Evaluating Hopfield-network-based linear solvers for hardware constrained neural substrates

Rohit Shukla, Erik Jorgensen, and Mikko Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
Madison, WI 53706
Email: (rshukla3, ejorgensen2)@wisc.edu, and lipasti@engr.wisc.edu

***Abstract*—Emerging neural hardware substrates, such as IBM's "TrueNorth" neurosynaptic system, can provide an appealing platform for deploying numerical algorithms. For example, a recurrent Hopfield neural network can be used to solve for the Moore-Penrose matrix inverse, which enables a broad class of linear optimizations to be solved efficiently, with very low energy cost. However, deploying numerical algorithms on hardware platforms that severely limit the range and precision of representation for numeric quantities can be quite challenging. This paper discusses these challenges and experimentally validates that generalized inverses can be correctly computed on such substrates. Specifically, we show that several real-time applications that require matrix inverse computations to solve systems of linear equations can be correctly implemented on hardware-constrained neural substrates. The Hopfield linear solver model is empirically validated on the IBM TrueNorth platform, and results show promising potential for deploying an accurate and energy-efficient generalized matrix inverse engine calculator, with compelling real-time applications including target tracking (object localization), optical flow, and inverse kinematics.**

I. Introduction

Systems of linear equations can be used to describe a broad class of computational problems with compelling practical applications. For example, the mammalian visual system is continuously attempting to map visual stimuli to learned objects in spite of variations in scale, rotation, and position in the visual field. Numerous prior works have addressed the mammalian visual system's ability to recognize objects based on their invariant features, and have extended these concepts from the mammalian visual system to computer vision algorithms that enable robust object recognition and classification [1], [2], [3] and [4]. On the hardware front, building on foundational work on silicon neural systems [5], IBM's "TrueNorth" neurosynaptic system has shown promise for new generation of extremely low power embedded computing platforms [6]. One question that remains to be addressed is how a human being is able to learn the invariant transformations that map the seen object to the reference object present in the memory. And once this mechanism has been understood, can it be mapped onto neural substrates such as IBM TrueNorth?

Prior work such as [7], and [3] have addressed the problem of transformation discovery and proposed a computational model which explains how the mammalian visual system learns invariant transformations such as translation, rotation and scaling, and re-organizes itself as a new input stimuli is presented so that the input object can be mapped to the object stored in the memory. Both frameworks have proposed that the invariant transformations are stored as groups and once the input stimuli is presented, the model searches through all of the stored transformations and eventually selects the set of transformations that have the best match between input stimuli and the stored memory object. Even though both the algorithms have strong biological evidence backing up their claims, due to the structure of their framework, mapping these algorithms on the current generation of neural substrates is quite challenging. To address this problem, this paper proposes a mathematical framework for a Hopfield neural-network based linear solver that discovers these transforms, and robustly matches objects with visual stimuli while revealing the corresponding affine transform that maps the input to its corresponding memory template.

Mapping such an algorithm to a substrate such as TrueNorth requires the designer to carefully choose a strategy for quantizing inputs and weights. This paper examines the challenges that might come up when implementing a Hopfield network-based linear solver. In addition to affine transform-based object recognition and localization, which enables real-time object tracking, we also study two additional applications which also require finding the pseudo-inverse of a system of linear equations under real-time constraints: optical flow, which determines the direction and velocity of motion from successive frames of visual input, and inverse kinematics, which is used for motion planning of robotic arms. We report the computing resources that are required for these algorithms, and quantify how the errors or inefficiencies can be addressed to enable practical deployment of the Hopfield linear solver for these applications.

II. TrueNorth Architecture

The IBM Neurosynaptic System "TrueNorth" is a low power spiking neural network architecture that integrates 1 million programmable spiking neurons. The single chip system consists of 4096 cores where each core is composed of 256 axons (inputs) and 256 neurons (outputs), connected via a 256 x 256 crossbar of configurable synapses (about 65536 programmable synapse connections). The system operates at a rate of 1KHz, during which membrane potential processing and spike event

routing occur asynchronously inside the chip. Spikes generated by a neuron can target any axon on the chip, with each neuron presenting over 20 individually programmable features (e.g. threshold, leak, and reset). Each neuron's equations and synaptic states are updated every millisecond, which is referred to as 1 tick.

