A Spiking Neural Algorithm for Network Flow

A pipeline from theory to practice for neuromorphic computing

Course code: SOW-MKI92

MSc thesis Artificial Intelligence

ABDULLAHI ALI

s4420241

5 June 2019

Supervisor:

Johan Kwisthout

Second reader:

Iris van Rooij

Radboud University

Abstract

It is not clear what the potential is of neuromorphic hardware beyond machine learning and neuroscience. In this project, a problem is investigated that is inherently difficult to fully implement in neuromorphic hardware by introducing a new machine model in which a conventional Turing machine and neuromorphic oracle work together to solve such types of problems. A lattice of complexity classes is introduced: $\mathsf{C}^{\mathsf{SNN}(R_S)}$, in which a neuromorphic oracle is consulted using only resources at most R_S. We show that the P-complete MAX NETWORK FLOW problem is in $\mathsf{L}^{\mathsf{SNN}(\mathcal{O}(n),\mathcal{O}(n),\mathcal{O}(n))}$ for graphs with n edges. A modified variant of this algorithm is implemented on the Intel Loihi chip; a neuromorphic manycore processor developed by Intel Labs. We show that by off-loading the search for augmenting paths to the neuromorphic processor we can get energy efficiency gains, while not sacrificing runtime resources. This result demonstrates how P-complete problems can be mapped on neuromorphic architectures in a theoretically and potentially practically efficient manner.

1 Introduction

Neuromorphic computing has been one of the proposed novel architectures to replace the von Neumann architecture that has dominated computing for the last 70 years [18]. These systems consist of low power, intrinsically parallel architectures of simple spiking processing units. In recent years numerous neuromorphic hardware architectures have emerged with different architectural design choices [1, 8, 21, 14].

It is not exactly clear what the capabilities of these neuromorphic architectures are, but several properties of neuromorphic hardware and application areas have been identified in which neuromorphic solutions might yield efficiency gains compared to conventional hardware architectures such as CPUs and GPUs [1, 8, 14, 21]. These applications are typically inherently event-based, easy to parallelize, are limited in terms of their energy budget and can be implemented on a sparse communication architecture where processors can communicate with small packets of information.

A potential major application area of these neuromorphic architectures is machine learning. This is motivated by the results deep neural networks have achieved in machine learning [16], where these loosely brain-inspired algorithms have dramatically redefined machine learning and pattern recognition. However, deep neural networks tend to consume a significant amount of energy. This energy bottleneck is one of the major reasons why these deep networks have not been successfully employed in embedded AI applications such as robotics. Neuromorphic processors, on the other hand, could potentially solve this bottleneck and fuel a new leap forward in brain-inspired computing solutions for AI.

There are several other areas that could greatly benefit from energy efficiency. One of these applications is numerical algorithms in scientific computing [22]. Traditionally, neural networks are trained by automatically modifying their connection weights until a satisfactory performance is achieved. Despite its success in machine learning, this approach is not suitable for scientific computing or similar areas since it may require many training iterations and does not produce precise results.

Alternatively, we can abandon learning methods and design the networks by hand. One way to do this is to carefully construct a network of (non-stochastic) spiking neurons to encode information in the spiking patterns or spike-timing. One can, for example, introduce a synaptic delay to encode distance or intro-

duce a spiking clock mechanism to encode values between the spike-time difference of a readout neuron and a clock.

Efforts have been undergone on designing neural algorithms for primitive operations [24, 22] and relatively straightforward computational problems [17, 13], but it is not clear how these methods can be scaled up to more complex algorithms. In this work, we build upon the algorithm design approach advocated by [24, 22] and propose a systematic way to analyse and expand the potential application space of neuromorphic hardware beyond machine learning. In this light, we look at a problem of non-trivial complexity: the maximum flow problem.

The input of the maximum flow problem consists of a weighted graph, where each weight denotes the capacity of a certain edge. We have two special nodes: a source and a sink. The source is the starting point of the graph and *produces* flow and the sink is the terminal point of the graph and *consumes* flow. The objective is to push as much flow as possible from source to sink while respecting the capacity constraints for each edge. The canonical method to solve this problem is the Ford-Fulkerson method [10]. In this method, one repeatedly searches for augmenting paths. These are simple paths from source to sink through which we can still push flow (i.e. no edge on the path has reached full capacity). If such a path is found, we determine the minimum capacity edge on this path and increment the flow through each edge in this path with this minimum capacity. This process is repeated until all augmenting paths have been found. In figure 1 you can see an example of such a flow network.

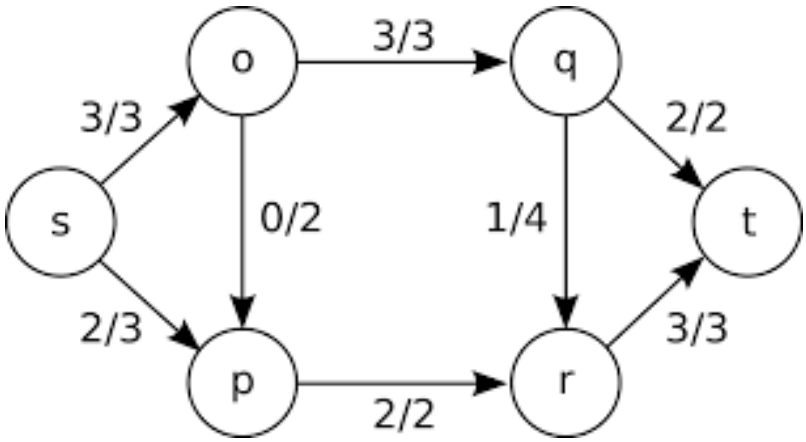

Figure 1: *Example of a flow network with source 's' and sink 't'. Note that each has two numbers f/c associated to it. 'f' indicates the current flow through the edge and 'c' indicates the total capacity of the edge.*

The maximum flow problem arises in many areas, such as logistics, computer networking, operations research and bioinformatics. With the huge increase in data in these application areas, the flow networks will similarly increase in size, demanding a need for faster and parallel algorithms.

Unfortunately, network flow algorithms are difficult to parallelize. The vast majority of network flow algorithms are implementations of the Ford-Fulkerson method. Augmenting paths have to be found from source to sink and flow has to be pushed through these paths, requiring fine-grained locking of the edges and nodes on the path which introduces expensive overhead.

Several parallel implementations for finding the maximum flow in a network exist [4, 6, 12], but they often are carefully optimised against contemporary computer architecture or do not offer significant performance advantages over optimised serialised solutions. As a matter of fact, theoretical results show that Network flow is a P-complete problem [11]. This means that it probably cannot be efficiently solved on a parallel computer[1]. It is therefore likely that it cannot be fully implemented in neuromorphic hardware. Another corollary is that P-complete problems also have the property that they likely cannot be solved with a guaranteed logarithmic space bound (henceforth denoted as logspace constraint). The class of polynomial-time solvable problems that respect the logspace constraint is called L and it is likely that no

[1] A decision problem D belongs to the class NC (class of decision problems that can be decided in polylogarithmic time on a parallel computer) if there exist constants c and k such that D can be decided in $log(n^c)$ time using only k processors [5].

P-complete problem is in this class.

An interesting direction of investigation is whether we can achieve this logspace constraint by utilizing aspects of conventional and neuromorphic hardware. In this light, we introduce a new machine model in which a conventional computing device can consult a neuromorphic oracle to offload certain computations. Under this machine model, we introduce a lattice of complexity classes: $\mathsf{C}^{\mathsf{SNN}(R_S)}$, in which the conventional computing device can consult a neuromorphic oracle using only resources at most R_S, where $R_S = (\mathrm{TIME}, \mathrm{SPACE}, \mathrm{ENERGY})$ are the time, space and energy constraints of the neuromorphic oracle. We show that the P-complete MAX NETWORK FLOW problem is in $\mathsf{L}^{\mathsf{SNN}(\mathcal{O}(n),\mathcal{O}(n),\mathcal{O}(n))}$ for graphs with n edges. We can further refine that to $\mathsf{L}^{\mathsf{SNN}(\mathcal{O}(l(s,t)),\mathcal{O}(n),\mathcal{O}(n))}$ where $l(s,t) \leq n-1$ denotes the maximum path length between source and sink.

