SpikeHard: Efficiency-Driven Neuromorphic Hardware for Heterogeneous Systems-on-Chip

JUDICAEL CLAIR, GUY EICHLER, and LUCA P. CARLONI, Columbia University, USA

Neuromorphic computing is an emerging field with the potential to offer performance and energy-efficiency gains over traditional machine learning approaches. Most neuromorphic hardware, however, has been designed with limited concerns to the problem of integrating it with other components in a heterogeneous System-on-Chip (SoC). Building on a state-of-the-art reconfigurable neuromorphic architecture, we present the design of a neuromorphic hardware accelerator equipped with a programmable interface that simplifies both the integration into an SoC and communication with the processor present on the SoC. To optimize the allocation of on-chip resources, we develop an optimizer to restructure existing neuromorphic models for a given hardware architecture, and perform design-space exploration to find highly efficient implementations. We conduct experiments with various FPGA-based prototypes of many-accelerator SoCs, where Linux-based applications running on a RISC-V processor invoke Pareto-optimal implementations of our accelerator alongside third-party accelerators. These experiments demonstrate that our neuromorphic hardware, which is up to 89× faster and 170× more energy efficient after applying our optimizer, can be used in synergy with other accelerators for different application purposes.

CCS Concepts: • **Hardware** → **Neural systems**; • **Computer systems organization** → *Heterogeneous (hybrid) systems*; Real-time system architecture;

Additional Key Words and Phrases: Neuromorphic computing, machine learning, spiking neural network

ACM Reference format:
Judicael Clair, Guy Eichler, and Luca P. Carloni. 2023. SpikeHard: Efficiency-Driven Neuromorphic Hardware for Heterogeneous Systems-on-Chip. *ACM Trans. Embedd. Comput. Syst.* 22, 5s, Article 106 (September 2023), 22 pages.
https://doi.org/10.1145/3609101

1 INTRODUCTION

Inspired by the biological nature of computation in the brain, Neuromorphic Computing is a field of Artificial Intelligence (AI) with a promise to provide higher energy efficiency for smart embedded devices [24, 37, 38] and autonomous machines [30, 31, 35]. In neuromorphic computing, the learning model is the Spiking Neural Network (SNN). As in biological neural networks, an SNN is built from neurons that communicate with one another through channels called axons [5], and

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES), 2023.
This work was partially supported by the National Science Foundation under Award No. 1764000.
Authors' address: J. Clair, G. Eichler, and L. P. Carloni, Dept. of Computer Science, Columbia University, Mudd Building, 500 W 120th St, New York, NY 10027, USA; emails: jsc2268@columbia.edu, {guyeichler, luca}@cs.columbia.edu.

1539-9087/2023/09-ART106 $15.00
https://doi.org/10.1145/3609101

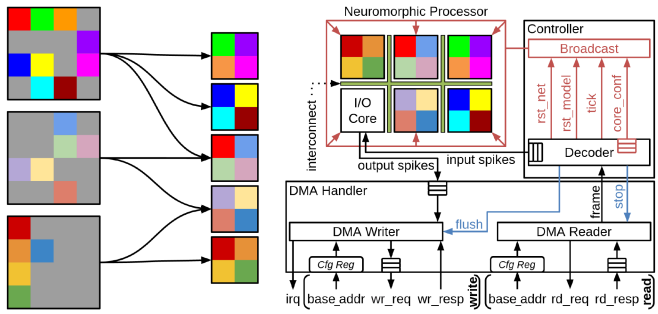

Fig. 1. SNN model restructuring, and accelerator design for SoC integration.

information flows through the network via events called spikes [18]. While specialized hardware for SNNs can be very efficient [2, 13, 25], the ultimate goal is to approach the computation and communication efficiency of the human brain [32]. In contrast, traditional approaches to machine learning, such as deep learning, use dense computation, which has limited scalability and energy-efficiency [8]. To mitigate these problems, deep learning models, such as a Convolutional Neural Network (CNN), can be converted into SNNs [14].

Several neuromorphic architectures have already been developed into SoCs, such as Loihi 2 [23], TrueNorth [2], and Akida [6]. However, these designs focus mostly on the neuromorphic processing, with limited attention to the aspects of its integration with components from other application domains in a heterogeneous SoC. To create embedded systems that utilize SNNs among other computational kernels [3, 7, 8, 38], we would like to design heterogeneous many-accelerator SoCs capable of accelerating a variety of applications.

Towards this goal, we developed SpikeHard, a runtime-programmable neuromorphic hardware accelerator designed under the premises of high efficiency, scalability, and seamless integration in heterogeneous many-accelerator SoCs. As shown in Figure 1, our design methodology follows innovative restructuring of SNNs, which optimally remaps an SNN model to a desired hardware architecture. This promotes multi-objective design-space exploration (DSE) that helps to meet strict performance and energy-efficiency requirements in a many-accelerator SoC.

SpikeHard was designed with an interface that supports direct-memory access (DMA) and real-time reconfigurability, which simplifies the integration of neuromorphic hardware in an SoC. We deployed many-accelerator SoCs with different versions of SpikeHard on FPGA, classifying the MNIST dataset [15], and approximating Vector-Matrix Multiplication (VMM). By using software applications running on a 64-bit CVA6 RISC-V processor [39], we show that SpikeHard can be invoked together with other accelerators, and without hurting the overall performance of the applications. To the best of our knowledge, this is the first work to standardize a neuromorphic accelerator and integrate it in a heterogeneous many-accelerator SoC. Our main contributions are:

(1) A novel optimization algorithm that restructures SNN models to minimize resource utilization of neuromorphic processors.
(2) A programmable neuromorphic hardware accelerator that can be easily integrated into a heterogeneous SoC via a scalable interface, which could be repurposed for a variety of existing neuromorphic processors with little to no modification.
(3) The DSE of a neuromorphic accelerator through breaking dependencies between SNNs and hardware architectures.

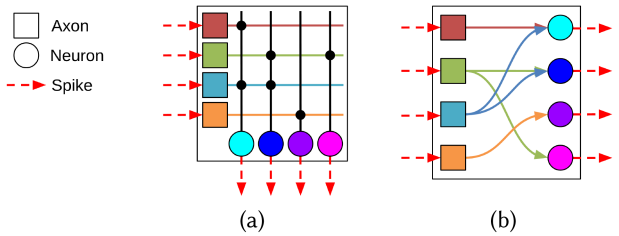

Fig. 2. Neuromorphic core depicted as a crossbar (a), and as a graph (b).

(4) The integration of neuromorphic and non-neuromorphic accelerators on the same SoC, and the investigation of their performance at the level of the application.
(5) The open-source release of SpikeHard.

2 BACKGROUND

2.1 Spiking Neural Network (SNN)

The computational model used in neuromorphic computing is the Spiking Neural Network (SNN), which mimics the behavior of the human brain. In particular, an SNN is composed of computational elements called neurons and channels called axons. A neuron is an event-driven state machine controlled by a set of 1-bit inputs. When an input is high, the event is called an input spike. Depending on its state, a neuron can react to a given input configuration by sending an output spike to other neurons via an axon.

2.2 Neuromorphic Cores

In hardware design, the SNN model is split into neuromorphic cores [2, 13], where each core represents a subset of axons and neurons from the SNN. As shown in Figure 2(a), a core implements a crossbar that receives input spikes through its axons. Once an axon receives a spike, the spike triggers all of the connected neurons. If a neuron is ready, it will send an output spike directed to an axon in another core. As shown in Figure 2(b), the crossbar can also be viewed as a directed graph that captures the one-way connections between axons and neurons. *In this paper, the term core refers to a neuromorphic core.*

2.3 Event-Driven Computation

The computation in an SNN is event-driven. Specifically, the result of a computation depends on the exact timing of events (or spikes). In several architectures, these events are controlled by time intervals delineated by ticks. TrueNorth [2] and RANC [26] use a global tick control signal whose pulse notifies all the cores to execute the next step of computation, which involves calculating the next output spikes, and sending them to the corresponding axons. On the other hand, Loihi [13] uses a non-global tick signal. That is, once a core has finished processing the current computation interval, it will synchronize with its adjacent cores to advance to the next interval (i.e., tick).

2.4 Open-Source Neuromorphic Hardware

TrueNorth [2] and Loihi 2 [23] are the most prominent neuromorphic chips as of today. Both contain ~1 million neurons, and over 100 million synapses (connections between neurons). However, the designs of these chips are closed-source. In contrast, the Reconfigurable Architecture for Neuromorphic Computing (RANC) [26] has been released open-source and is an excellent baseline design to explore different optimizations for accelerating SNNs with specialized hardware. Similar to the state-of-the-art TrueNorth and Loihi 2, RANC is based on a multi-core architecture that

can execute pre-trained SNN models. RANC leverages a similar tick control as in TrueNorth, but unlike Loihi 2, both RANC and TrueNorth do not support broadcasting a spike from one neuron to multiple axons. Thanks to its high reconfigurability, RANC was successfully tested on FPGA with SNN-based CNN classifiers, MNIST [15] image classifiers, and VMM models [17]. Overall, these features make RANC a great candidate for integration in a heterogeneous SoC.

