{aligned} \tag{6}$$

where t is the iteration number, φ_1, φ_2 are constants and P_{best} (and G_{best}) is the particles own (and neighbors) experience. Finally, local and global bests are updated as

$$\begin{aligned} P^l_{\text{best}} &= F(\theta_l) \text{ if } F(\theta_l) < F(P^l_{\text{best}}) \\ G_{\text{best}} &= \min_{l=0,\ldots n_p-1} P^l_{\text{best}} \end{aligned} \tag{7}$$

The mapping with the highest throughput is retained.

C. Constructing Static-Order Schedule and Estimating Throughput

To estimate throughput, the SDF^3 tool constructs a static order schedule for each core of the neuromorphic hardware. This is to arbitrate the access of shared resources of a core (input/output channel, synaptic memory, etc.) among neurons mapped to the core. A list-scheduler is used to construct these static-order schedules for all cores at once. The schedules are constructed via an execution of the clustered SNN graph mapped to the cores of hardware, assuming that for each core 50% of the available time wheel is allocated to the SNN graph. The latency to communicate spikes between cores is taken into account in the schedule construction. When a neuron becomes ready, it does not start its firing immediately. Instead the neuron is added to the ready list of the core it is bound to. When no neuron is firing on the core, the first ready neuron is removed from the list and its firing is started. The neuron ends firing after the time it takes to generate a spike. At this moment, the neuron is added to the schedule of the core. The execution ends as soon as a recurrent state is found. At this point, a finite-length schedule has been constructed for each core. After constructing the schedule, an optimization is performed to remove all recurrent occurrences of the same scheduling sequence. The static-order schedule on each core consists of a transient phase followed by a steady-state phase [60]. Throughput is computed as the inverse of the long-term period in the steady-state.

VII. Evaluation Methodology

We conduct all simulations on a Lambda workstation, which has AMD Threadripper 3960X with 24 cores, 128 MB cache, 128 GB RAM, and 2 RTX3090 GPUs. We evaluate 5 convolutional neural network (CNN) models – LeNet, AlexNet, ResNet (ResNet18), DenseNet, and VGG (VGG16). All these models are trained on the CIFAR-10 dataset. We use Keras [61] to train these models. Trained models are converted to SNN using the conversion toolbox [30], [62] and simulated using PyCARL [51] with the CARLsim backend simulator [63]. All spiking neurons are programmed as integrate-and-fire (IF) type [64]. The simulator is configured to use OxRRAM NVM model as the synaptic cell [65].

Our hardware simulation framework includes a cycle-level multi-core neuromorphic system simulator [21]. We configure this framework to simulate Loihi neuromorphic PEs with parameters listed in Table IV.

TABLE IV
Major simulation parameters extracted from Loihi [18].

Neuron technology	16nm CMOS (original design is at 14nm FinFET)
Synapse technology	HfO_2-based OxRRAM [65]
Supply voltage	1.0V
Energy per spike	23.6pJ at 30Hz spike frequency
Energy per routing	3pJ
Switch bandwidth	3.44 G. Events/s

VIII. Results and Discussions

A. Maximum Throughput

Figure 9 reports the maximum throughput obtained using the proposed design-flow compared to DFSynthesizer and SDFSNN for the five CNN applications. Results are normalized to DFSynthesizer. We make the following two key observations.

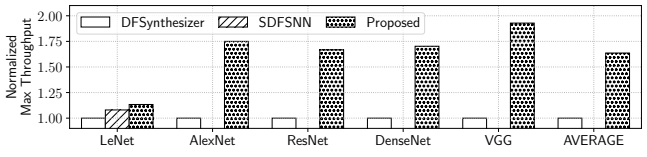


Fig. 9. Maximum throughput.

First, the maximum throughput of SDFSNN is 8% higher than DFSynthesizer. This is because, SDFSNN performs throughput analysis treating an entire SNN graph as an SDFG, and performing both partitioning and hardware placement at once during its analysis stage. DFSynthesizer, on the other hand, applies dataflow analysis technique on an SNN model that is already partitioned into clusters. Therefore, the search space of DFSynthesizer is smaller than SDFSNN (see Figure 7), resulting in lower maximum throughput. However, SDFSNN is not scalable for large problem sizes due to its integrated partitioning and placement steps. For these applications, SDFSNN fails to generate a mapping solution as we see in the figure. Second, maximum throughput of the proposed design flow is the highest for all CNN models. The maximum throughput is on average 63% higher than DFSynthesizer for all CNN models, and 5% higher than SDFSNN for the LeNet model. The improvement is because the proposed design flow uses an iterative approach, performing partitioning using the KL heuristic and throughput analysis exploiting the rich semantics of SDFG. Due to the use of KL heuristic, the proposed design flow is scalable to large problem sizes. Additionally, due to creating different partitioning alternatives and performing design-space exploration with them, the proposed design flow is able to explore a much larger throughput-buffer size search space than DFSynthesizer.