In addition to that, we experimentally demonstrate that off-loading the search for augmenting paths to a neuromorphic processor could potentially yield energy efficiency gains, while not sacrificing runtime complexity.

The remainder of this thesis is organised as follows: In section 2 we will give an introduction to neuromorphic computing in general, and neuromorphic complexity analysis and neural algorithm design in particular. This will provide a grounding for the reader to understand the subsequent sections.

Following this introduction, we will formalize our machine and neural model in section 3 and give a formal definition of the MAX NETWORK FLOW problem.

In section 4 we will describe our algorithm and follow up with a complexity analysis of this algorithm in section 5.

In section 6 we will give a description of the methodology for our empirical validation of the theoretical analyses and provide the results from our experiments. In section 7 we discuss these results and evaluate the proposed pipeline and in section 8 we will end with some concluding remarks.

The goal of this project is threefold, (1) to use the tools and methods from computational complexity theory to come up with a systematic way of evaluating the feasibility of implementing computational

problems in neuromorphic hardware, (2) to demonstrate that we can potentially expand the application space of neuromorphic hardware by suggesting an alternative model of computation, (3) inform neuromorphic hardware designers about potential architectural design limitations in expanding the application space to the class of problems under scrutiny in this project. By satisfying these goals, we hope to demonstrate the potential of a top-down analytical evaluation pipeline in demystifying the application space of neuromorphic hardware.

2 Preliminaries

2.1 Neuromorphic Hardware

Neuromorphic computing has been one of the proposed novel architectures to replace the von Neumann architecture that has dominated computing for the last 70 years [18]. These systems consist of low power, intrinsically parallel architectures of simple spiking processing units.

In recent years numerous neuromorphic hardware architectures have emerged with different architectural design choices [1, 8, 21, 14]. These distinctions include digital vs. mixed-signal approaches, the underlying neural model (simple neurons vs. more complex neuronal models), the scale of the systems in terms of the size of the networks that can be simulated, the number of features they offer, and whether the operation speed is accelerated or real-time.

These design decisions are mostly motivated by the type of applications the original designer had in mind. For example, the BrainScaleS system [21] is an accelerated mixed-signal system that operates at 10.000x biological real-time. This architecture is appropriate for simulating realistic biological processes over multiple time-scales (from the millisecond scale to years).

The SpiNNaker system, on the other hand, [14] has a digital architecture and runs at biological real-time. Due to its scalable digital architecture, it can run very large neural simulations and its real-time operation opens up opportunities for robotics applications.

A different approach is to focus on flexibility. An example of such an architecture is the Loihi chip [8], a neuromorphic chip developed by Intel Labs. The Loihi chip has a digital architecture inspired by a sim-

ple computational model of neural information processing: a network of leaky integrate-and-fire (LIF) neurons. In addition to that, it offers a wide array of features that enable users to build more complex neuronal models.

In this project, we aim to complement these more bottom-up approaches by a strictly top-down approach, in which we analyse the resource demands of a computational problem in terms of energy, time and space. We abstract away from any hardware architecture and use a model of networks of LIF neurons to describe our computations on neuromorphic hardware. We will elaborate on this in the remaining parts of this section.

2.2 Neuromorphic Complexity Analysis

In traditional computing, computational complexity theory gives an indication of the resources needed to solve a computational problem in terms of their input size under the traditional Turing machine model [23]. Through computational complexity theory, we are able to define classes of problems that require at least a certain amount of time and space resources, including methods and tools to analytically assign a specific computational problem to a certain class. This not only gives us a fundamental understanding of what types of problems can and cannot be efficiently solved but also provides us with an analytical approach to determine whether a new computational problem is efficiently solvable or not.

In contrast to traditional computing, we do not have a strong understanding of the types of problems that can and cannot be solved with neuromorphic hardware. The methods and tools from computational complexity could potentially be very useful in understanding the application space of neuromorphic processors, but traditional complexity theory might not be the ideal way to analyse the resource constraints of neuromorphic systems. The resources analysed in computational complexity - time and space - are coarse and derived from an abstract model of computation. More significantly, it does not capture the resource arguably of most interest when moving towards neuromorphic solutions: energy expenditure.

Efforts are underway in designing a neuromorphic complexity theory that is more equipped to describe the resource demands of neuromorphic processors [15]. In section 3, we will build upon this work and

formalise an alternative machine model in which a traditional Turing machine communicates with a neuromorphic oracle. In sections 4 and 5 we will demonstrate how the maximum flow problem can be mapped on this model and how this machine model can capture energy expenditure.

2.3 Neural Algorithm Design

In spiking neural networks, the weights can be trained (e.g. through spike-time dependent plasticity) or programmed. In the latter case, two approaches currently exist. One approach is to design a network of stochastic spiking neurons in such a way that it corresponds to an instance of a particular optimization problem, e.g.the travelling salesperson problem (TSP). We can achieve this by constructing basic circuits such as winner-take-all circuits or logic circuits. These circuits constrain the spiking behaviour of the network in such a way that it creates an energy landscape (distribution of spike-based network states) that leads to a fast convergence to the optimal solution [13].

Another approach is to carefully handcraft the network on the neuron level to obtain desirable signals and computation in the entire networks, which has proven to be successful for basic computational operations [22, 24]. Notably [22] introduced a simple discrete spiking neural model that is able to capture many interesting computational primitives, such as delays, spike-timing and leakages. We will use this model to implement our neuromorphic oracle. We will describe this model in detail in section 3.

There is currently no straightforward way to scale these approaches up to more complex compounded problems such as the maximum flow problem. In sections 3, 4 & 5 we expand on previous neural algorithm design work and demonstrate a novel way of tackling more complex problems such as the maximum flow problem.

3 Model & Problem Definition

3.1 Machine Model

We introduce a new machine model consisting of two components: (1) a Turing machine M with a read-only input tape and a working memory, (2) a neuromorphic oracle O, a computing device that receives a spiking neural network (SNN) definition from M, can simulate this SNN and output spiking events. Let L be a language of yes-instances of problem N and I be a specific instance of N. Then M implements an algorithm $A_L(I)$ that decides whether $I \in L$. In addition to that, given input I, A_L can construct any spiking network $\mathcal{S}_{L,I}$ that is subsequently communicated to and simulated by O. Both $A_L(I)$ and $\mathcal{S}_{L,I}$ work under constrained resources R_A and R_S, where R_A is a two-tuple $(\mathrm{TIME}, \mathrm{SPACE})$ and R_S is a three-tuple $(\mathrm{TIME}, \mathrm{SPACE}, \mathrm{ENERGY})$. This way we can prevent the construction of $\mathcal{S}_{L,I}$ from being trivial. In order to respect the resource constraints of A_L we introduce a working memory tape from which A_L can read and write. This working memory will have size $R_A[SPACE]$. In figure 2 you can find an illustration of this model.