2.5 Integration Challenges in a Heterogeneous SoC

The successful integration of neuromorphic hardware as an accelerator in a heterogeneous SoC depends on multiple features. We would like to be able to program the accelerator by means of software applications running on a general-purpose processor [10]. For computational flexibility, the processor should be able to offload the processing of an SNN model to the accelerator at runtime, by transferring input data to the accelerator and receiving output data from the accelerator. To that end, the accelerator interface must be standardized to enable data movement between the accelerator and the rest of the SoC (e.g., main memory and third-party accelerators).

In a heterogeneous SoC, multiple hardware accelerators typically run in parallel after being invoked by a given application. Hence, it is important that a neuromorphic accelerator does not become a performance bottleneck compared to other accelerators. Additionally, if the accelerator is not energy-efficient, the SoC can become unsuitable for real-time applications, as well as for devices in constrained environments [24, 37, 38]. These challenges must be addressed by effectively exploring the design space of SNN accelerators, simplifying their integration in an SoC, as well as prototyping and testing the overall SoC on FPGA [16].

3 MODEL RESTRUCTURING

Fitting an SNN model to a neuromorphic hardware architecture is a non-trivial task. For this reason, we developed a model restructuring method that partitions an SNN into fine-grained groups of neurons and axons. The method reorganizes these groups efficiently in neuromorphic cores by solving an optimization problem that minimizes resource utilization. In Section 6, we show that model restructuring can be used productively to expand the design space of the SpikeHard accelerator.

3.1 Model Generation

Our SNN models and neuromorphic hardware architecture for MNIST classification and VMM are based on a generation tool provided by RANC. The hardware architecture (e.g., the neuron and axon capacities of a core, and number of cores) is given as input alongside training data. Then, RANC generates the SNN model in a sequential manner, where the connections between axons and neurons are calculated at the level of each individual core, according to the architectural parameters.

This model generation has two main weaknesses. First, the sequential generation at the level of each core imposes a dependency between the computational model and the hardware architecture. Second, after model generation, there is no process that checks for redundancy and efficient resource utilization.

As part of developing SpikeHard, we initially focused on the second weakness by optimizing SNNs that were generated by RANC with a dependency to a hardware architecture. In particular, we analyzed the original placement of neurons and axons, and provided an optimized placement according to the original capacity of each core. As shown in Figure 3, our method dramatically improves inner core utilization from an average of 50% for the original model to 85%, while also reducing the overall core count by 40%. In this way, we eliminate the redundancies at the level of a core and of the full neuromorphic processor, thereby solving the second weakness.

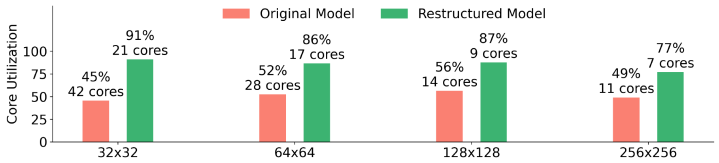

Fig. 3. Core utilization per core capacities: Axons × Neurons.

In addition, we succeeded in transferring SNNs generated for specific hardware architectures into other architectures (Section 6). In doing so, we have partially addressed the first weakness by relaxing the dependency between model generation and the hardware architecture. In the future, our method could be incorporated in the initial model generation itself, as long as the connections between neurons and axons in the SNN can be determined. Our model restructuring method is detailed in the following subsections.

3.2 Minimal Connected Component Analysis

Given an initial mapping of an SNN to cores, we analyze the connections between neurons and axons in the SNN so as to partition the SNN into Minimal Connected Components (MCCs). Inspired by graph theory [11], MCCs are the smallest subsets of connected neurons and axons in a core. That is, for a set S of neurons and axons in a core to be an MCC, two conditions must be satisfied:

(1) All neurons and axons in S are connected only to neurons and axons in S.
(2) No subset of S can be removed from S without breaking a neuron-axon connection.

During MCC analysis, we only consider intra-core connections from axons to neurons, and not inter-core connections from neurons to axons. However, inter-core connections are preserved by appropriately updating the output axon identifier of each neuron at the end of model restructuring.

As shown in Figure 4, we partition all the cores into MCCs and then solve a problem similar to bin packing [28]. The solution to this problem involves placing a set of items of varying size into a minimum number of bins with limited capacity. In our case, the items are MCCs and the bins are the cores, where the capacity of a core is two-dimensional and determined by the number of neurons and axons available. We now formulate our MCC packing problem as an Integer Linear Program (ILP), which has the following indicator variables:

$$x_{ij}, y_j \in \{0, 1\} \qquad \forall i \in \{1, \ldots, \#\text{MCCs}\}, j \in \{1, \ldots, \#\text{Cores}\}$$

where x_{ij} is 1 only when the i-th MCC is placed in the j-th core, and y_j is 1 only when the j-th core is not empty. The objective is to maximize inner core utilization by minimizing the number of non-empty cores: $\min\left(\sum_j y_j\right)$. The solution is subject to the following constraints:

$$\forall i. \sum_j x_{ij} = 1 \tag{1}$$

$$\forall j. \sum_i x_{ij} n_i \leq y_j N \tag{2}$$

$$\forall j. \sum_i x_{ij} a_i \leq y_j A \tag{3}$$

where N is the neuron capacity per core, A the axon capacity per core, n_i the number of neurons in the i-th MCC, and a_i the number of axons in the i-th MCC. Constraint (1) ensures that each MCC is placed in exactly one core. Constraints (2) and (3), respectively, prevent the number of neurons and axons in each core from exceeding the capacity of the core. The optimal solution to this ILP

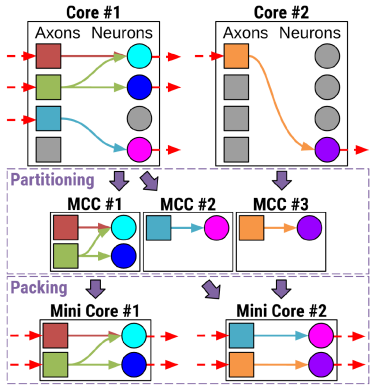

Fig. 4. Partitioning cores into MCCs and packing them into smaller cores.

can be computed using an off-the-shelf ILP solver, notably SCIP [1]. This forms the basis for our model restructuring algorithm.

3.3 Core Placement within the Neuromorphic Processor

The solution to the MCC packing problem provides an optimized clustering of MCCs for a fixed number of cores with specific capacity. However, the location of cores within the neuromorphic processor should also be taken into account. As illustrated in Figure 1(b), the interconnect between cores is a 2D grid that allows for communication between all cores. An increase in interconnect traffic is expected between a core with a neuron that sends spikes, and a core with an axon that receives these spikes. Hence, the placement of cores inside SpikeHard can determine the level of congestion and might affect the overall performance of the accelerator. In this work, however, we focused on MCC packing itself and left this exploration for future work. Instead, we organize the utilized cores so that the resulting shape of the grid of cores is as close as possible to a square while also not exceeding the maximum grid dimensions of the new architecture. In this way, spike traffic should a priori be distributed across the interconnect.

3.4 Model Restructuring Algorithm

The entry point of the optimization algorithm that restructures SNNs is `restructure_model`, which is presented in Figure 5 and takes as input (1) *old_model*: the SNN model to be restructured that has already been mapped to an existing architecture; (2) *new_core_capacity*: a 2D tuple specifying the axon and neuron capacity of each core in the new architecture; and (3) *new_max_grid_dims*: a 2D tuple specifying the maximum dimensions for the grid of cores in the new architecture. The only requirement on *new_core_capacity* is that each MCC of *old_model* must be able to fit within a new core. Given *new_core_capacity*, *new_max_grid_dims* can be automatically deduced by gradually increasing its value until model restructuring succeeds.