The following equations define the membrane potential (defined as $V_j(t)$) dynamics of a j-th TrueNorth neuron at time t.

$$V_j(t) = V_j(t-1) + \sum_{i=0}^{255} A_i(t) w_{i,j} s_j^{G_i} \tag{1}$$

$$V_j(t) = V_j(t) + \lambda_j \tag{2}$$

$$\begin{aligned} &\text{if } V_j(t) < 0 : V_j(t) \leftarrow 0 \\ &\text{if } V_j(t) \geq \alpha_j : \text{Spike and set } V_j(t) \leftarrow V_j(t) - \alpha_j \end{aligned} \tag{3}$$

Equation 1 represents the integration of membrane potential at time t when a neuron i receives binary valued input spike $A_i(t)$ from i-th axon. The term $w_{i,j}$ is the binary-valued synaptic connection between axon i and neuron j. Finally the term $s_j^{G_i}$ is the synaptic weight between axon i and neuron j, where G_i indicates which one of the four types was selected for i-th axon. Each neuron has configurable 9-bit signed integer weight, where most significant bit serves as the sign bit and the possible range of value that weights can take $\in [-255, 255]$.

Equation 2 integrates leak value λ_j to the neuron's membrane potential $V_j(t)$. Lastly, equation 3 compares the updated membrane potential (after weight integration) with the threshold. This equation shows the linear rest mode property of truenorth neuron. If $V_j(t)$ is equal to or surpasses the threshold (α_j), the neuron spikes and its membrane potential is subtracted by the threshold value. Truenorth has assigned 18 bits of precision to threshold parameter. Therefore, the possible range of values that α_j can take is from 0 to 262143.

In the following sections we will look at the mathematical equations which computes generalized inverse of matrix, followed by the implementation details regarding how the said equation would have to be modified so that it is easily mapped onto TrueNorth.

III. Hopfield Neural Network-based Linear Solver

Equation 4 shows the linear equation that maps the set of values in A to the set of values in B. A represents the initial set of features / values before undergoing any form of transformation, B represents the new set of observed features / values after the object has undergone transformation and X is the transformation matrix that maps elements of A to elements of B.

$$AX = B \tag{4}$$

Let there be m number of features and each feature is represented by n dimensional vector, then matrices A and B will have $m \times n$ dimensions. If $rank(m) \geq rank(n)$, the left-inverse of matrix A is computed, as shown in eqn. 5. Otherwise, the right-inverse of matrix A is computed, as shown in eqn. 6.

$$X = (A_L)^\dagger B = (A^T A)^\dagger A^T B \tag{5}$$

$$X = (A_R)^\dagger B = A^T (A A^T)^\dagger B \tag{6}$$

Where, the sign † represents pseudoinverse of a matrix.

For the set of applications that have been presented in this paper, only the Hopfield neural network architecture that can compute left-inverse of matrix A will be considered, that is, eqn. 5. As per the Hopfield network model presented in [9] and the mathematical proofs given in [10](page 678 theorem 16), equation 5 can be solved iteratively using equation 7.

$$X_{k+1} = (I_n - \alpha A^T A) X_k + \alpha A^T B \tag{7}$$

In order for the Hopfield network to solve equation 7, α has been chosen to ensure convergence. The architecture of the Hopfield neural network based linear equation solver can be summarized as,

1) A feedforward layer with input matrix B and weight matrix $W_{ff} = \alpha A^T$
2) A recurrent layer with weight matrix $W_{hop} = I - \alpha A^T A$ whose inputs are the outputs of feedforward layer.
3) The term α is assigned based on the inequality defined in equation 8.

$$0 < \alpha < \frac{2}{trace(A^T A)} \tag{8}$$

α is chosen such that the spectral radius, $\rho(W_{hop})$ is less than 1 (The eigenvalues of W_{hop} lie within the unit circle of the complex plane).