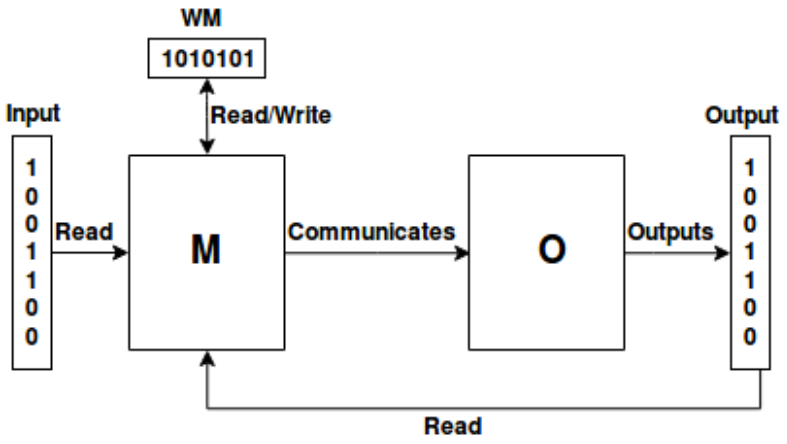

Figure 2: *Illustration of the machine model. We have a conventional device M which acts as a Turing machine with one input tape and a neuromorphic oracle O to which M can communicate a spiking neural network definition $\mathcal{S}_{L,I}$. If we have a specific language L and input I with imposed resource constraints $\{R_A, R_S\}$, M implements $A_L(I)$ and can construct a spiking network $\mathcal{S}_{L,I}$ and communicate this network to O. O subsequently simulates this network and outputs designated spiking events in the order of their spike timings. In order to constrain the space requirements of M we only allow read access from the input tape of M and we introduce a Working Memory (WM) with size $R_A[SPACE]$, where $R_A[SPACE]$ is the space constraint of A_L.*

3.2 Neural Model

For the realisation of the oracle, we adopt the neural model in [22], in which a neuron H_i is defined as a 3-tuple:

$$H_i = (T_i, R_i, m_i)$$

Where T_i, R_i, m_i are the firing threshold, reset voltage and multiplicative leakage constant respectively. A synapse is defined as a 4-tuple:

$$S_{a,b} = (d, w)$$

Where a is the pre-synaptic neuron, b is the post-synaptic neuron and d and w are the synaptic delay and synaptic weight respectively.

The spiking behaviour is determined by a discrete-time difference equation of the voltage. Suppose neuron y has voltage V_{ty} at time step t. Then we can compute the voltage at time step $t+1$ in the following way:

$$V_{t+1y} = m_y V_{ty} + \sum_{S_{xy} exists} w_{xy} x_{t+1-d}$$

Where $x_{t+1-d_{xy}} = 1$ if neuron x spiked at time step $t+1-d_{xy}$ and $x_{t+1-d_{xy}} = 0$ otherwise.
Additionally, we have voltage V_0 that denotes the initial potential of a neuron. We assume $V_0 = 0$, unless explicitly mentioned otherwise.

We make a distinction between four types of neurons. A *standard neuron* for internal computations a *readout neuron*, from which the spike events will be written on the output tape of the neuromorphic oracle O a scheduled neuron, a programatically defined neuron that full fills a certain specific role (e.g. scheduled firing or constantly firing) and an input neuron that represents the input in case the problem cannot be fully encoded in the neurons and synapses and needs external information to drive computation. In figure 3 you can find an illustration of these neuron types. Note that in this project we only make use of the readout and standard neurons.

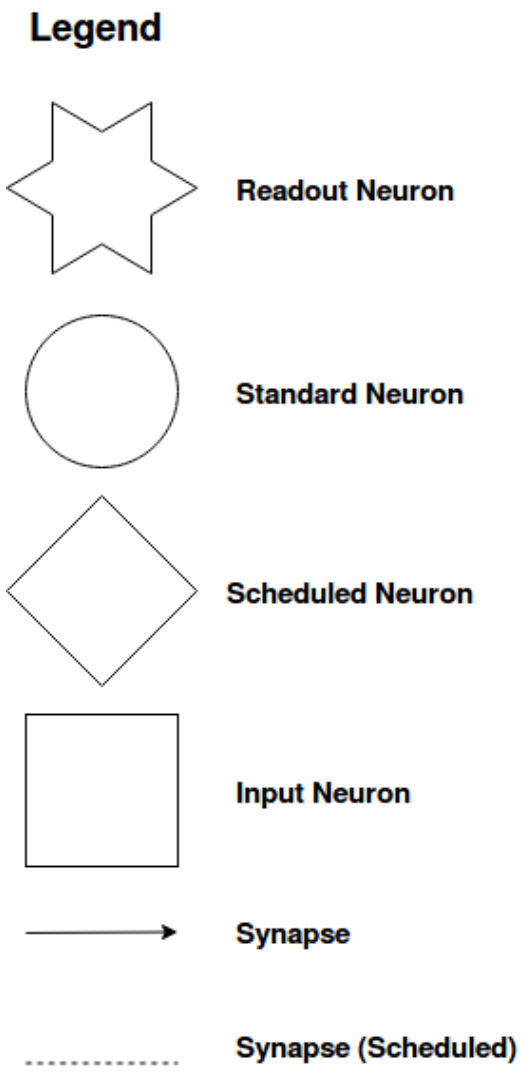

Figure 3: *Illustration of the four neuron types. The standard neuron is used for internal computations in the network, while the readout neuron can submit their spiking events on the output tape of the oracle. The scheduled neuron and input neurons are auxiliary neurons that can be used to represent an external drive and to drive standard neurons with a bias current.*

3.3 Problem Definition

Under these models we define the following problem definition for finding the maximum flow in a network based on a harder generalised decision problem variant of the maximum flow problem [2]:

Problem: MAX NETWORK FLOW

Input: A directed graph $G = (V, E)$, with designated vertices $s, t \in V$ referred to as the sink (no outgo-

ing arcs) and source (no incoming arcs) where for each edge $e \in E$ we have a non-negative integer $c(e)$, the capacity of that edge,

Output: A flow assignment $f(e)$ for each edge $e \in E$ such that $0 \leq f(e) \leq c(e)$ and $\sum_e f(e)$ is maximised.

We show that we can solve MAX NETWORK FLOW under this model with $R_A = (\mathcal{O}(n^3), \mathcal{O}(1))$ and $R_S = (\mathcal{O}(n), \mathcal{O}(n), \mathcal{O}(n))$, where n is the number of edges in the flow network.

4 Algorithm

We adopt a variant of the Ford-Fulkerson Method: the Edmonds-Karp Algorithm [9]. This algorithm repeatedly finds shortest augmenting paths, pushes flow through these paths based on the edge with the minimum capacity until all augmenting paths have been exhausted. There are three components in this algorithm that violate the logspace constraint.

1. The queue maintained by the search algorithm
2. Maintenance of the path
3. Maintenance of the flow through each edge

In all three cases, the memory demands can be linear in the input in the worst case. We introduce one spiking operation and two neuromorphic data structures, through which we offload these components to the neuromorphic oracle. A modification of the wave propagation algorithm discussed in [20] and two networks that maintain the augmenting path and the flow of the edges. Algorithm 1 describes the algorithm in full. The algorithm is defined on the Turing machine M. M can consult a neuromorphic oracle O during execution.

Algorithm 1: Spiking E-K algorithm

Data: weighted graph G = (V,E) with a capacity c for each edge

```
1  Write_Capacity(E);
2  while there is an augmenting path do
3      Spike_Search(E);
4      Min_Cap = ∞;
5      Prev_Neuron = Null;
6      while not End(O) do
7          Neuron = Read(O);
8          if Continuation(Neuron, Prev_Neuron) then
9              if Min_Cap > Cap(Neuron) then
10                 Min_Cap = Cap(Neuron);
11             Write_Path(Neuron);
12             Prev_Neuron = Neuron;
13     while not End(N) do
14         Neuron = Read_Path(O);
15         Write_Voltage(Neuron, Min_Cap);
16         Write_Capacity(Neuron);
17 Max_Flow = 0;
18 while not End(O) do
19     Neuron = Read(O);
20     Max_Flow += Voltage(Neuron);
21 return Max_Flow
```

Algorithm 1 gives a full description of a spiking version of the E-K algorithm. In line 1 we write away a readout capacity neuron for each edge on the neuromorphic oracle O, which keeps track of the

flow that has gone through the neuron. The capacity neurons are defined as:

$$C_i = (c + (|E| + 1), 0, 1)$$

Where C_i[2] is the capacity neuron for edge i in the flow network, c is the capacity of the edge that the neuron codes for. When C_i reaches its firing threshold, it will fire exactly once and instantaneously inhibit its postsynaptic neurons.