We first `partition` the model into its constituent MCCs (Section 3.2). The implementation is described in Figure 6. At a higher level, for every core used by the model, we perform the following. For each utilized axon within the core, we skip processing the axon if it has already been added to an MCC. Otherwise, we create a new MCC containing the axon along with the neurons that receive spikes from the axon. We then find all the axons that send spikes to each of these neurons and add them to the MCC as well. This process continues recursively until no new axons nor neurons are added to the MCC, yielding a complete MCC. The result of this procedure is illustrated in Figure 4.

```
procedure restructure_model(old_model, new_core_capacity, new_max_grid_dims):
  mccs ← partition(old_model)
  new_used_cores ← pack(mccs, new_core_capacity, new_max_grid_dims)
  new_model ← place(new_used_cores, new_max_grid_dims)
  return new_model
```

Fig. 5. Main procedure for model restructuring.

```
procedure partition(model):
  mccs ← ∅
  for core ∈ cores utilized by model :
    rem_axons ← set of axons within core that are utilized by model
    while rem_axons ≠ ∅ :
      mcc_axons, mcc_neurons, axons_to_add_to_mcc ← ∅
      axons_to_add_to_mcc.add(rem_axons.get_first_element())
      while axons_to_add_to_mcc ≠ ∅ :
        axon_i ← axons_to_add_to_mcc.pop()
        mcc_axons.add(axon_i)
        rem_axons.remove(axon_i)
        for neuron ∈ neurons that receive spikes from axon_i :
          if neuron ∉ mcc_neurons :
            mcc_neurons.add(neuron)
            for axon_j ∈ axons that send spikes to neuron :
              if axon_j ∉ (axons_to_add_to_mcc ∪ mcc_axons) :
                axons_to_add_to_mcc.add(axon_j)
      mcc ← (mcc_axons, mcc_neurons)
      mccs.add(mcc)
  return mccs
```

Fig. 6. Procedure to partition model into its constituent MCCs.

We then pack the MCCs of the model into the minimum number of new cores by solving the MCC packing problem with the approach described in Section 3.2, where the new neuron capacity N and axon capacity A are given by *new_core_capacity*, and the maximum number of new cores (so as to bound j) is the product of the dimensions defined by *new_max_grid_dims*.

Given the clustering of MCCs to a minimum number of new cores, the final step is to `place` these cores into a 2D grid. The pseudocode for this procedure is shown in Figure 7. As we alluded to in Section 3.3, the resulting grid shape is as close as possible to a square while also not exceeding the maximum grid dimensions given by *new_max_grid_dims*; noting that grid position $(0, 0)$ is reserved for the specialized I/O core (Figure 1(b)).

4 NEUROMORPHIC ACCELERATOR

As shown in Figure 1(b), we encapsulated the neuromorphic design as a hardware accelerator. We implemented an interface that supports both 32-bit and 64-bit DMA, and a controller that enables runtime reconfiguration of SpikeHard. We used the Verilog hardware description language for the implementation.

```
1  procedure place(cores, max_dims):
2    model ← initialize_empty_model()
3    x_dim, y_dim ← 1
4    while (cores.size() + 1) > (x_dim · y_dim) :
5      if x_dim ≠ max_dims.x and (y_dim = max_dims.y or y_dim ≥ x_dim) :
6        x_dim += 1
7      else:
8        y_dim += 1
9    assert(x_dim ≤ max_dims.x and y_dim ≤ max_dims.y)
10   for i ← 1 to cores.size() :
11     y ← ⌊i / x_dim⌋
12     x ← i − y · x_dim
13     model.add_core(cores.pop(), x, y)
14   return model
```

Fig. 7. Procedure to place cores within the neuromorphic processor.

4.1 Runtime Programmability

RANC only allows the SNN model to be configured at design time [26]. Specifically, each core stores model parameters, such as how neurons and axons are connected to one another, in immutable local buffers initialized via Verilog `initial` blocks. Consequently, in the case of an FPGA implementation, loading a new model to RANC requires synthesizing a new bitstream and reconfiguring the FPGA, which is time-consuming and impractical for real-time systems. In the case of an ASIC implementation, the immutable buffers would have to be initialized using an additional mechanism instead of `initial` blocks. Nevertheless, without programmability, the ASIC design would be unable to accommodate multiple SNN models at runtime.

In contrast, SpikeHard is a flexible hardware accelerator whose configuration can be programmed at runtime to process various SNN models. The neuromorphic processor inside SpikeHard uses the same core architecture as RANC with a few changes to support runtime programmability. To load a new model at runtime, SpikeHard overwrites the aforementioned local buffers with new model parameters. To this end, as illustrated in Figure 1(b), the new model parameters are broadcasted one after the other to all cores while being streamed from main memory. Alongside each parameter is a pointer, which specifies the destination core and buffer address. The destination core then writes the parameter to the specified address. Given a SpikeHard accelerator embedded with a grid of cores and a fixed core capacity, multiple different SNN models can be offloaded onto the accelerator by simply restructuring the models to fit within the core capacity and the grid dimensions (as described in Section 3.4).

Moreover, in RANC, the tick control signal that marks the timing of computation events (Section 2.3) is set by design with the same fixed frequency for all models. In SpikeHard, we made the tick frequency programmable in real-time, and optimized it for each of the workloads in our test environment (Section 6).

As illustrated in Figure 1(b), the controller decodes data fetched via DMA, which may result in one of four commands listed in Table 1. The `core_conf` command modifies the model parameters of a specific core. The `rst_net` command initializes the loaded model to be ready for the next computation. The `rst_model` command unloads the current model from the accelerator. The `tick` command marks the beginning of the next computation interval. All the information to execute these commands is passed to the broadcast block, which in turn configures the cores.

Table 1. Commands Broadcasted to All Cores of the Neuromorphic Processor

Name	Description
`rst_net`	Reset dynamic state of loaded model.
`rst_model`	Unload model from accelerator.
`tick`	Signal beginning of next computation interval.
`core_conf`	Configure neurons and axons of a given core.

Table 2. Data Frames

Acronym	Frame Type	Description	Payload
RST	Reset	Reset model or network.	N/A*
CD	Core Data	Model parameters for a given core.	Raw core-local buffer data.
IN	Input Spikes	Send spikes to specific axons.	Contiguous array of spikes.
OUT	Output Spikes	Output spikes for a given tick.	Contiguous array of spikes.
TICK	Tick	Tick a given number of times.	N/A*
TERM	Terminate	End-of-File (EOF) token.	N/A*

*Frame does not have a payload.

4.2 DMA Handler

To be successfully integrated into a general-purpose SoC, SpikeHard must support off-chip memory access. To this end, we implemented a DMA handler that can read and write to main memory via DMA transactions.

As shown in Figure 1(b), at invocation time, base addresses to main memory are loaded into 32-bit configuration registers. The DMA reader is then able to send read requests to main memory containing an address along with the number of bytes to read. Read responses from main memory are stored in a 4-slot buffer to enable pipelining and forwarded by the DMA reader to the controller, which decodes the data. Write requests are similarly stored in a 4-slot buffer with an address alongside the data to be stored in main memory. A write response signals that the DMA write was accepted.

The DMA handler reads input frames, which hold the information to program and run the SNN model on SpikeHard, and writes output frames that contain the computation results. Each frame has a fixed-size header of 128 bits, where the first 3 bits indicate the frame type, and the other bits are frame-type dependent. A frame may also have a payload adjacent to the header, where the payload size is stored in the header. Table 2 gives a summary of each frame type and its contents. The unit of data transferred on the DMA bus is equal to the bus width, which is either 32-bit or 64-bit. Thus, the end of each payload is zero-padded so that each frame is aligned to the bus width; thereby making the payloads aligned as well. No padding is used within a payload as SpikeHard appropriately parses data of arbitrary word width. Each DMA transaction can span multiple units of data. In this way, reading and writing a frame to memory takes at most two transactions, one for the header and optionally another one for the payload.

4.3 Data Protocol

To read a stream of input frames and produce a stream of output frames, SpikeHard uses the data protocol illustrated in Figure 8 (the acronyms are defined in Table 2).

First, SpikeHard reads a reset (RST) frame. This can initiate a hard reset that will unload the model (`rst_model`) by disabling all the cores, or initiate a soft reset, which will only initialize the state of all neurons in the SNN and remove any spikes in transit (`rst_net`).

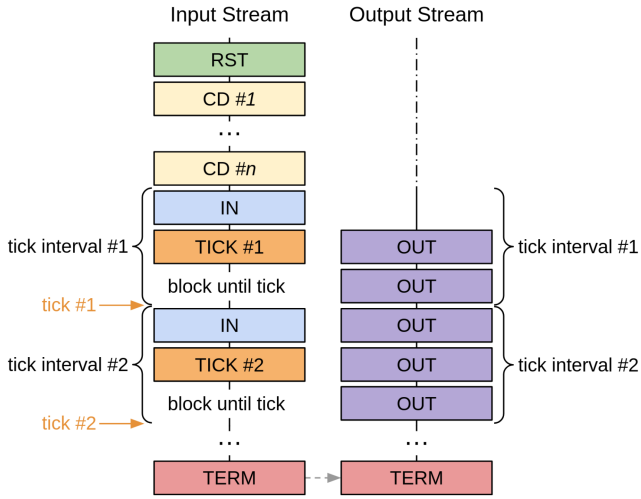

Fig. 8. Sequence of reads from input stream and writes to output stream.

Then, core data (CD) frames are read to configure the cores (`core_conf`). Each CD frame contains a core identifier in the header, and the model data in the payload. Cores that do not receive a CD frame stay disabled.