B. Buffer Size

Figure 10 reports the minimum buffer size needed to achieve a throughput constraint for each evaluated model using the three approaches. The throughput constraint is set to 70% of the highest throughput obtained using the proposed design flow. We selected this throughput constraint as a case study because both DFSynthesizer and SDFSNN are not able to find a mapping solution for throughput constraint set to anything higher than this value due to limited size of their exploration space (see Section VIII-C). Results for each application are normalized to DFSynthesizer. We make the following three key observations.

First, the minimum buffer size needed to achieve the throughput constraint is the least for the proposed design

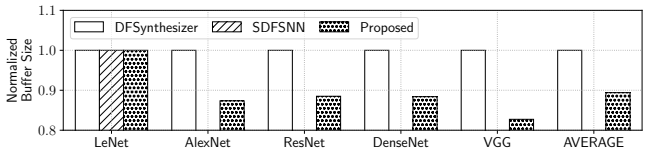


Fig. 10. Minimum buffer size.

flow (on average 10% lower than DFSynthesizer). This is because the proposed design flow is able to explore larger design space than DFSynthesizer, which we have discussed in Section VIII-A (see also Figure 7). Second, the buffer size needed for LeNet in order to achieve the throughput constraint is the same for all three approaches. Combining results of buffer size and maximum throughput for LeNet, we conclude that for a given amount of buffer size, the proposed design flow results in higher throughput than the two state-of-the-art dataflow-based mapping frameworks.

To give further insight, Figure 11 reports the minimum buffer size needed to achieve different throughput constraints using the proposed design flow. There are four settings evaluated for each application – minimum buffer size needed to achieve 70%, 80%, 90%, and 100% of the highest throughput (T_{max}). Results for each application are normalized to the buffer size needed to achieve the highest throughput. We make the following two key observations.

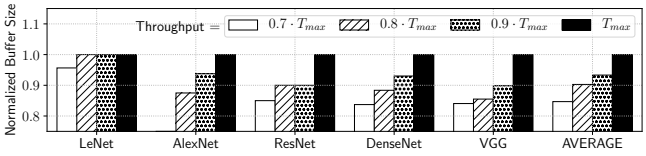


Fig. 11. Minimum buffer size for different throughput constraints.

First, to achieve 70%, 80%, and 90% of the highest throughput, the minimum buffer size needed in the proposed design flow is on average 15.3%, 9.7%, and 6.7% lower than the buffer size needed to achieve the highest throughput. These results show that for scalable throughput applications, i.e., those applications with acceptable throughput degradation, the buffer requirement of the hardware can be reduced significantly using the proposed design flow. Second, for LeNet, which is a smaller CNN model compared to the rest, there are only a fewer Pareto points generated using the design flow. Therefore, we see no change in the minimum buffer size for this application as we increase the throughput requirement from 80% to 100% of the highest throughput.

C. Design Space Explorations

Table V reports performance of the design-space exploration using the proposed design flow compared to DFSynthesizer. For the proposed design flow, we report results for three different settings of the user-defined parameter η. Results in Sections VIII-A and VIII-B are obtained by setting $\eta = 10$. Design-space exploration is compared in terms of the number of Pareto points generated during the exploration and the time (s) it takes to explore the design space. We make the following four key observations.

TABLE V
DESIGN SPACE EXPLORATION.

Model	DFSynthesizer		$\eta = 1$		$\eta = 5$		$\eta = 10$	
	Pareto Points	Exploration Time (s)	Pareto Points	Exploration Time (s)	Pareto Points	Exploration Time (s)	Pareto Points	Exploration Time (s)
LeNet	4	108	4	432	4	3024	4	6288
AlexNet	4	1463	7	4389	9	39501	10	75519
ResNet	5	2723	8	10892	9	65352	11	118872
DenseNet	4	4399	8	8798	12	70384	13	176144
VGG	6	6563	7	13126	10	131260	12	293940

First, the number of Pareto points obtained using the proposed design flow is higher than DFSynthesizer. For smaller models such as LeNet, the number of Pareto points are comparable. However, for larger models, the proposed design flow generates higher number of Pareto points than DFSynthesizer, resulting in higher maximum throughput (Section VIII-A) and lower buffer requirement (Section VIII-B). Second, the number of Pareto points increases with increase in η. This is because with more iterations of the partitioning algorithm (Algorithm 1), the proposed design flow can explore larger design space, leading to generating more Pareto points. Third, the exploration time using DFSynthesizer is the least. This is because, DFSynthesizer's design space exploration is limited to exploration using the clusters only, which are fewer than the number of neurons. The proposed design flow explores the design space using neurons. Therefore, the exploration time is higher than DFSynthesizer, even with $\eta = 1$. Finally, the exploration time of the proposed design flow increases with increase in η. Designer can select η based on the required throughput-buffer size tradeoff.