In line 3 we map our flow network onto two spiking networks in which each neuron codes for an edge in the flow network. We use the first network to find an augmenting path in the flow network and we use the second network to sort the edges in order, such that we can read out the path. We define the neurons in the first network as:

$$H_i = (1 + (|E| + 1), 0, 1)$$

The reset of the neuron is set such that this neuron can only spike exactly once. The timing of the spike will then be proportional to the length of the shortest path from the source to the edge that the neuron codes for.

The connectivity of this network is defined as follows: let $a \rightarrow b$ and $c \rightarrow d$ be two edges in the original flow network with vertices a, b, c and d. If $b = c$, we define a synaptic connection:

$$S_{H_{c\rightarrow d}, H_{a\rightarrow b}} = (1, 1)$$

This means that the direction of flow in the spiking network is reversed w.r.t. the original flow network. We then read out and connect the earlier defined capacity neurons according to:

$$S_{C_i, H_i} = (0, -(|E| + 1))$$

[2]In this instance (and the remaining part of this section), we can set $V_0 = |E| + 1$ without loss of generality.

In addition to that, we introduce a transmitter neuron T that kick-starts the wave propagation:

$$T = (1, 0, 1)$$

And connect this transmitter neuron to the neurons that code for the sink edges according to:

$$S_{T,H_{*\to t}} = (1, 1)$$

Where t is the sink vertex. The transmitter neuron will have $V_0 = 1$ at the start of the algorithm and will spike exactly once.

We define a second readout network with neurons:

$$R_i = (1 + (|E| + 1), 0, 1)$$

Where each neuron codes for a specific edge in the original flow network. The connection topology of this network is defined as follows: if $H_{s\to a}$ codes for a source edge in the first network we connect it to the second network according to

$$S_{H_{s\to a},R_{s\to a}} = (1, 1)$$

In addition to that, we define the connectivity between neurons R in the following way. let $a \to b$ and $c \to d$ be two edges in the original flow network with vertices a, b, c and d. If $b = c$, we define a synaptic connection:

$$S_{R_{a\to b},R_{c\to d}} = (1, 1)$$

This means that the direction of flow from the flow network is preserved. This will enable us to read out the path in the correct direction. We also connect the earlier defined capacity neurons according to:

$$S_{C_i,R_i} = (0, -(|E| + 1))$$

This makes sure that the readout neuron will not spike if the capacity of the edge it codes for is exhausted. Each spike event in this network will be written on the output tape of O. Note that in the second network, neurons will only fire if any of the neurons that code for the source edges in the first network fire. If this is not the case, it means that there is no path from source to sink left and we need to wait for $2 \times |E| + 1$ time steps (the longest possible wave through both networks) in order to determine that we are done. The first half of the network thus guarantees that there is indeed a path from source to sink, and the second part of the network sorts edges in such a way that we can reliably decode the path from the network. The neural algorithm will run until any of the readout neurons that code for a sink edge has spiked. If that is the case, it means that we found a path from the source to the sink. Otherwise, the algorithm will run for $2 \times |E| + 1$ steps and terminate, which means that all the augmenting paths in the network have been exhausted.

In figure 4 you can find an illustration of how a flow network is mapped on the described SNN topology.

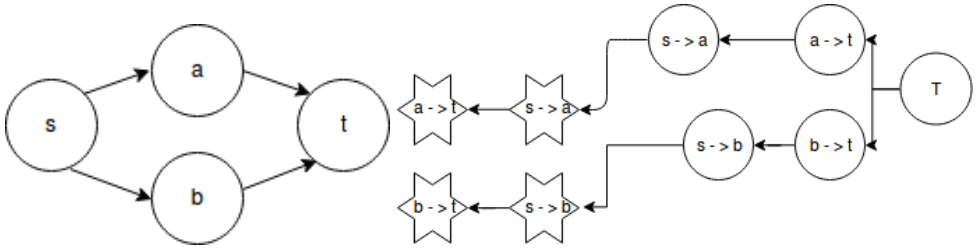

Figure 4: *Illustration of how a flow network is mapped on the SNN topology described in the text. The left graph depicts a simple flow network with a source s and a sink t and two intermediate nodes. On the right, you can see the SNN topology of the flow network. The first half of the SNN consists of standard neurons that compute the shortest path from the sink to the source. The second half consists of readout neurons that will reverse and sort the paths such that we can read out the path in the correct order. For clarity, we left out the capacity neurons that can inhibit the readout and standard neurons when the capacity of an edge is exhausted.*

In lines 6 - 12 we read out the neurons and write them on the neuromorphic oracle as a path network. We check whether the neuron is a good continuation of the path, i.e. if we have two neurons $R_{a\to b}$, $R_{c\to d}$

we have a continuation if $b = c$. In addition to that, we keep track of the minimum capacity of the neuron we found in order to determine the minimum capacity edge of the path we found. We can trivially read out the path by letting the neurons spike and read out their spike events from the output tape of O.

In lines 13 - 16 we read out the path data and update the capacity neurons based on the flow of the minimum capacity edge we found in lines 6 - 12. We then write this neuron back as a capacity readout neuron on the neuromorphic oracle.

In lines 17 - 20 we read out the voltages from the capacity neurons in order to determine the final flow through each edge. We then sum them up to determine the maximum flow through the network and return that value in line 21.

5 Complexity Analysis

In this section, we discuss the complexity analysis of A_L and $\mathcal{S}_{L,*}$. We strictly divide resources between both processors, meaning that if a computation is only counted towards the complexity of the processor if the computation happens on that processor. We will assume that the communication resources count towards the complexity of A_L. We also assume that a read or write operation takes $\mathcal{O}(1)$ time and $\mathcal{O}(1)$ space. And finally, we assume that a spike exerts $\mathcal{O}(1)$ energy.

5.1 Complexity of A_L

5.1.1 A_L runs in $\mathcal{O}(n^3)$ Time

We show that A_L has time complexity $\mathcal{O}(n^3)$, where n denotes the number of edges in the network. In line 1, we write capacity neurons on O. Given that we have $|E|$ edges, this part is linear in the number of edges. In line 3, we create a spiking network that can determine the shortest augmenting path in the flow network. In order to create this network, we need to read the edges from the input tape of M, create a neuron that codes for this edge and communicate it to O. Next, we need to connect each neuron to

their neighbouring edges and their respective capacity neurons. We can achieve that by reading out each neuron in the spiking network, read out the respective capacity neurons and read out the search neurons and communicate a connection to O if the identifiers of the neurons match. For each neuron, you need to check at most $2 \times |E|$ neurons and matching the identifiers only takes $\mathcal{O}(1)$ time, which means that this procedure takes $\mathcal{O}(n^2)$ time. In lines 6 - 12 we read out the neurons that spiked in the spiking network and build up a path. Every operation within the while statement takes $\mathcal{O}(1)$ time. Since there are $|E|$ readout neurons the complexity of the entire loop will be $\mathcal{O}(n)$ time. Similarly in lines 13 - 16 we only read out neurons that code for the path, which has size at most $|E|$, which means that we also only need $\mathcal{O}(n)$ time. Computing the maximum flow in lines 17 - 20 then also only takes $\mathcal{O}(n)$ time. The outer loop depends on the number of augmenting paths. Since in each iteration, one edge will be saturated, there are $\mathcal{O}(|E|)$ possible paths, which is still polynomial in the input. Given that dominating time complexity within the loop is $\mathcal{O}(n^2)$, the entire procedure runs in $\mathcal{O}(n^3)$ time.

5.1.2 A_L runs in $\mathcal{O}(1)$ Space

We show that the algorithm described in section 4 only uses $\mathcal{O}(1)$ space for the preprocessor, besides the encoding of the input. In section 4, we identified three bottlenecks of E-K algorithm. We will address how all three bottlenecks are resolved. Since the search algorithm is entirely written off to the neuromorphic oracle, there is no in-memory maintenance of any sort of queue so we only use $\mathcal{O}(1)$ memory. Likewise, we do not fully maintain the path in memory but use the property that the neurons are sorted according to their spike timing. We then read every neuron and determine on the basis of their ID's whether it is a correct continuation and we can at the same time record the capacity encoded by the neuron in order to determine the minimum flow. Hence we only need to allocate $\mathcal{O}(1)$space. Likewise, since we code the flow into the threshold of the capacity neurons, we do not need to maintain it in the WM of M, since we can read and communicate flow information from and to the oracle O, which means that we only need to allocate $\mathcal{O}(1)$ space. This means that we can run this algorithm using only $\mathcal{O}(1)$ memory on the preprocessor and therefore A_L runs in $\mathcal{O}(1)$ space.