After the model is loaded, input spikes (IN) frames are read. These spikes represent the input data (e.g., MNIST images [15] to classify). IN frames are decoded into input spikes, which get transferred to the designated I/O core (Figure 1(b)). The I/O core is a specialized core that receives all the input spikes and sends them to the other cores through the interconnect [26].

Following an IN frame, a tick (TICK) frame is read. A TICK frame specifies a number of ticks, and a time interval (in clock cycles). The controller will only proceed to decoding the next input frame after ticking the specified number of times, where a `tick` is generated every time a counter has reached the specified number of clock cycles. In the meantime, during each tick time interval, output spikes are tagged with the current interval identifier, and sent out as output spikes (OUT) frames, where the frame header stores the interval identifier and the payload contains the spikes.

The computation is finished once a terminate (TERM) frame is read. This causes the controller to send a `flush` signal to the DMA writer, and a `stop` signal to the DMA reader to prevent it from fetching more frames (Figure 1(b)). The `flush` signal tells the DMA writer to send out all remaining OUT frames and finish execution. At the end, the DMA writer also sends a TERM frame to mark the end of the output stream.

4.4 Accelerator Interface Scalability and Reusability

Our accelerator interface supports DMA, which is necessary for loosely-coupled accelerators in heterogeneous SoCs, especially when multiple accelerators are concurrently accessing memory [10, 12]. Moreover, our interface is scalable since we can add new frame types to both the input and output streams. Additionally, there is currently no restriction as to the location of specific frame types and their ordering in the input stream. Thus, the sequence of input frames discussed in Section 4.3 can be arbitrarily reordered to achieve more advanced behaviors unsupported by RANC. For instance, a CD frame can be sent after a TICK frame, which would appropriately modify the specified core. In this way, a model can be partially reconfigured during execution.

In both RANC and SpikeHard, every neuron-axon pair within a core is processed sequentially once per tick. That is, for each pair, if the neuron is connected to the axon and the axon just spiked,

the neuron state is updated. This computation is deterministic and has time complexity $O(AN)$, where A and N are the number of axons and neurons per core, respectively. Accordingly, RANC uses an independent tick generator implemented in hardware with an immutable tick frequency specified at design time based on the capacity of each core.

To improve performance and energy-efficiency, the tick frequency must be maximized within the limits of functional correctness of the neuromorphic computation. However, maximizing an immutable tick frequency with respect to the expected computation time within a core can create issues when scaling to a larger system. Specifically, the effective maximum tick frequency can vary during runtime as it depends on the amount of spike traffic and congestion not only within the neuromorphic processor but also to and from main memory. If the tick frequency is too high, spikes may not arrive at their destination on time.

Since different SNN models can generate dissimilar spike traffic, models running on the same hardware can have different optimal tick frequencies. The optimal frequency may also vary during model execution due to changing spike traffic patterns. For instance, the input vector for VMM is provided as input spikes at the start, and no further input is provided during subsequent ticks while the model converges to a solution.

The optimal tick frequency also depends on the larger system surrounding the neuromorphic accelerator including the application using it. Specifically, instructions being sent to the accelerator at runtime can affect the optimal tick frequency. For example, partially reconfiguring the loaded model during execution when sending CD frames after a TICK frame, can add significant latency, which would require the tick frequency to be temporarily slower. The execution time of instructions is also non-deterministic due to main memory access. This includes writing OUT frames to main memory, which must be done within a single tick period. Since SpikeHard is intended to be part of a larger system, congestion and other overheads within the system may be hard to anticipate at design time, and so the ability to dynamically modify the tick frequency is an important feature.

Altogether, tick generation must be tunable at runtime and synchronizable with the accelerator execution flow. This fine-grained control over the performance of the accelerator is made possible with TICK frames, which ensure that instructions are executed at the desired tick; noting that a TICK frame is only processed after all preceding frames have been executed. This also removes the need for a safety margin in the tick frequency, which would otherwise limit performance.

To alert when the tick frequency is too high, RANC uses error flags that are exposed as read-only top-level hardware signals. More precisely, if a tick occurs before all neuron-axon pairs are processed, or a spike is unable to arrive at its destination on time, a corresponding error flag is set. In SpikeHard, these flags are instead accessed from the Linux application via debug bits in the output-stream TERM frame. Accordingly, if a debug bit is set, this signals that the tick frequency was too high and must therefore be decreased before re-execution. In Section 6.2, we detail how we leveraged these debug bits to tune the tick frequency for our experiments.

The model-loading mechanism in SpikeHard is designed for reusability and efficiency as well. As mentioned previously, hard resetting a model (`rst_model`) involves disabling all the cores, where a core is re-enabled upon receiving a corresponding CD frame. In this way, model data for empty cores is not loaded, which would otherwise be time-consuming. Moreover, this mechanism could be leveraged to reduce the power consumption of empty cores by clock gating the cores or even power gating them. This mechanism is especially beneficial when the accelerator has over-provisioned the number of cores, which happens when the accelerator must be able to run a diverse set of models with different resource requirements. For now, a disabled core is blocked from generating spikes.

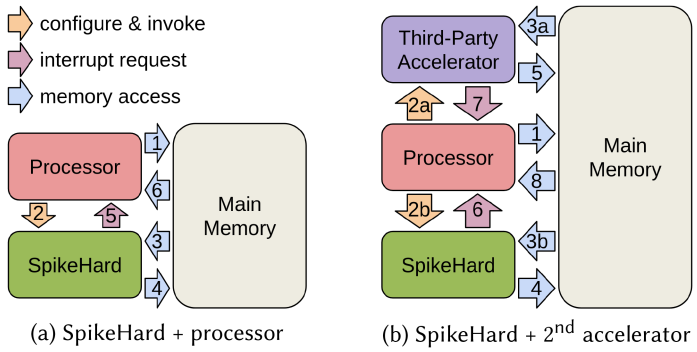

Fig. 9. Data-flows on the heterogeneous SoC.

Furthermore, the data protocol relies on features used in various neuromorphic processor designs. In particular, the tick control signal and neuron-axon structure are notably used by Loihi [13] and TrueNorth [2]. Hence, we expect that the hardware interface and data protocol could be extended and reused to interface with a variety of neuromorphic processors with little to no modification.

5 SOC INTEGRATION

5.1 SoC Design Platform

To integrate SpikeHard in an SoC we used ESP, an open-source platform for the agile design of heterogeneous SoCs and their prototyping on FPGA [27]. We designed SpikeHard with a standard interface, and by following the accelerator integration flow of ESP [19], we were able to integrate SpikeHard in a tile-based SoC with a multi-plane network-on-chip (NoC) as the main on-chip interconnect, together with a desired mix of provided accelerators, processor tile, memory tile (containing a channel to main memory), and an I/O tile that manages various peripherals.

ESP also provided us with a full software stack to load data into main memory and invoke SpikeHard from a Linux user-space application. As the main processor in the SoC, we selected the 64-bit CVA6 RISC-V processor [39].

5.2 SpikeHard and the Processor

Figure 9(a) shows the interaction between the processor and SpikeHard. Before invoking SpikeHard, the processor loads the base addresses into the read and write configuration registers (Figure 1(b)). Additionally, all the frames are loaded by the processor into the input-stream region in main memory, including the sequence of commands to execute and the data for computation.

Upon invoking SpikeHard, the accelerator reads the input stream from main memory, executes the sequence of instructions, completes the computation, and writes all the output frames into the output-stream region in main memory. The accelerator then sends an interrupt through the `irq` signal (Figure 1(b)) to notify the processor that the computation is done.

The processor then reads the output stream from main memory until it reaches a TERM frame. At that point, the processor holds all the computation results, and can decode the output spikes.

5.3 SpikeHard and Other Accelerators

To run complex applications in constrained environments, namely on the edge, we need devices based on heterogeneous many-accelerator SoCs as we require a variety of specialized hardware to efficiently run many different tasks [9, 20].

Neuromorphic hardware is one of the trends in the development of ASIC-based edge devices [20], and we are aware of applications that partially use neuromorphic computations [3, 7, 38]. In particular, Arsalan et al. [3] present an algorithm where the input data is pre-processed using a Fast Fourier Transform (FFT) before being fed into a neuromorphic model. Similarly, Chandarana et al. [7] describe a method that applies 2D-convolutions (CONV2D) to an input image so as to encode it into a sequence of spikes, which is then processed by a neuromorphic model. However, we did not notice any work that investigates neuromorphic hardware as a specialized hardware accelerator in a largely non-neuromorphic many-accelerator SoC. For this reason, we integrated SpikeHard in an SoC containing an FFT accelerator, as well as a CONV2D accelerator (Section 6.8).

Figure 9(b) describes a scenario where the application running on the processor takes advantage of the combined computation of SpikeHard and a third-party accelerator, where SpikeHard is not the execution bottleneck. Note that in Figure 9(b), arrows labeled with the same number followed by "a" or "b" denote actions that are performed concurrently with no defined order. Similar to the aforementioned scenario, we evaluated software applications that invoke SpikeHard and third-party accelerators in parallel on FPGA (Section 6.8).