Equation 9 shows the steps on how the variable X gets updated at each iteration

$$\begin{aligned} X_0 &= W_{ff} B \\ X_1 &= W_{ff} B + W_{hop} X_0 \\ &\vdots \\ X_{k+1} &= W_{ff} B + W_{hop} X_k \end{aligned} \tag{9}$$

Thus, the update rule for X_{k+1} can be formulated as equation 10

$$X_{k+1} = \sum_{i=0}^{k+1} (W_{hop})^i W_{ff} B \tag{10}$$

By selecting the appropriate value of α as shown in equation 8, and as per the theorem and proof presented in [11], it can be stated that equation 11 holds true.

$$\lim_{k \to \infty} (W_{hop})^{k+1} = 0 \tag{11}$$

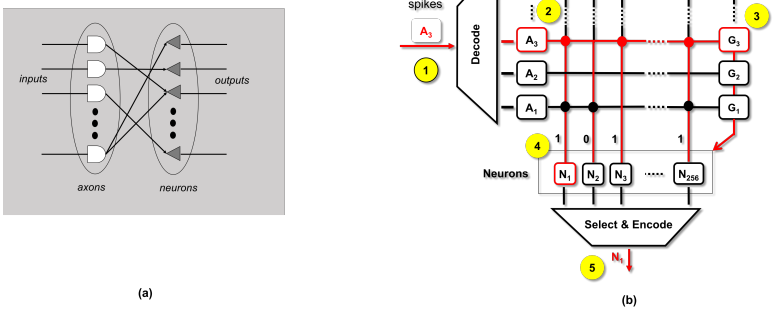

Fig. 1. A higher level abstraction of Truenorth [8] that shows, (a) Axons serve as inputs to the core. Each axon can be connected to 256 neurons. (b) Synaptic connections are programmable on a core with a weight value associated to each connection.

IV. Implementation

The implemented hopfield neural network based linear solver has a similar architecture as the one presented in [9]. To represent the values of input matrix B, stochastic based rate coding technique is chosen. This coding scheme states that probability of occurrence of a spike at tick t is directly proportional to input value.The recurrent hopfield neural network weight matrices W_{ff} and W_{hop} are encoded using the neuron parameters provided on TrueNorth.

Algorithm 1 Computes the weights and threshold values for performing dot product on TrueNorth

Input: Floating point values in the i^{th} row of weight matrices $(W_{i,.})$

Output: Assigned TrueNorth weight and threshold

1: **procedure** WeightThresholdAssignment
2: $Threshold = Round(\frac{255}{\max_i(|W_{i,.}|)})$
3: $Weights = Round(\frac{255}{\max_i(|W_{i,.}|)} \times W_{i,.})$
4: **end procedure**

To perform matrix multiplication with weight matrices W_{ff} and W_{hop}, the floating point values of these two weight matrices are encoded as a ratio of weights to thresholds. Here, a single synapse is used for each term in the dot product computation. In TrueNorth, each neuron can have up to four axon types as input, each of which can be assigned a unique synaptic weight. Figure 2 shows the synaptic connections in TrueNorth that implement a dot product between $[H_k(1,1)H_k(2,1)H_k(3,1)]$ with the values in 3×3 weight matrix W_{hop} (Which can have values either in positive domain or negative domain). Each of the three values in H_k have been assigned a different axon type so that they are multiplied with the corresponding weight value to compute the dot product $w_1H_k(1,1) + w_2H_k(2,1) + w_3H_k(3,1)$.