5.2 Complexity of $\mathcal{S}_{L,*}$

5.2.1 $\mathcal{S}_{L,*}$ runs in $\mathcal{O}(n)$ Time

The time complexity of $\mathcal{S}_{L,*}$ is dominated by the search procedure since retrieving information from the path and capacity neurons can be trivially solved by letting them spike in $\mathcal{O}(1)$ time. The worst case that can occur consists of one path (i.e. a chain of nodes). In that case, we need $2 \times |E| + 1$ time steps to receive an output from the oracle, which reduces to $\mathcal{O}(n)$ time. Since we have that $|E| = |V| - 1$ when the network is a chain of nodes, we can further refine that to $\mathcal{O}(l(s,t))$ where $l(s,t)$ denotes the path length from source to sink.

5.2.2 $\mathcal{S}_{L,*}$ runs in $\mathcal{O}(n)$ Space

We need to allocate $|E|$ edges for the capacity neurons. For the path neurons, we need to allocate at worst $|E|$ neurons and for the search network, we need to allocate $2 \times |E|$ neurons. Which results in $4 \times |E|$ neurons in total. Which means that we need $\mathcal{O}(n)$ space.

5.2.3 $\mathcal{S}_{L,*}$ uses $\mathcal{O}(n)$ Energy

Each neuron in the search network spikes at most one time. Since we have $3 \times |E| + 1$ neurons (including the capacity neurons) in the networks we need $\mathcal{O}(n)$ energy.

From the above description we have that $R_A = (\mathcal{O}(n^3), \mathcal{O}(1))$ and $R_S = (\mathcal{O}(n), \mathcal{O}(n), \mathcal{O}(n))$ It therefore must be the case that MAX NETWORK FLOW is in $\mathsf{L}^{\mathsf{SNN}(\mathcal{O}(n),\mathcal{O}(n),\mathcal{O}(n))}$. Since the time complexity of the search query depends on the longest path between source and sink we can further refine this result to $\mathsf{L}^{\mathsf{SNN}(\mathcal{O}(l(s,t)),\mathcal{O}(n),\mathcal{O}(n))}$ where $l(s,t) \leq n - 1$ denotes the maximum path length between source and sink.

6 Empirical Analysis

In this section, we will describe our empirical methodology. The goal of our empirical investigation is to validate our theoretical complexity results in actual neuromorphic hardware. In this way, we will be

able to analyse whether our formal complexity results translate to practical reality and if our proposed theoretical machine model can be mapped onto neuromorphic hardware. This analysis consists of two phases. In the first phase, we will validate our complexity results in a neural simulator we have developed. This phase will serve as a sanity check prior to the hardware implementation and is primarily aimed at testing the algorithm we defined in section 4. In the second phase, we will implement our algorithm on the Loihi chip. Our aim in this phase is to determine whether (1) our energy analysis holds on actual neuromorphic hardware, (2) the communication line between conventional processor and neuromorphic coprocessor introduces significant additional overhead that is not captured in our machine model, (3) it is practical to implement our algorithm under the proposed machine model and to map out the compromises we have to make in order to end up with a workable solution.

6.1 Software Validation

We will first describe our methodology w.r.t software validation. We implemented a variant of the proposed algorithm in a neural simulator we have developed for the purposes of this project. We look at the part of the algorithm that dominates resources constraints. This will be the search algorithm. We will compare the BFS algorithm on a conventional processor against the search algorithm implemented under our machine model in terms of time and energy resources. For our machine model, we will split this into two parts, the part of the algorithm that runs on the conventional processor and the part of the algorithm that runs on the neuromorphic co-processor. On the neuromorphic co-processor, we measure the time, space and energy demands of the spiking network.

We compute two measures of interest after running each spiking network. The first measure is the number of total spikes in the network, which serves as a proxy for the energy demands.The second measure is the number of time steps needed until the sink neuron spikes in order to see how the time demands of the spiking networks increase with the number of nodes. We also look at the absolute difference of the flow computed by the spiking network and a reference maximum flow computed with the Edmonds-Karp algorithm [9] in order to validate the accuracy of the solution. For the part of the algorithm that runs on the conventional processor, we need to take into account the resources that it takes to write the network

onto the tape. As the size of the network does not change over execution time, we can abstract away and estimate the resources in terms of time.

In order to estimate energy demands on the conventional processor, we will assume that the energy demands are bounded by the time complexity on the conventional processor by a constant. This approximation is based under a different (but equivalent) machine model in which each operation can be decomposed into a set of primitive manipulations on the register [7]. We will use this same method for the search algorithm that is fully implemented in the conventional processor.

We will compare two types of graphs. Graphs that have a low degree of connectivity relative to their number of nodes (i.e. sparse graphs) and graphs that have high connectivity relative to their number of graphs. This allows us to untangle in which instances the hybrid model might outperform the conventional method.

Since we are interested in how these measures grow when the size of the network increases we compute these measures over a series of networks with an increasing number of nodes. For each size, we randomly generate flow networks according to the following procedure[3].

1. Specify the number of nodes n and the number of edges m in the network

2. generate n nodes, for each node identifier i (a natural number): connect i to all nodes with identifier $i + 1$ or higher.

3. The above procedure will yield a fully connected acyclic component. If this component contains more than m edges: pick a random edge and delete it from the network. If the remaining network is not a fully connected component, put the removed edge back in the network. Repeat this procedure until you end up with a network of m edges.

4. Specify a maximum capacity c_{max} and randomly assign a capacity $[1, c_{max}]$ to each edge.

We will then run our algorithm on the generated networks, compute the aforementioned measures and average them out over these networks in order to obtain an average estimate for each network size. The

[3]All non-proprietary software developed for this project can be found **here**.

maximum capacity and connection density is fixed between network sizes in order to obtain comparative results.

6.2 Software Results

There was no divergence between the conventional E-K algorithm implementation and the spiking version of the E-K algorithm, indicating that the spiking version works correctly. In figure 5 you find the results of the software simulation.