6 EVALUATION

6.1 Experimental Setup

We evaluated various implementations of SpikeHard by integrating them into different SoC prototypes, each containing a 64-bit RISC-V CVA6 processor [39], an FFT accelerator, and a CONV2D accelerator. We deployed SoC prototypes on the Xilinx Virtex UltraScale+ VCU128 FPGA with a clock frequency of 75MHz, which is set by the ESP platform. We synthesized these prototypes with Xilinx Vivado, which reported the power dissipation and usage of FPGA resources: look-up tables (LUTs), flip-flops (FFs), block RAMs (BRAMs), and digital signal processors (DSPs). We do not discuss DSP usage as it remained constant across all implementations. Since SpikeHard is prototyped on FPGA, we estimated power dissipation based on the dynamic power reported by Xilinx Vivado for each of the synthesized designs. We used this power estimate together with the accelerator execution time on FPGA to calculate the energy used by each design. To test SpikeHard on FPGA, we implemented Linux applications in C that run on the CVA6 processor and invoke the accelerator via a device driver.

6.2 Tick Frequency Tuning

As detailed in Section 4.4, maximizing the tick frequency improves performance and in turn energy efficiency. Accordingly, for each model and hardware-architecture combination, we optimized the tick frequency using the following procedure.

Given a tick period and set of input spikes, we run the model on FPGA, or alternatively in a Verilog simulation consisting of the accelerator in isolation and an emulated DMA bus. The period is valid if no debug bits in the output-stream TERM frame (Section 4.4) are set. We first try with a small period, then increase it using a gradually larger step size until we obtain a valid period. The largest invalid period is the lower bound on the optimal tick period, and the smallest valid period is the upper bound. We then perform binary search within these bounds to find the optimal tick period. To account for real-world uncertainty, notably due to non-deterministic spike traffic, we add 10% slack to the optimal tick period.

In all of our experiments, the most time-consuming operation during each tick period was updating neuron states. The duration of this operation depends only on the core capacity (Section 4.4). Accordingly, we found that for a given core capacity, all evaluated models have the same optimal tick period. This means that communication, such as reading an IN frame from main memory and

sending the spikes in its payload to their corresponding cores, is not a performance bottleneck during computation. Thus, the only performance overhead incurred by the runtime configurability of SpikeHard is the model-loading overhead, which can be amortized away (Section 6.3). However, communication may become a performance bottleneck for larger models with smaller cores. This is because larger models produce more spike traffic, and smaller cores can run at higher tick frequencies. Furthermore, applications that interact with a mixture of hardware accelerators can create fluctuating congestion on the SoC, which is why dynamic tick frequency tuning is essential.

6.3 SNN Models

The models loaded on SpikeHard classify images from the MNIST dataset [15] and perform signed VMM using the algorithm described in [17]. For MNIST, we classified $batch\_size = 200$ images per run in a pipelined fashion, where a new image was provided as input every tick. In contrast, VMM uses feedback connections to iteratively converge to an approximate solution. Consequently, only a single vector-matrix product can be computed per run, that is, $batch\_size = 1$. The performance of a model is its throughput, which is defined as the $batch\_size$ divided by the average latency of a run. This includes the model-loading overhead at the start of every run. In practice, if the same model is used across several accelerator invocations, it only needs to be loaded once so that the model-loading overhead becomes negligible. Furthermore, if a variety of models are being frequently used, multiple instances of SpikeHard can be integrated within a single SoC, where each model is loaded to a distinct instance and reused across invocations.

We tested models with various combinations of neurons (N) and axons (A) per core, denoted as $A \times N$. We started with two models generated by RANC: 64×64 VMM (VMM-O) and 256×256 MNIST (MNIST-O). We optimized the tick frequency for each model using the method described in Section 6.2, resulting in performance and energy-efficiency gains of 1.9× for MNIST-O and 27× for VMM-O over the original RANC models while preserving functional correctness. *Unless stated otherwise, the results reported in this section are relative to the corresponding RANC-generated model with optimized tick frequency.*

Throughout evaluation, we used the same bit precision as in TrueNorth [2], namely signed 9-bit values for weights and neuron potentials, which is also the default used in RANC [26]. In practice, to reduce resource utilization, the bit precision would be minimized within the requirements of the models to run.

6.4 MNIST Performance and Energy-Efficiency

By applying model restructuring on MNIST-O without changing $A \times N$, we decreased the number of cores used from 11 down to 5. Accordingly, FPGA resource usage was reduced by 2×, yielding the same improvement in energy efficiency. Although core capacity was unchanged, performance improved by 0.2% thanks to a 2.4× lower model-loading overhead as a result of utilizing fewer cores. However, computation time excluding overheads remained the same since the tick frequency was unaffected.

We were unable to restructure MNIST-O to a smaller power-of-two core capacity ($A, N < 256$) as MNIST-O has an MCC (Section 3.2) with 256 axons and 250 neurons. Restructuring MNIST-O into architectures satisfying $A, N > 256$ yielded a negative impact on resource utilization without an improvement in performance. Ultimately, MNIST-O restructured to the same core capacity (256×256) gave the best performance and energy-efficiency.

6.5 Design-Space Exploration for a Fixed-Size VMM

We restructured VMM-O to all core capacities satisfying $A, N \in \{32, 64, 128, 256, 512\}$. We were able to reduce the number of axons and neurons per core down to 32 since none of the MCCs have more

Table 3. Original VMM-O Implementation Compared to its Pareto-optimal Implementations

Name	$A \times N$	#Cores	Pareto Optimal			Resource Usage			Latency (ms)	Tick Freq. (kHz)	Power (mW)
			LUT	FF	BRAM	LUT	FF	BRAM			
Original	64×64	17			N/A*	53314	30296	0	12.69	15.49	211
A	32×32	15	✓	✓	N/A*	29483	18540	0	3.82	60.53	112
B	64×64	8	✓	✓	N/A*	26669	15286	0	12.61	15.49	92
C	32×128	15			✓	28406	16456	60	12.75	15.49	135
D	128×64	5	✓		N/A*	25502	15807	0	23.35	8.05	80
E	64×128	8	✓	✓	✓	25240	13895	40	23.98	8.05	113
F	128×128	3	✓	✓	✓	16334	9539	21	48.93	3.82	61
G	128×256	3	✓			16320	9544	21	91.12	2.04	60

*Implementation does not use any BRAMs.

than 32 axons or neurons. The DSE of VMM-O is summarized in Figure 10 with Pareto-optimal points for each FPGA resource. These points are also listed in Table 3. Several implementations did not use BRAMs, which were replaced by FFs. These implementations were not considered when determining the Pareto-optimal points for BRAM utilization.

Pareto-optimal implementations with smaller core capacity reflected more utilization of FFs and LUTs. This is due to the fact that in these implementations, FFs were preferred over BRAMs, and more cores were needed to fit the model. In some cases, the use of BRAMs instead of FFs and LUTs improved power consumption. In fact, implementation *F* and *G*, which replace FFs and LUTs with BRAMs, consume between 1.3× and 1.9× less power than Pareto-optimal implementations that do not use BRAMs. Furthermore, implementation *G*, which has the largest core capacity among the Pareto-optimal implementations, has the lowest power consumption. That being said, cores of larger capacity require a lower tick frequency, resulting in higher latency and thus worse performance. *In cases where stringent power constraints are prioritized above overall latency, or when there is a limited amount of LUTs or FFs available, the use of cores of large capacity should be considered.*

As shown in Figure 10(d), implementation *A* (smallest core capacity) gave the best results, with 3.3× better performance and 6.3× higher energy efficiency over VMM-O. In total, the improvements in performance and energy-efficiency over the original RANC model without optimized tick frequency were 89× and 170×, respectively. Moreover, implementation *B* has the same core capacity as VMM-O but was restructured into 2.1× fewer cores. As a result, energy efficiency improved by 2.3× mostly thanks to a less power-consuming design. Additionally, the latency of implementation *B* with respect to the original design was reduced as the model-loading overhead was halved. However, the reduction was only by 0.6% as model loading originally constituted only 1.6% of the overall latency, hence, it had no significant impact on energy efficiency. *Altogether, model restructuring to the same core capacity significantly improves energy efficiency, and restructuring into smaller core capacities yield higher gains in energy-efficiency and performance.*

6.6 Design-Space Exploration for Varying-Size VMM

In neuromorphic models that compute a VMM, the connections between neurons and axons as well as the model size are highly dependent on the matrix shape. Accordingly, to determine whether model restructuring is effective regardless of the model structure, we tested our optimizer on models with different matrix shapes. As VMM-O was generated for a matrix with 6 rows and 6 columns, we performed DSE on VMM models trained with larger matrices. We first chose two matrices, one with 15 rows and 15 columns and the other with 16 rows and 64 columns. We named the generated models as VMM-15 for the 15 by 15 matrix, and VMM-64 for the 16 by 64 matrix. The results for VMM-15 and VMM-64 are summarized in Figures 11 and 12, respectively.