Algorithm 1 summarizes the procedure to assign weights and threshold values on TrueNorth hardware. In the proposed implementation, a single synapse per weight is being used for computation. First, it is important to separate the matrices W_{ff} and W_{hop} into positive and negative planes, since the values

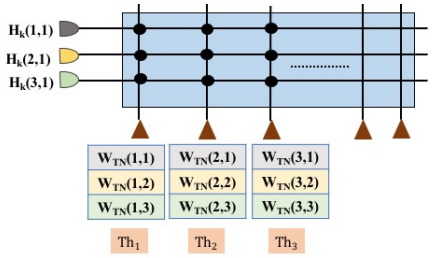

Fig. 2. Synapse connections showing the dot product between first column of H_k with the weight matrix W_{hop} and the corresponding threshold values for each neuron.

in these two domains will be dealt with separately. Next, for encoding the values on the board, we want to maximally utilize the 8 bit neuron weights of TrueNorth, therefore, the maximum absolute value in $W_{i,.}$ is assigned the weight of 255. The threshold value for this neuron would be $Round(^{255}/_{max(|W_{i,.}|)})$ and the corresponding neuron weights for all of the other values would be $Round(^{255}/_{max(|W_{i,.}|)} * W_{i,.})$.

The other method to perform multiplication between matrices W_{hop}, W_{ff} and B is by using a multiplication neuron. IBM had shown in [12] that TrueNorth neurons can operate as arithmetic operation units when the input values are represented as rate coded spikes. If the weights of TrueNorth neurons are set appropriately, then they can also operate as multipliers. This implementation technique is especially useful for online matrix inverse computations, where matrix A might change dynamically after every interval. But the downside of such a scheme is that since the weights are represented as spikes, it would take longer duration for the solution to converge to a solution. Section VI shows how the computing efficiency gets affected when arithmetic operations are done using TrueNorth neurons.

Once the weights have been assigned onto TrueNorth it is critical that the values of input matrix B are scaled in such a way that the intermediate computations should not saturate. Thus, Equation 10 would have to be re-written as follows to account for the scaling factor η.

$$X = \eta \sum_{k=0}^{\infty} (I - \alpha A^T A)^k \alpha A^T \frac{B}{\eta} \\ = \eta \sum_{k=0}^{\infty} (I - \alpha A^T A)^k \alpha A^T B_n \tag{12}$$

For clarity, the scaled partial summations of Equation 12, is expressed as the term H_{j+1} in equations 13 and 14. Here, $B_n = {}^{B}/_{\eta}$, and as per the Hopfield linear solver presented in [9], $W_{hop} = (I - \alpha A^T A)$ and $W_{ff} = \alpha A^T$.

$$H_{j+1} = \sum_{k=0}^{j} (W_{hop})^k W_{ff} \frac{B}{\eta} \tag{13}$$

$$H_{j+1} = \sum_{k=0}^{j} (W_{hop})^k W_{ff} B_n \tag{14}$$

Figure 3 presents the importance of selecting a correct scaling factor to represent multiple values in an input vector or an input matrix. In the example shown three values are given as inputs to TrueNorth (2, 4 and 5) and later they are represented as spikes. In fig 3 (a), all three values have been scaled by the element that has the maximum magnitude (5 in the shown example). Thus, all three values can be represented with available range of spiking rate. Figure 3 (b) presents an example when the inputs are scaled by a value that is smaller than the maximum magnitude. As a result, two elements (viz., 4 and 5) are being represented by the same spike rate. Choosing the correct scaling factor becomes even more important when doing arithmetic operations using spikes. Example presented in figure 3 (c), performs addition between two values when they are represented as spikes. If the proper scaling factor is not selected then the resulting addition value will saturate at 1. Since the proposed algorithm that is implemented on TrueNorth is an iterative algorithm, an incorrect intermediate value might serve is an input to next set of iterations, as a result, the algorithm will converge to incorrect output values. Hence, it is imperative to have a scaling factor which guarantees that intermediate computations will never saturate.

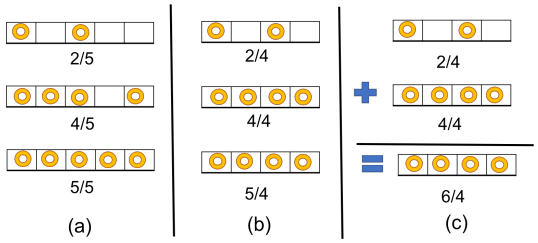

Fig. 3. Example images that illustrate the importance of proper scaling for spike based computation. (a) This image shows the example when all three values of a vector are scaled properly. (b) This image shows the example when all three values of a vector are not scaled appropriately. As a result, two values, 4 and 5, are being represented by the same spike rate. (c) In this example addition between two values when the spike values are not scaled properly. If the proper scaling factor is not selected then the resulting addition value will saturate to an incorrect value.