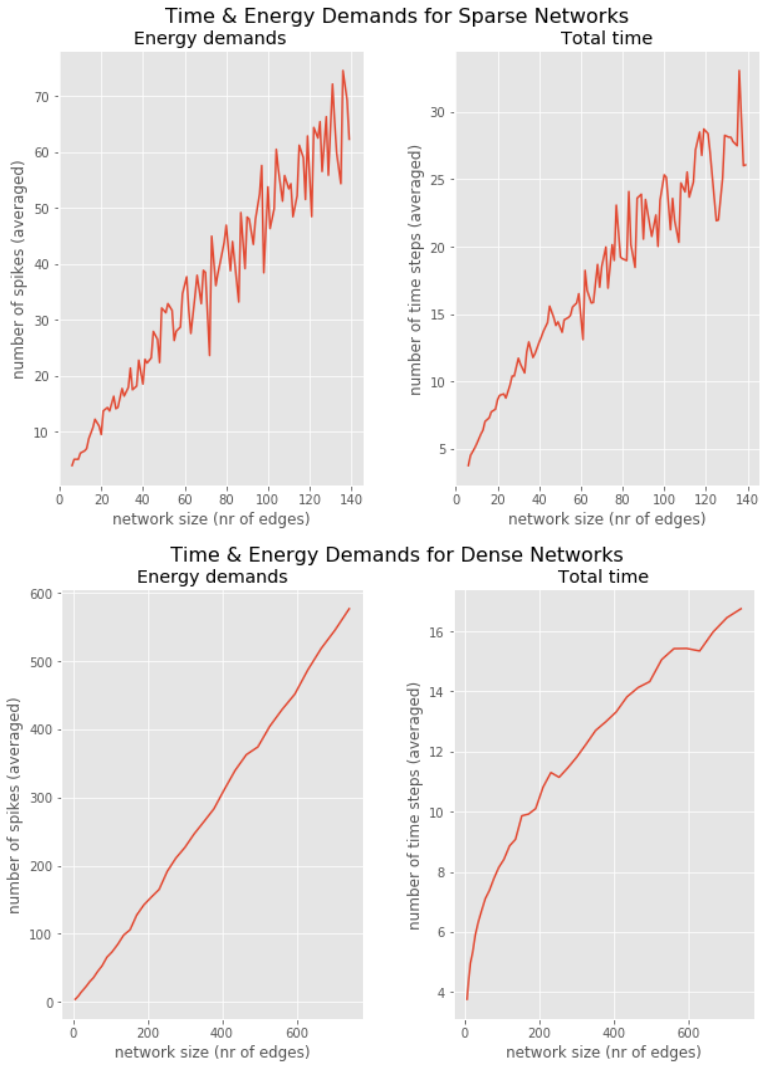

Figure 5: *Average growth of energy expenditure and time complexity of randomised flow graphs as a function of the number of edges under the hybrid model. We tested two types of graphs. Sparse graphs, i.e.* $|E| = 1.4|V|$ *and dense graphs, i.e.* $|E| = \frac{|V|(|V|-1)}{2}$*. For sparse graphs, we generated graphs with 5 - 100 nodes and for the dense graphs, we generated graphs with 5-40. Each point in each graph is an average over the mean number of time steps and spikes over the search queries.*

Since there is no obvious first-order proxy for energy consumption on a conventional model at the algorithmic level, we will first lay out our assumptions in this comparison. We assume that resources spent in moving data, i.e. from conventional processor to neuromorphic coprocessor in the hybrid model, and from RAM to CPU in the conventional model will be roughly proportional to each other and this will be left out of this analysis. We will also assume that the energy complexity of the conventional model will

be proportional to the time that it takes to execute a certain computation. That is, if a certain computation takes k time steps, the energy expended will be counted as ck where c is certain nonnegative constant. As already mentioned earlier we will use the BFS algorithm [19] as a benchmark which has worst-case time complexity

$$R_{BFS}[TIME] = \mathcal{O}(|V| + |E|)$$

Under our aforementioned assumptions this will lead to the following energy expenditure:

$$R_{BFS}[ENERGY] = c|V| + c|E|$$

In figure 5 we observe that for sparse networks, energy grows as $\frac{|E|}{2}$. If we assume $c > 0$ this means that for sparse networks we might predict an improvement in energy efficiency since $\frac{|E|}{2} < c|E| + c|V|$, but hardware validation needs to confirm this observation. In dense networks, we see a strict linear growth in terms of energy. Since we have that $|E| = \frac{|V|(|V|-1)}{2}$. We end up with the inequality:

$$c|V| + c\frac{|V|(|V|-1)}{2} > \frac{|V|(|V|-1)}{2}$$

Which will be satisfied if $c > 0$, which means that if the energy expenditure is a multiple of the number of time steps, we can predict that we would also see improvements in terms of energy efficiency for dense nets.

In terms of time complexity we can see in figure 5 that both networks type show growth of $log(|E|)$, which is strictly more efficient than the conventional model. This indicates strong evidence for an improvement in terms of time complexity in the hybrid model.

6.3 Hardware Validation

In section 6.2 we concluded that there are potential efficiency gains in time and energy in the hybrid model. In order to arrive to this conclusion we made two assumptions: (1) the communication overhead

between the conventional processor and neuromorphic processor is negligible, (2) one operation on the neuromorphic processor is proportional to one unit of energy and therefore one operation on the neuromorphic processor will be more efficient than an operation on the conventional processor. In this section, we will attempt to verify these 2 assumptions.

We implemented the described algorithm in section 4 on the Loihi Nahuku board. In order to meet the restrictions of the API of the Loihi processor we only implemented the search algorithm on the neuromorphic cores. Since the search algorithm is the major resource consumer in this algorithm we do not expect this revision to have major effects on the obtained results. We benchmarked the Nahuku board on two different levels of detail, the macro -and micro level. On the macro-level, we look at the entire system (neuromorphic chip and auxiliary systems) and measured network set-up, compilation and execution time. The set-up and compilation time gives us an estimate of how much resources the communication line between the conventional -and neuromorphic processor consumes and the runtime will give us an estimate of the execution time on the Loihi chip. On the micro-level, we zoom in and look at energy and execution performance on the neuromorphic chip. This will give us a more detailed estimate of how the energy and runtime demands evolve for larger networks. We repeated the experiments in section 6.2 under the same conditions.

6.4 Macro-level Results

In figure 6 you can find the results for the macro-level benchmarks. We tested sparse and dense connected network as formulated in section 6.2, each data point is an average of 10 randomly sampled flow networks.

For sparse networks, we see an initial high offset in execution time followed by slow growth and a negligible cost in communication overhead (setup time and compile time).

For dense networks, however, we see a higher cost in runtime. This can be explained by the fact that the number of edges grows much faster in dense networks compared to sparse networks. Note that in our original algorithm we could stop the execution on the basis of a signal of the neuron (spike of a source

neuron). This is not possible in the Loihi API so we had to upper bound our execution time to the worst possible outcome (each neuron spikes before we have a solution). This, in particular, deteriorates the runtime results for dense networks. For dense networks, we again see a negligible cost in communication overhead.

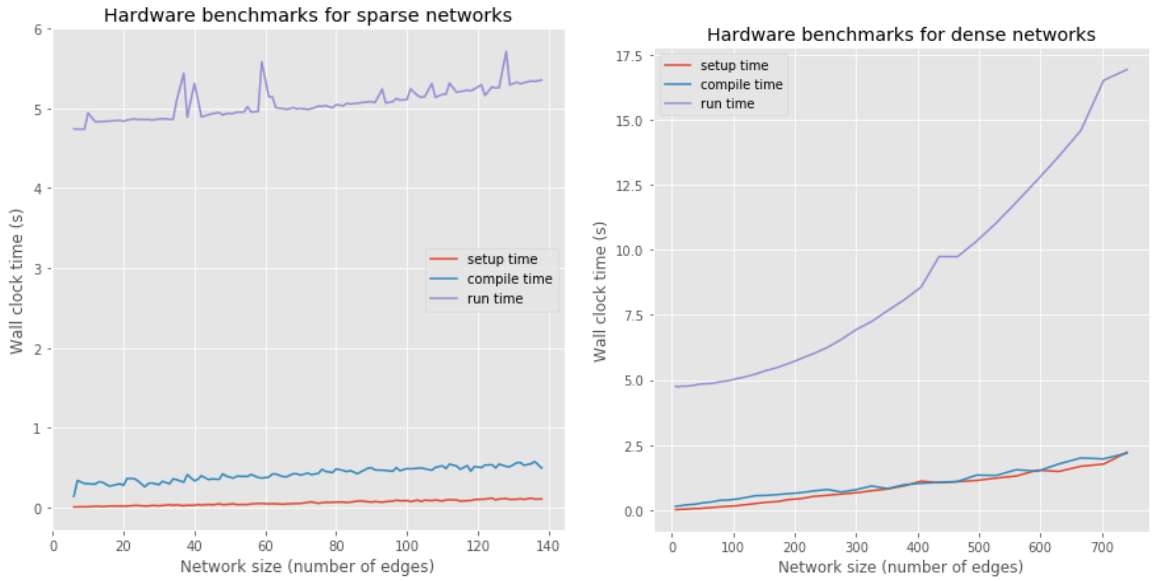

Figure 6: *Macro-level benchmark of Loihi system. We benchmarked the set-up time of the network (mapping flow net to a NxNet definition), compilation-time of the network and the execution time. On the x-axis, we have the number of edges in the original flow network and on the y-axis the wall clock time in seconds. For both network types, we see a constant/fairly slow growth in runtime indicating that the search algorithm scales very well in terms of actual simulation on the neuromorphic chip. In terms of set-up and compilation we see that both network types show slow scale-up as predicted by our previous simulations.*

These results demonstrate that communication overhead is negligible, verifying our first assumption. In the case of sparse networks, we even see an improvement in runtime results.