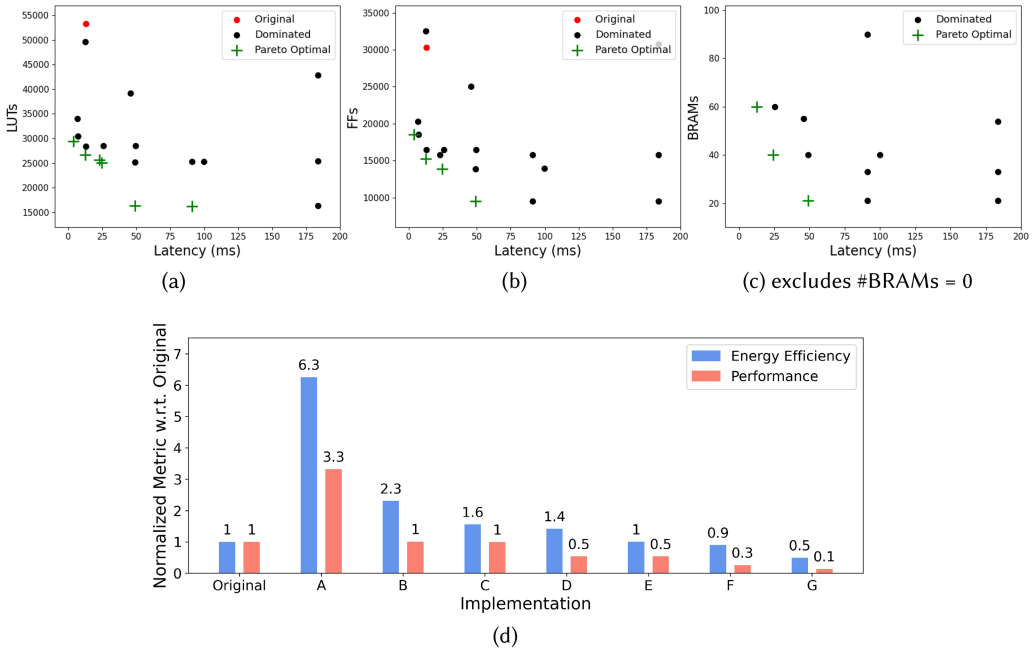

Fig. 10. Design-space exploration of hardware implementations originating from VMM-O.

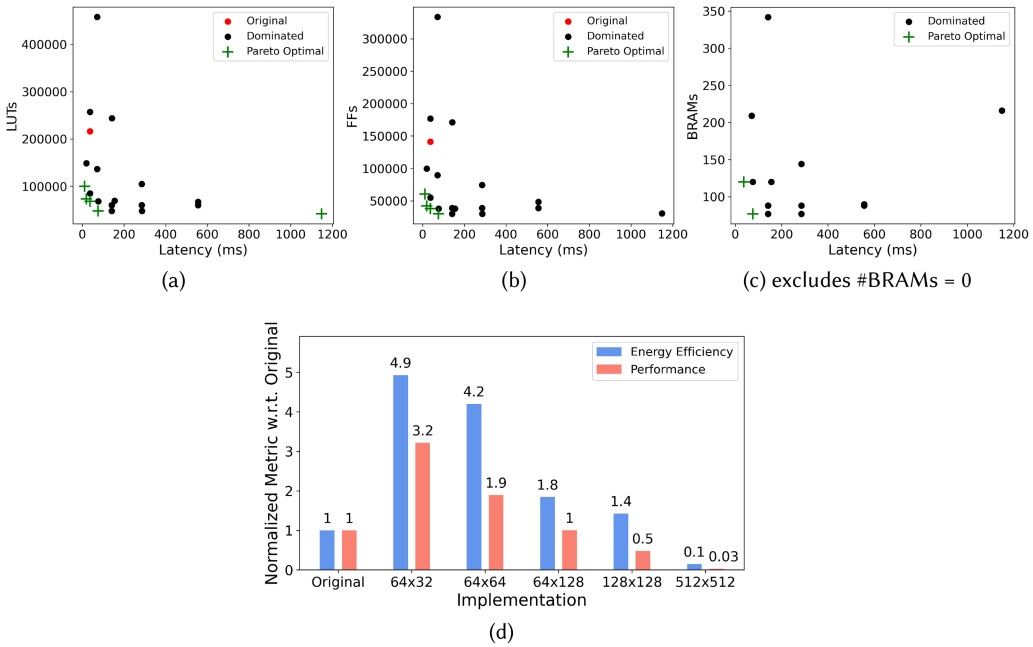

Fig. 11. Design-space exploration of hardware implementations originating from VMM-15.

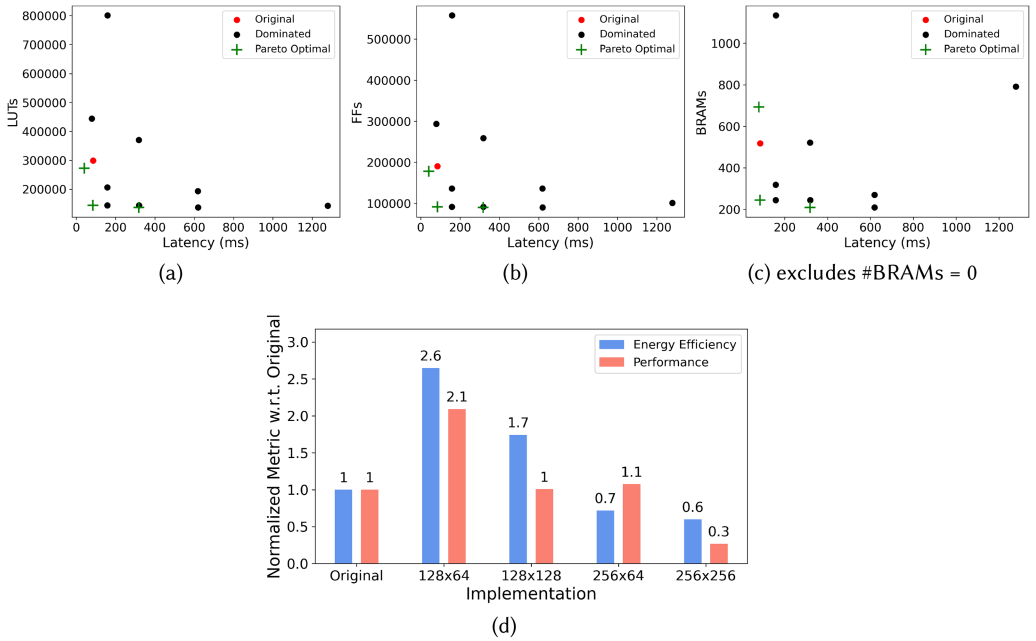

Fig. 12. Design-space exploration of hardware implementations originating from VMM-64.

VMM-15 was generated with core capacity 128×64. We restructured the model to a variety of power-of-two core capacities, with the smallest one being 64×32 so as to fit the biggest MCC, which has 60 axons and 28 neurons. We gathered all the Pareto-optimal implementations and compared their performance and energy-efficiency normalized to that of the original VMM-15 implementation. This comparison is illustrated in Figure 11(d), where the x-axis denotes the core capacity of the implementation. The implementation with the smallest core capacity (64×32) achieved the best gain in performance and energy-efficiency of 3.2× and 4.9×, respectively.

We performed an analogous DSE for VMM-64, which unlike VMM-15, was generated with core capacity 128×128 and could only be restructured down to 128×64 as the biggest MCC of VMM-64 has 92 axons and 64 neurons. After model restructuring and DSE, the best implementation for VMM-64 is the one with the smallest core capacity (128×64), which improved performance by 2.1× and energy efficiency by 2.6×.

For all the models we have evaluated, namely MNIST-O, VMM-O, VMM-15, and VMM-64, restructuring to the smallest core capacity possible gave the best performance and energy-efficiency. Typically, as we increase the size of the matrix in the VMM model, the size of the largest MCC increases as well, which directly affects the minimum core capacity. A bigger matrix also results in a larger model with more MCCs.

Hence, we generated additional VMM models with different matrix shapes and inherently distinct maximum MCC sizes, which ranged between 264 (22 axons · 12 neurons) and 10112 (158 axons · 64 neurons). We restructured each model to its corresponding minimum power-of-two core capacity. We compared the performance and energy-efficiency of the restructured architectures to the ones generated by RANC, with the results being described in Figure 13.