IX. CONCLUSIONS

We have developed a design flow for predictable mapping of SNN-based machine learning models to many-core neuromorphic hardware. The flow consists of an iterative approach to partition an SNN into clusters such that each cluster can be mapped to a core of the many-core hardware. The partitioning step minimizes inter-cluster spike communication, which improves latency. The design flow then uses an instance of the Particle Swarm Optimization (PSO) to generate SNN mapping solutions, exploring the design space between throughput and buffer size requirement of the cores. Pareto optimal mappings are provided to system designer. We evaluate our design flow using large-scale spiking CNN models. Results demonstrate 63% higher maximum throughput and 10% lower buffer requirement than state-of-the-art mapping solutions.

ACKNOWLEDGEMENT

This material is based upon work supported by the U.S. Department of Energy under Award Number DE-SC0022014 and by the National Science Foundation under Grant Nos. CNS-2008167 and CCF-1937419.

REFERENCES

[1] C. Mead, "Neuromorphic electronic systems," *Proc. of the IEEE*, 1990.
[2] S. Bose *et al.*, "Is my neural network neuromorphic? taxonomy, recent trends and future directions in neuromorphic engineering," *ACSSC*, 2019.
[3] D. V. Christensen *et al.*, "2021 roadmap on neuromorphic computing and engineering," *arXiv*, 2021.
[4] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, 1997.
[5] F. Catthoor *et al.*, "Very large-scale neuromorphic systems for biological signal processing," in *CMOS Circuits for Biological Sensing and Processing*, 2018.
[6] C. Liu *et al.*, "A spiking neuromorphic design with resistive crossbar," in *DAC*, 2015.
[7] A. Das *et al.*, "Mapping of local and global synapses on spiking neuromorphic hardware," in *DATE*, 2018.
[8] A. N. Burkitt, "A review of the integrate-and-fire neuron model: I. homogeneous synaptic input," *Biological Cybernetics*, 2006.
[9] J. Wu *et al.*, "A tandem learning rule for effective training and rapid inference of deep spiking neural networks," *TNNLS*, 2021.
[10] X. Liu *et al.*, "Neu-NoC: A high-efficient interconnection network for accelerated neuromorphic systems," in *ASP-DAC*, 2018.
[11] A. Balaji *et al.*, "Exploration of segmented bus as scalable global interconnect for neuromorphic computing," in *GLSVLSI*, 2019.
[12] C. Frenkel *et al.*, "A 0.086-mm^2 12.7-pj/sop 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28-nm CMOS," *TBCAS*, 2018.
[13] J. Stuijt *et al.*, "μBrain: An event-driven and fully synthesizable architecture for spiking neural networks," *Frontiers in Neuroscience*, 2021.
[14] S. Moradi *et al.*, "A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs)," *TBCAS*, 2017.
[15] J. Schemmel, "The BrainScaleS accelerated analogue neuromorphic architecture," *Brain-inspired Computing*, 2021.
[16] S. Furber *et al.*, "The SpiNNaker project," *Proc. of the IEEE*, 2014.
[17] B. Benjamin *et al.*, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proceedings of the IEEE*, 2014.
[18] M. Davies *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, 2018.
[19] M. V. Debole *et al.*, "TrueNorth: Accelerating from zero to 64 million neurons in 10 years," *Computer*, 2019.
[20] Y. Ji *et al.*, "NEUTRAMS: Neural network transformation and co-design under neuromorphic hardware constraints," in *MICRO*, 2016.
[21] A. Balaji *et al.*, "NeuroXplorer 1.0: An extensible framework for architectural exploration with spiking neural networks," in *ICONS*, 2021.
[22] A. Amir *et al.*, "Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores," in *IJCNN*, 2013.
[23] F. Galluppi *et al.*, "A hierachical configuration system for a massively parallel neural hardware platform," in *CF*, 2012.
[24] A. Balaji *et al.*, "Mapping spiking neural networks to neuromorphic hardware," *TVLSI*, 2020.
[25] T. Titirsha *et al.*, "On the role of system software in energy management of neuromorphic computing," in *CF*, 2021.
[26] A. Balaji *et al.*, "Run-time mapping of spiking neural networks to neuromorphic hardware," *JSPS*, 2020.
[27] A. Das *et al.*, "Dataflow-based mapping of spiking neural networks on neuromorphic hardware," in *GLSVLSI*, 2018.