6.5 Micro-level Resuts

We benchmarked sparse networks varying from 15 nodes up to 100 nodes and dense networks varying from 15 to 40 nodes. We measured the execution time, energy demands and power demands. Each

data point consists of 10 randomly sampled flow networks and in figure 7 and 8 you can see the results obtained from these benchmarks.

There is a slightly super-linear relationship between the network sizes and the measured statistics, but note that the execution times and power and energy scale in exactly the same way. These results fall in line with our second assumption and show that running the search query on a neuromorphic chip will be more energy-efficient, under the assumption that an operation on a neuromorphic chip will consume less energy.

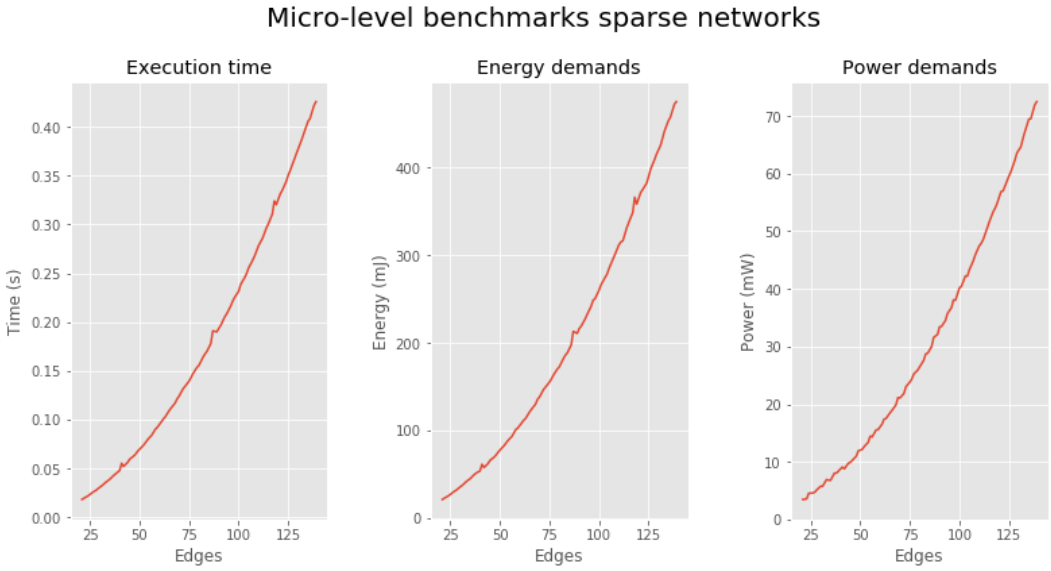

Figure 7: *Micro-level benchmarks on the Loihi chip. Sparse networks varying from 15 nodes up to 100 nodes were benchmarked. We measured the execution time, energy demands and power demands. Each data point consists of 10 randomly sampled flow networks. On the x-axis are the number of edges in the network and on the y-axis are the magnitudes of interest. Execution time is measured in seconds, energy in milli-Joules and power in milliwatts.*

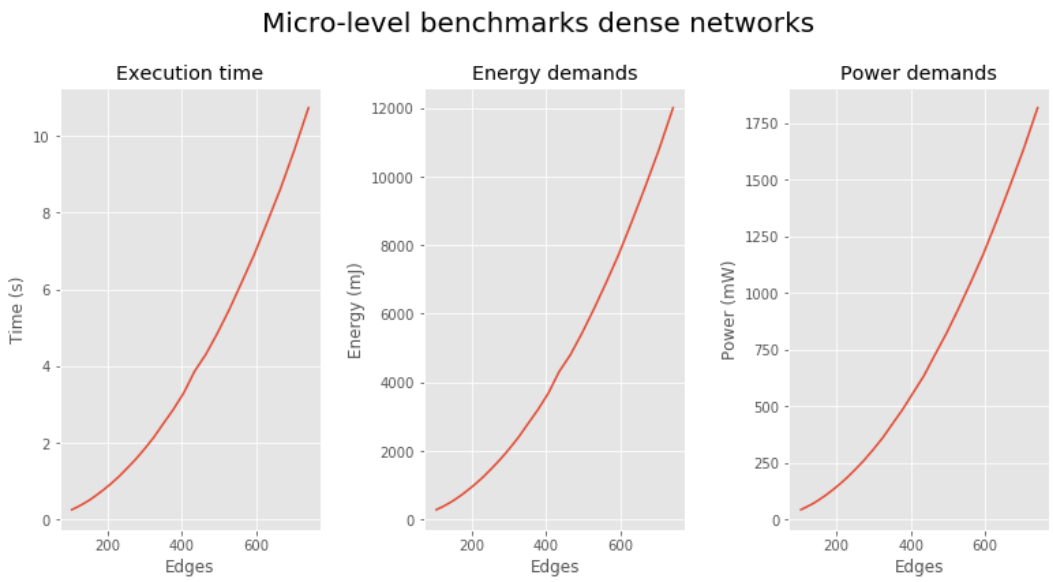

Figure 8: *Micro-level benchmarks on the Loihi chip. Dense networks varying from 15 nodes up to 40 nodes were benchmarked. We measured the execution time, energy demands and power demands. Each data point consists of 10 randomly sampled flow networks. On the x-axis are the number of edges in the network and on the y-axis are the magnitudes of interest. Execution time is measured in seconds, energy in milli-Joules and power in milliwatts.*

7 Discussion

We have demonstrated a potential pipeline from theory to practical implementation in order to systematically investigate potential application areas for neuromorphic processors. Below we will discuss the advantages and limitations of this pipeline.

7.1 What can we learn from theory?

From classical complexity [11], we learned that the MAX NETWORK FLOW is hard to implement on contemporary neuromorphic processors due to an inherently serial component in the problem. In sections 3 to 5 we demonstrated a theoretical approach in which we introduced a new machine model, with a new lattice of complexity classes, in which we unfold a new source of complexity: energy. On the basis of

this model, we proposed an algorithm that shows that we can satisfy the logspace constraint. Something that was not possible under the classical Turing model.

In addition to that, under the assumption that one operation takes one unit of energy and communication overhead is negligible, we unveiled that we can get potential efficiency gains through off-loading parts of the algorithm on a neuromorphic co-processor.

While some of the theoretical results did not hold when moving to practice, this at least shows in what way such a theoretical approach could be beneficial. It helps us understand what aspects of a problem make a problem hard or easy to implement in neuromorphic hardware. Moreover, it gives us the ability to generalize this understanding to a larger class of problems (in this case the class of P-complete problems). And finally, it allows us to come up with an alternative way to solve this problem and gives us pointers to potential efficiency gains we can get in terms of time, space and energy. Especially the last point is hard to arrive at without a fundamental understanding of the hardness of a certain problem. Theory can, therefore, provide a sound and rigorous basis on which we can motivate why a certain application area is suited for neuromorphic solutions and help us come up with novel ways to solve a certain problem based on our understanding of the sources of complexity in the problem.

7.2 From theory to practice: what do we sacrifice and what do we learn?

In sections 6 we continued to build on our theoretical results and implemented the algorithm we described in section 4. We split this part up in two phases: a software phase and a hardware phase. The software phase served as an initial sanity check. It helped us spot potential problems in the proposed algorithm and understand in what way we have to modify our algorithm to make it work on neuromorphic hardware. We notably only offloaded the search part on the neuromorphic processor and did not include the maintenance of the flow and the maintenance of the path. This violates the logspace constraint, but through the results in section 5 we were able to predict potential efficiency gains in time and energy.