Algorithm 2 summarizes the recurrent linear equation solver. Since rate-coded neurons can only represent values in the range 0 to 1, the computations in the Hopfield linear solver are divided into positive and negative planes. Algorithm 2 performs the iterative computation that can be modeled using a recurrent neural network connection. As stated earlier, this iterative computation is similar to performing a summation of geometric series and after sufficient iterations the equation will converge to a solution.

Algorithm 2 Compute the transformation matrix that will map initial set of features (B) to the current set of observed input features (A). This recurrent algorithm was proposed in [9]

Input: Currently observed set of features (after proper scaling), $B_n = {}^{B}/_{\eta}$

Output: Transformation matrix that will map matrix A to matrix B

procedure HOPFIELDSOLVER
  while $\delta \geq$ Mininum Error **do**
    $H_{j+1} = W_{ff} B_n + W_{hop} H_j$
    $\delta = \|H_{j+1} - H_j\|$
    $j = j + 1$
  end while
  $X = H_{\infty} \eta$
end procedure

Figures 4 (a) and 4(b) show the end-to-end flow of the proposed Hopfield linear solver for image tracking. Figure 4(a) shows the flow of how the neural network weights are set for computation based on the image template. Figure 4(b) shows the computation flow for estimating affine transformation matrix, using the computed weights and extracted features of the new image template.

V. EXPERIMENTAL SETUP

This section describes the experimental setup for applications that were tested using TrueNorth based linear hopfield solver.

A. Target tracking

The first class of application that is considered is a typical target tracking scenario, shown in Fig. 5(a). These examples are widely popular in drone control, image localization and image tracking. Usually for such applications the image or target is predefined, that is the shape of the object is known beforehand and the goal is to be able to continuously monitor the motion of the said object by placing a bounding box around it.

In the proposed example, a real-time video input is preprocessed to extract features (e.g. edges of particular orientations) to form a feature set. This feature set is then compared against a set of templates to identify objects of interest, with the goal of tracking the objects in the image frame as they move in three dimensions. As a proof of concept, a very simple image was drawn whose feature set consists of just three edges similar in appearance to the letter H. The height and width of this symbol

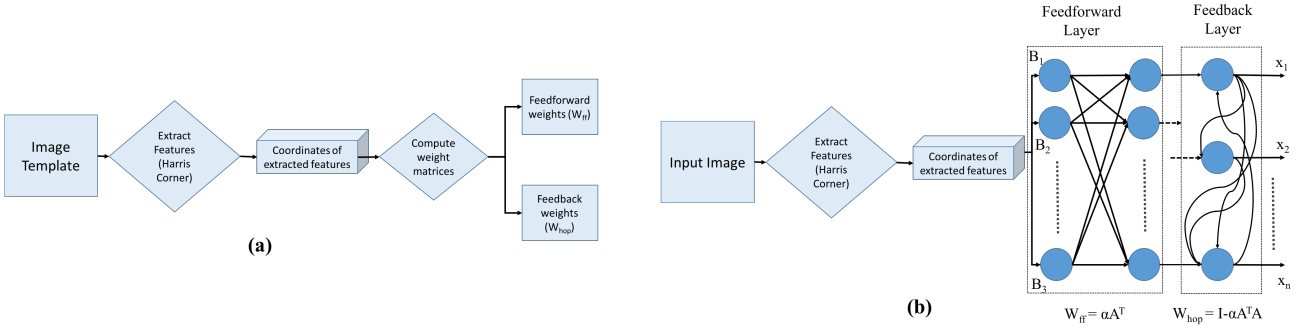

Fig. 4. (a) End-to-end flow of how the Hopfield synapse weights are set for computation based on the image template. (b) Computation flow for estimating affine transformation matrix, using the computed weights and extracted features of the new image template.