[28] A. Balaji *et al.*, "A framework for the analysis of throughput-constraints of SNNs on neuromorphic hardware," in *ISVLSI*, 2019.
[29] S. Song *et al.*, "Compiling spiking neural networks to neuromorphic hardware," in *LCTES*, 2020.
[30] S. Song *et al.*, "DFSynthesizer: Dataflow-based synthesis of spiking neural networks to neuromorphic hardware," *TECS*, 2021.
[31] A. Balaji *et al.*, "Enabling resource-aware mapping of spiking neural networks via spatial decomposition," *ESL*, 2020.
[32] A. Balaji *et al.*, "Compiling spiking neural networks to mitigate neuromorphic hardware constraints"," in *IGSC Workshops*, 2020.
[33] C.-K. Lin *et al.*, "Mapping spiking neural networks onto a manycore neuromorphic architecture," in *PLDI*, 2018.
[34] S. Song *et al.*, "Dynamic reliability management in neuromorphic computing," *JETC*, 2021.
[35] A. Balaji *et al.*, "A framework to explore workload-specific performance and lifetime trade-offs in neuromorphic computing," *CAL*, 2019.
[36] S. Song *et al.*, "Improving dependability of neuromorphic computing with non-volatile memory," in *EDCC*, 2020.
[37] S. Song *et al.*, "A case for lifetime reliability-aware neuromorphic computing," in *MWSCAS*, 2020.
[38] S. Kundu *et al.*, "Special session: Reliability analysis for ML/AI hardware," in *VTS*, 2021.
[39] S. Song *et al.*, "Improving inference lifetime of neuromorphic systems via intelligent synapse mapping," in *ASAP*, 2021.
[40] T. Titirsha *et al.*, "Reliability-performance trade-offs in neuromorphic computing," in *IGSC Workshops*, 2020.
[41] T. Titirsha *et al.*, "Thermal-aware compilation of spiking neural networks to neuromorphic hardware," in *LCPC*, 2020.
[42] T. Titirsha *et al.*, "Endurance-aware mapping of spiking neural networks to neuromorphic hardware," *TPDS*, 2021.
[43] E. Lee *et al.*, "Synchronous data flow," *Proceedings of the IEEE*, 1987.
[44] S. Sriram *et al.*, *Embedded Multiprocessors; Scheduling and Synchronization*, 2000.
[45] L. Jiashu *et al.*, "A design flow for partially reconfigurable heterogeneous multi-processor platforms," in *RSP*, 2012.
[46] A. Das *et al.*, *Reliable and Energy Efficient Streaming Multiprocessor Systems*. Springer, 2018.
[47] S. Stuijk *et al.*, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *DAC*, 2006.
[48] A. K. Singh *et al.*, "RAPIDITAS: RAPId design-space-exploration incorporating trace-based analysis and simulation," in *DSD*, 2013.
[49] A. Das *et al.*, "Fault-aware task re-mapping for throughput constrained multimedia applications on NoC-based MPSoCs," in *RSP*, 2012.
[50] D. Goodman *et al.*, "The brian simulator," *Front. in Neuroscience*, 2009.
[51] A. Balaji *et al.*, "PyCARL: A PyNN interface for hardware-software co-simulation of spiking neural network," in *IJCNN*, 2020.
[52] B. W. Kernighan *et al.*, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, 1970.
[53] M. Adé *et al.*, "Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets," in *DAC*, 1997.
[54] M. Geilen *et al.*, "Minimising buffer requirements of synchronous dataflow graphs with model checking," in *DAC*, 2005.
[55] R. Govindarajan *et al.*, "Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks," *JSPS*, 2002.
[56] S. Stuijk *et al.*, "SDF^3: SDF for free," in *ACSD*, 2006.
[57] S. Stuijk *et al.*, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *DAC*, 2006.
[58] J. Kennedy *et al.*, "Particle swarm optimization," in *ICNN*, 1995.
[59] A. H. Ghamarian *et al.*, "Throughput analysis of synchronous data flow graphs," in *ACSD*, 2006.
[60] A. Das *et al.*, "Energy-aware task mapping and scheduling for reliable embedded computing systems," *TECS*, 2014.
[61] A. Gulli *et al.*, *Deep learning with Keras*, 2017.
[62] A. Balaji *et al.*, "Power-accuracy trade-offs for heartbeat classification on neural networks hardware," *JOLPE*, 2018.
[63] T. Chou *et al.*, "CARLsim 4: An open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters," in *IJCNN*, 2018.
[64] S. Fusi *et al.*, "Collective behavior of networks with linear (VLSI) integrate-and-fire neurons," *Neural Computation*, 1999.
[65] A. Mallik *et al.*, "Design-technology co-optimization for OxRRAM-based synaptic processing unit," in *VLSIT*, 2017.