An important comment to make is that these predictions only held under two assumptions: (1) negligible communication overhead, (2) a linear relationship between runtime and energy. In the hardware

phase, we verified these assumptions and ran a modified algorithm on the Loihi platform. We split these analyses up in a macro and micro part. In the macro part, we showed that in the full system (i.e. Loihi chip + network setup, compilation and sending and handling jobs) the biggest source of complexity is the runtime. In both networks, we found that the communication overhead was negligible. This indicates that our first assumption holds. In the case of sparse networks, we even found that the runtime grows very slow w.r.t. the network size indicating potential runtime efficiency gains. Important to note is that the overall runtime has an initial high cost but subsequently grows rather slow indicating that a hybrid approach is only cost-efficient if the networks are somewhat of large scale.

In the micro part, we zoomed in on the Loihi chip and looked at how the execution time, energy demands and power demands grow as the network sizes grow. In figures 7 and 8 we see that there is a roughly linear relationship with time, energy and power, this means that our earlier prediction that time would scale roughly logarithmically with network size does not hold (see figure 5, but it does confirm that energy demands roughly scale linearly with network size. This means that our assumption that one operation is roughly proportional to one unit of energy holds. Since the Loihi processor is much more energy efficient than conventional processors [8], we have strong evidence that off-loading the search query to a neuromorphic processor yields efficiency gains while not sacrificing runtime complexity for relatively large scale networks.

The above discussion shows that software validation phase is a good complement to actual hardware benchmarking. It helps you understand limitations in your algorithm and it helps you pinpoint under what circumstances you might see efficiency gains. This in term helps in interpreting subsequent results obtained in hardware. We, therefore, see value in explicitly incorporating a software validation phase in the pipeline.

7.3 Future work: A more comprehensive complexity theory and new computational problems

From our results, several pointers of future research arise. We need a more comprehensive neuromorphic complexity theory. That includes hardness proofs, a notion of completeness, complexity classes and a means to reduce problems to other problems while preserving essential properties of the problem (e.g. time and energy) and potentially models that unfold different type of sources of complexity. Work is already done in this direction (e.g. see [15]), and this work would enhance our fundamental understanding of what makes a problem efficiently solvable on a neuromorphic processor and would greatly help us in mapping the space of potential applications for neuromorphic hardware.

In addition to that, we need to investigate new computational problems, that can lead to new algorithm design patterns such as the hybrid approach we proposed for the maximum flow problem. This, in turn, could enhance the programming tools available to neural algorithm designers.

8 Conclusion

In this project, we described a pipeline from complexity theory to practical implementation in order to systematically explore the application space of neuromorphic processors. We picked the maximum flow problem [10], a more complex algorithm than previous algorithms studied in the neural algorithm design field [24, 3].

By introducing a hybrid computational model, we were able to show that MAX NETWORK FLOW is in $\mathsf{L}^{\mathsf{SNN}(\mathcal{O}(n),\mathcal{O}(n),\mathcal{O}(n))}$. We can further refine that to $\mathsf{L}^{\mathsf{SNN}(\mathcal{O}(l(s,t)),\mathcal{O}(n),\mathcal{O}(n))}$ where $l(s,t) \leq n-1$ denotes the maximum path length between source and sink. This means that we have found a link between P-complete problems and a machine model in which we potentially could reduce the space requirements of these problems.

Through our practical analyses, we were able to confirm that there are potential efficiency gains by off-loading the search procedure to a neuromorphic processor, while not sacrificing runtime complexity. Additionally, the practical investigation also showed what comprises were needed in order to implement

this algorithm on neuromorphic hardware. Most notably, the logspace constraint was violated.

This shows that theory and practice should ideally be tightly interlinked. Theoretical analyses help us understand why certain problems can or cannot be efficiently implemented in neuromorphic hardware, and can help us in coming up with novel ways of solving problems. Practical investigations then help us refine our algorithm and/or theoretical model. Ideally when employing this pipeline, one should iterate back and forth from theory to practice.

Future endeavours would include, a more comprehensive neuromorphic complexity theory that would better allow us to map out the application space of neuromorphic hardware systems and new neural algorithm design patterns that could help us tackle problems in novel ways.

Acknowledgements

This work was supported by a grant obtained from Intel Corporation.

References

[1] Filipp Akopyan et al. “Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10 (2015), pp. 1537–1557.

[2] Abdullahi Ali and Johan Kwisthout. “A neural spiking algorithm for network flow problems”. In: *To appear on arXiv* (2019).

[3] James B Amione et al. “Non Neural Network Applications for Spiking Neuromorphic Hardware”. In: *3rd international workshop on post Moore era supercomputing*. PMES. 2018.

[4] Richard Anderson and Joao C. Setubal. “A parallel implementation of the push-relabel algorithm for the maximum flow problem”. In: *Journal of parallel and distributed computing* 29.1 (1995), pp. 17–26.

[5] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[6] David A Bader and Vipin Sachdeva. *A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic*. Tech. rep. Georgia Institute of Technology, 2006.

[7] Stephen A Cook and Robert A Reckhow. “Time bounded random access machines”. In: *Journal of Computer and System Sciences* 7.4 (1973), pp. 354–375.

[8] Mike Davies et al. “Loihi: A neuromorphic manycore processor with on-chip learning”. In: *IEEE Micro* 38.1 (2018), pp. 82–99.

[9] Jack Edmonds and Richard M Karp. “Theoretical improvements in algorithmic efficiency for network flow problems”. In: *Journal of the ACM (JACM)* 19.2 (1972), pp. 248–264.

[10] Lester R Ford Jr and Delbert R Fulkerson. *A simple algorithm for finding maximal network flows and an application to the Hitchcock problem*. Tech. rep. RAND CORP SANTA MONICA CA, 1955.

[11] Leslie M Goldschlager, Ralph A Shaw, and John Staples. “The maximum flow problem is log space complete for P”. In: *Theoretical Computer Science* 21.1 (1982), pp. 105–111.

[12] Bo Hong and Zhengyu He. “An asynchronous multithreaded algorithm for the maximum network flow problem with nonblocking global relabeling heuristic”. In: *IEEE Transactions on Parallel and Distributed Systems* 22.6 (2011), pp. 1025–1033.

[13] Zeno Jonke, Stefan Habenschuss, and Wolfgang Maass. “Solving constraint satisfaction problems with networks of spiking neurons”. In: *Frontiers in neuroscience* 10 (2016), p. 118.

[14] Muhammad Mukaram Khan et al. “SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor”. In: *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*. Ieee. 2008, pp. 2849–2856.

[15] Johan Kwisthout. “Towards Neuromorphic Complexity Analysis”. In: *Extended abstract presented at the NICE workshop 2018* (2018).

[16] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), p. 436.

[17] Wolfgang Maass. “Noise as a resource for computation and learning in networks of spiking neurons”. In: *Proceedings of the IEEE* 102.5 (2014), pp. 860–880.

[18] Carver Mead. “Neuromorphic electronic systems”. In: *Proceedings of the IEEE* 78.10 (1990), pp. 1629–1636.

[19] Edward F Moore. “The shortest path through a maze”. In: *Proc. Int. Symp. Switching Theory, 1959*. 1959, pp. 285–292.

[20] Filip Jan Ponulak and John J Hopfield. “Rapid, parallel path planning by propagating wavefronts of spiking neural activity”. In: *Frontiers in computational neuroscience* 7 (2013), p. 98.

[21] Johannes Schemmel et al. “Live demonstration: A scaled-down version of the brainscales wafer-scale neuromorphic system”. In: *Circuits and systems (ISCAS), 2012 IEEE international symposium on*. IEEE. 2012, pp. 702–702.

[22] William Severa et al. “Spiking network algorithms for scientific computing”. In: *Rebooting Computing (ICRC), IEEE International Conference on*. IEEE. 2016, pp. 1–8.

[23] Alan Mathison Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London mathematical society* 2.1 (1937), pp. 230–265.

[24] Stephen J Verzi et al. “Optimization-based computation with spiking neurons”. In: *Neural Networks (IJCNN), 2017 International Joint Conference on*. IEEE. 2017, pp. 2015–2022.