In particular, restructuring to the smallest core capacity improved performance by $1.90 - 3.73\times$ and energy efficiency by $1.97 - 6.96\times$. Overall, the greater the maximum MCC size, the lower the performance, which in turn results in lower energy efficiency. The main reason is that bigger MCCs

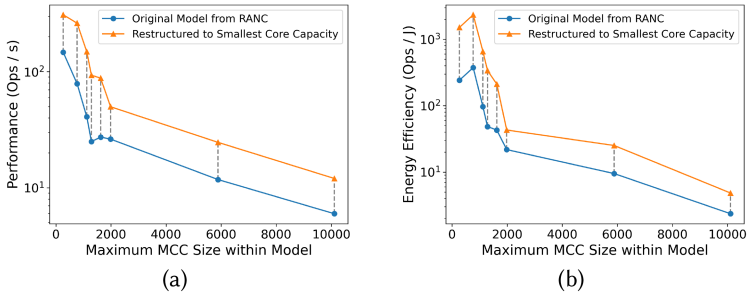

Fig. 13. Performance and energy-efficiency for VMM models with different matrix shapes.

necessitate larger core capacities, which operate at slower tick frequencies. Another reason is that VMM models with bigger matrices typically require more tick periods to provide results, which also worsens performance.

In Figure 13(b), the relative positioning of the two models with the smallest maximum MCC size is anomalous. Specifically, energy efficiency is better for the one with bigger MCCs, but usually the opposite is true. These two models correspond to an 8 by 8 matrix (VMM-8) and a 6 by 6 matrix (VMM-O). The largest MCC size for VMM-8 is 264 (22 axons · 12 neurons) and for VMM-O it is 768 (32 axons · 24 neurons). Therefore, although VMM-8 has a bigger matrix, it has a smaller maximum MCC size. As a result, VMM-8 was restructured to a core capacity of 32×16, which is smaller than the capacity used by VMM-O, namely 32×32. However, VMM-8 has 1.6× more axons than VMM-O, and so VMM-8 requires more resources than VMM-O, which is why VMM-8 is less energy efficient.

Typically, energy-efficiency and performance worsen as MCCs grow in size, but model restructuring helps to minimize this effect for varying-size VMM computations.

6.7 Resource Overhead of Programmability

While SpikeHard is a runtime-programmable neuromorphic accelerator, the architecture of the inner neuromorphic processor is largely the same as in RANC. The controller and DMA handler at the interface of SpikeHard (Figure 1(b)) represent the main components that enable programmability and differentiate it from the non-programmable RANC. The SpikeHard interface replaces the basic AXI interface and immutable tick generator used in RANC, and requires additional resources for dynamic model-loading that scale with the core capacity and number of cores.

To better understand the overhead of adding programmability to the neuromorphic processor, we observed the resource utilization of the SpikeHard interface in proportion to that of the entire accelerator. For all VMM implementations, namely VMM-O, VMM-15, and VMM-64, the interface does not use any BRAMs and requires only 11 DSPs, which is also the total number of DSPs used by the entire accelerator. The results for LUTs and FFs are summarized in Figure 14.

Typically, the larger the cores are and the greater their number, so are the total utilizations of LUTs and FFs. However, the amount of resources used for the interface scales logarithmically while for the rest of the accelerator, namely the neuromorphic processor, the amount of resources scales linearly. When the accelerator is small, the proportion of resources used by the interface is around 30%, but the larger the accelerator, the smaller this proportion becomes, which means less overhead is incurred by the programmability of SpikeHard. Overall, for an accelerator of average size, the proportion of LUTs and FFs used for programmability out of the total used by SpikeHard is only ~8% and ~5%, respectively.

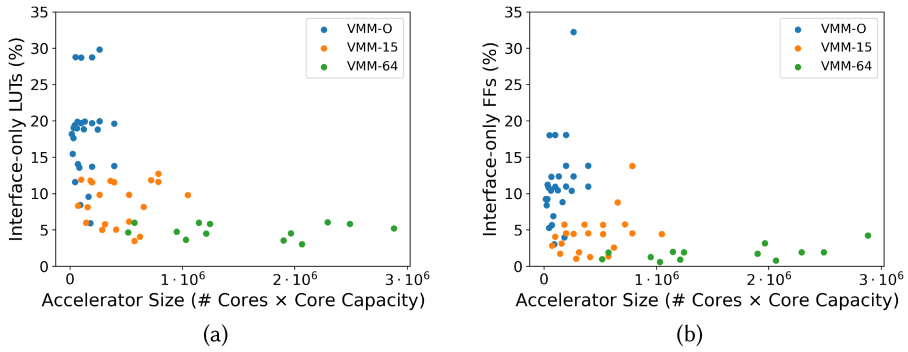

Fig. 14. Proportion of resources used by the accelerator interface of SpikeHard.

6.8 Parallel Execution of Accelerators in a Many-Accelerator SoC

We evaluated SpikeHard as an accelerator in a many-accelerator SoC to determine whether SpikeHard has equivalent performance to accelerators that execute popular non-neuromorphic kernels, namely FFT and CONV2D, which are both used in practice alongside neuromorphic kernels [3, 7]. To this end, we created a software application where SpikeHard, FFT, and CONV2D accelerators are invoked in parallel, each taking as input ~32 kB of data.

We first tested VMM-O (Section 6.3) without applying model restructuring as the baseline model for SpikeHard. In this test, SpikeHard, FFT, and CONV2D took an average of 12.9 ms, 5.1 ms, and 9.7 ms, respectively. In other words, SpikeHard with non-restructured VMM-O produced a slowdown of 1.3 – 2.5× with respect to the other accelerators.

We then tested implementation *A* of SpikeHard from our DSE as it showed the best performance and energy-efficiency among the Pareto-optimal implementations of VMM-O (Table 3). With this implementation, SpikeHard performed the same computation in only 5.0 ms, which is faster than for FFT and CONV2D that took the same amount of time as in the previous test. Thus, by using model restructuring and DSE, we were able to speed up SpikeHard to the point that it is not a performance bottleneck on the heterogeneous many-accelerator SoC, and can be safely used to execute neuromorphic workloads in parallel to other kinds of workloads in the same application.

We have demonstrated that SpikeHard is a neuromorphic accelerator with equivalent performance to non-neuromorphic accelerators, and can be seamlessly used with other accelerators in a heterogeneous many-accelerator SoC.

7 RELATED WORK

A variety of embedded applications perform both neuromorphic and non-neuromorphic tasks [3, 7, 38]. Typically, the input data is pre-processed using a non-neuromorphic operation, such as an FFT [3], a 2D convolution [7], or a Singular Value Decomposition (SVD) [38], with the output of this being fed into a neuromorphic model.

Many algorithms have already been proposed for mapping an SNN to neuromorphic hardware, most of which minimize spike traffic on the neuromorphic core interconnect [2, 4, 33, 36]. For instance, Titirsha et al. [36] iteratively try to find a randomly-generated mapping that minimizes the energy consumption of spike communication. Balaji et al. [4] run the SNN in simulation and analyze the spike traffic to find a mapping that minimizes spike communication latency and energy consumption. In contrast, our model restructuring method, which can be used for mapping SNNs to hardware, minimizes resource utilization by leveraging a fine-grained partitioning strategy for SNNs (MCC analysis), and has been shown to improve energy efficiency.

Unlike the TrueNorth [2] and Loihi 2 [23] neuromorphic processors, RANC [26] is a neuromorphic architecture that is highly reconfigurable at design time, available as an open-source project, and deployable on FPGA. To shorten the path of prototyping a heterogeneous SoC with neuromorphic hardware on FPGA, we used RANC as the baseline design for SpikeHard. Unlike RANC, SpikeHard can be reconfigured at runtime and has an interface that allows it to be seamlessly integrated in heterogeneous many-accelerator SoCs.

SoCs equipped with neuromorphic hardware are available, but they rarely consider the integration of non-neuromorphic hardware accelerators. For instance, Akida [6] is an SoC containing an Arm processor that controls the neuromorphic hardware, but does not support the on-chip integration of other hardware accelerators. Similarly, the Loihi SoC [13] consists of a neuromorphic processor and an Intel x86 processor, which merely manages the neuromorphic processor. Loihi is typically deployed as part of a Nahuku expansion board [21, 22], which contains up to 32 interconnected Loihi chips that all interface with the same on-board Arria 10 FPGA. This FPGA is used to interact with the surroundings, notably via sensors and actuators. Additionally, the FPGA is controlled by an off-chip processor (distinct from the aforementioned x86 processor) from which the high-level software application is executed, such as compiling the neuromorphic model and visualizing results.

By contrast, SpikeHard is integrated as part of a standalone SoC consisting of not only a general-purpose 64-bit CVA6 RISC-V processor [39], but also non-neuromorphic accelerators, where the processor orchestrates the execution of the accelerators via a common standard interface. In this way, the neuromorphic hardware interfaces directly with an on-chip processor that runs the software applications. This results in fewer levels of indirection and less off-chip communication, potentially yielding lower latency and higher efficiency. Moreover, designing an SoC with all the components integrated together on the same chip allows for a reduction in communication overhead.

Other neuromorphic chips, such as Caspian [29] and DYNAP-CNN [34], implement pure neuromorphic processing, without the realization of heterogeneous SoCs. Yet, embedded applications that use neuromorphic and non-neuromorphic tasks create a need for heterogeneous SoCs that can accelerate all of these tasks. Thus, we introduced SoCs where SpikeHard and other accelerators can be invoked from the same software application. *Compared to existing solutions, our approach facilitates development and deployment of neuromorphic hardware that can be efficiently integrated in largely non-neuromorphic and highly heterogeneous SoCs.*

8 CONCLUSION

We designed SpikeHard to advance the research and exploration of neuromorphic hardware as a main component in future heterogeneous SoCs. SpikeHard was designed through an efficiency-driven methodology that prioritizes heterogeneity and scalability, as it simplifies the integration of neuromorphic architectures in many-accelerator SoCs. SpikeHard is a first step in enabling the access of embedded applications to neuromorphic computations among a multitude of inter-dependent non-neuromorphic tasks. We have released the contributions of this work in the public domain.[1]

REFERENCES

[1] Tobias Achterberg. 2009. SCIP: Solving constraint integer programs. *Mathematical Programming Computation* 1, 1 (2009), 1–41.

[2] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, Brian Taba, Michael Beakes, Bernard Brezzo, Jente B. Kuang, Rajit Manohar,

[1] SpikeHard public code repository: https://github.com/sld-columbia/spikehard

William P. Risk, Bryan Jackson, and Dharmendra S. Modha. 2015. TrueNorth: Design and tool flow of a 65 mW 1 Million neuron programmable neurosynaptic chip. *IEEE TCAD* 34, 10 (2015), 1537–1557.
[3] Muhammad Arsalan, Avik Santra, and Vadim Issakov. 2022. RadarSNN: A resource efficient gesture sensing system based on mm-Wave radar. *T-MTT* 70, 4 (2022), 2451–2461.
[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. 2019. Mapping spiking neural networks to neuromorphic hardware. *TVLSI* 28, 1 (2019), 76–86.
[5] Morris H. Baslow. 2009. The languages of neurons: An analysis of coding mechanisms by which neurons communicate, learn and store information. *Entropy* 11, 4 (2009), 782–797.
[6] BrainChip. 2023. *Akida*. Retrieved July 2, 2023 from https://brainchip.com/products/
[7] Peyton Chandarana, Junlin Ou, and Ramtin Zand. 2021. An adaptive sampling and edge detection approach for encoding static images for spiking neural networks. In *Proc. of IGSC*. 1–8.
[8] Yanjiao Chen, Baolin Zheng, Zihan Zhang, Qian Wang, Chao Shen, and Qian Zhang. 2020. Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions. *CSUR* 53, 4, Article 84 (2020), 37 pages.
[9] Kuan-Lin Chiu, Guy Eichler, Biruk Seyoum, and Luca Carloni. 2023. EigenEdge: Real-time software execution at the edge with RISC-V and hardware accelerators. In *Proc. of CPS-IoT Week*. 209–214.
[10] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. 2012. Architecture support for accelerator-rich CMPs. In *Proc. of DAC*. 843–849.
[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill. Section 22.5.
[12] Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. 2015. An analysis of accelerator coupling in heterogeneous architectures. In *Proc. of DAC*. Article 202, 6 pages.
[13] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steven McCoy, Arnab Paul, Jonathan Tse, Guruguhanathan Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang. 2018. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 1 (2018), 82–99.
[14] Mike Davies, Andreas Wild, Garrick Orchard, Yulia Sandamirskaya, Gabriel A. Fonseca Guerra, Prasad Joshi, Philipp Plank, and Sumedh R. Risbud. 2021. Advancing neuromorphic computing with loihi: A survey of results and outlook. *Proc. IEEE* 109, 5 (2021), 911–934.
[15] Li Deng. 2012. The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.
[16] Guy Eichler, Luca Piccolboni, Davide Giri, and Luca P. Carloni. 2021. MasterMind: Many-accelerator SoC architecture for real-time brain-computer interfaces. In *Proc. of ICCD*. 101–108.
[17] Kaitlin L. Fair, Daniel R. Mendat, Andreas G. Andreou, Christopher J. Rozell, Justin Romberg, and David V. Anderson. 2019. Sparse coding using the locally competitive algorithm on the TrueNorth neurosynaptic system. *Frontiers in Neuroscience* 13 (2019), 754.
[18] Wulfram Gerstner. 1998. *Spiking Neurons*. MIT Press, 3–5.
[19] Davide Giri, Kuan-Lin Chiu, Guy Eichler, Paolo Mantovani, and Luca P. Carloni. 2021. Accelerator integration for open-source SoC design. *IEEE Micro* 41, 4 (2021), 8–14.
[20] Cong Hao et al. 2021. Enabling design methodologies and future trends for edge AI: Specialization and codesign. *IEEE Design & Test* 38, 4 (2021), 7–26.
[21] Intel. 2019. *Loihi Deep Dive*. Retrieved July 2, 2023 from https://niceworkshop.org/wp-content/uploads/2019/04/NICE-2019-Day-4c_Loihi-Overview.pdf
[22] Intel. 2021. *Loihi*. Retrieved July 2, 2023 from https://en.wikichip.org/wiki/intel/loihi
[23] Intel. 2021. *Taking Neuromorphic Computing to the Next Level with Loihi 2*. Retrieved July 2, 2023 from https://download.intel.com/newsroom/2021/new-technologies/neuromorphic-computing-loihi-2-brief.pdf
[24] Minseon Kang, Yongseok Lee, and Moonju Park. 2020. Energy efficiency of machine learning in embedded systems using neuromorphic hardware. *Electronics* 9, 7 (2020), 1069.
[25] Sixu Li, Zhaomin Zhang, Ruixin Mao, Jianbiao Xiao, Liang Chang, and Jun Zhou. 2021. A fast and energy-efficient SNN processor with adaptive clock/event-driven computation scheme and online learning. *IEEE TCAS-I* 68, 4 (2021), 1543–1552.
[26] Joshua Mack, Ruben Purdy, Kris Rockowitz, Michael Inouye, Edward Richter, Spencer Valancius, Nirmal Kumbhare, Md Sahil Hassan, Kaitlin Fair, John Mixter, and Ali Akoglu. 2021. RANC: Reconfigurable architecture for neuromorphic computing. *IEEE TCAD* 40, 11 (2021), 2265–2278.
[27] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. 2020. Agile SoC development with open ESP. In *Proc. of ICCAD*. 1–9.

[28] Silvano Martello and Paolo Toth. 1990. *Knapsack Problems: Algorithms and Computer Implementations*. Chapter 8.
[29] J. Parker Mitchell, Catherine D. Schuman, Robert M. Patton, and Thomas E. Potok. 2020. Caspian: A neuromorphic development platform. In *Proc. of NICE*. 1–6.
[30] Eric Nichols, Liam J. McDaid, and Nazmul Siddique. 2012. Biologically inspired SNN for robot control. *IEEE Transactions on Cybernetics* 43, 1 (2012), 115–128.
[31] Robert Patton, Catherine Schuman, Shruti Kulkarni, Maryam Parsa, J. Parker Mitchell, N. Quentin Haas, Christopher Stahl, Spencer Paulissen, Prasanna Date, Thomas Potok, and Shay Snyder. 2021. Neuromorphic computing for autonomous racing. In *Proc. of ICONS*. Article 23, 5 pages.
[32] Biswa Sengupta and Martin B. Stemmler. 2014. Power consumption during neuronal computation. In *Proc. of IEEE*, Vol. 102. 738–750.
[33] 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 *Proc. of ICCAD*. 1–9.
[34] SynSense. 2023. *DYNAP-CNN*. Retrieved July 2, 2023 from https://www.synsense.ai/products/dynap-cnn/
[35] Guangzhi Tang and Konstantinos P. Michmizos. 2018. Gridbot: An autonomous robot controlled by a Spiking Neural Network mimicking the brain's navigational system. In *Proc. of ICONS*. Article 4, 8 pages.
[36] Twisha Titirsha, Shihao Song, Adarsha Balaji, and Anup Das. 2021. On the role of system software in energy management of neuromorphic computing. In *Proc. of CF*. 124–132.
[37] Hang Yin, John Boaz Lee, Xiangnan Kong, Thomas Hartvigsen, and Sihong Xie. 2021. Energy-efficient models for high-dimensional spike train classification using sparse spiking neural networks. In *Proc. of KDD*. 2017–2025.
[38] Zheqi Yu. 2022. *Low-power neuromorphic sensor fusion for elderly care*. Ph. D. Dissertation. University of Glasgow.
[39] Florian Zaruba and Luca Benini. 2019. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-GHz 64-Bit RISC-V core in 22-nm FDSOI technology. *ITVL* 27, 11 (2019), 2629–2640.

Received 23 March 2023; revised 2 June 2023; accepted 30 June 2023