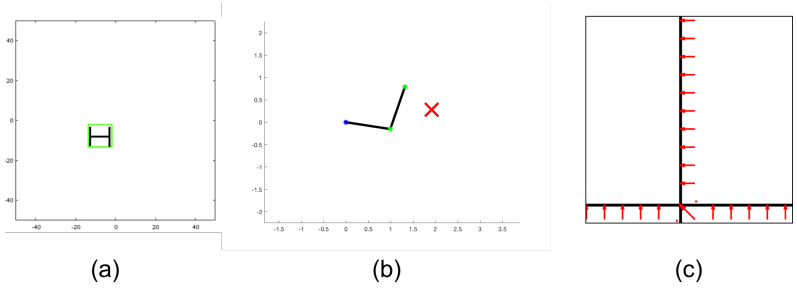

Fig. 5. (a) Screenshot that illustrates target tracking application. (b) Inverse kinematics experiment. (c) Optical flow application screenshot

is 1 cms, each. The position of the image can vary from -15 cms to 15 cms along vertical and horizontal directions and the scale can change by 0.5, 0.25, 1, 2, and 4 times from the previous size.

Equations 15 and 16 show the structure of matrices A and B that contains x and y coordinates of the features. To determine size and placement of the bounding box for the tracked image, the theory of affine transforms [13] is used which states that a current image B can be matched to its template A with an affine transformation X using a matrix multiplication $AX = B$, as long as the image has only been transformed with respect to that template in scale, rotation, or 2D translation (a similar self-learning visual architecture was investigated in [14]). By employing matrix division implemented in a recurrent Hopfield network, the affine transform X can be derived that maps the current image input B to the template A, and can determine the scale, horizontal, and vertical transformations from the corresponding entries in matrix X.

$$A = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix} \tag{15}$$

$$B = \begin{bmatrix} \widehat{x_1} & \widehat{y_1} & 1 \\ \widehat{x_2} & \widehat{y_2} & 1 \\ \widehat{x_3} & \widehat{y_3} & 1 \end{bmatrix} \tag{16}$$

B. Inverse Kinematics

The second class of application that is considered is inverse kinematics, specifically the two joint arm problem as shown in fig. 5(b). In this algorithm the objective is to move the two joints of an arm until the end effector is close to the target position (shown as a bold 'X' symbol in the image).

Eqn. 17 shows the equation for two-joint arm based inverse kinematics equation for which the hopfield network based linear solver was used. Equations 18 and 19 show A and B matrices, respectively. For the purpose of these experiments the arm lengths, viz., L_1 and L_2 have been selected as unit lengths or 1 cms. The target can appear at any spot within a radius of 2 cms from the origin. When the target changed its position, the arm would start moving from the position it stopped at in the previous iteration, that is, the position of end-effector would start from where it previously left-off.

This demonstration shows how a linear solver can be used in inverse kinematics based applications. The coordinates of arm 2 (represented by green markers in fig. 5(b)) can be extracted using a feature extraction technique. As the length of arms have been set to unit length, the values in matrix A will all have a magnitude of less than 1. When the arm moves from one position to another, the matrix Θ_{k+1} gets updated in every step. Based on the new position of arms, the value of matrices A and B are computed again so that they can be later used for the next step.

$$\Theta_{k+1} = \Theta_k - (LearningRate)A^{\dagger}B \tag{17}$$

$$A = \begin{bmatrix} -L_1 sin(\theta_{k,1}) & -L_2 sin(\theta_{k,2}) \\ L_1 cos(\theta_{k,1}) & L_2 cos(\theta_{k,2}) \end{bmatrix} \tag{18}$$

$$B = \begin{bmatrix} x_{target} - x_{current} \\ y_{target} - y_{current} \end{bmatrix} \tag{19}$$

C. Optical flow

Finally, the third class of application which we considered is optical flow, shown in fig. 5(c). Optical flow is another popular algorithm that is used in applications such as drone control and object tracking. Unlike the previously described target tracking algorithm, there is no need for feature extraction, rather the computations are carried out by calculating the Jacobian between pixel intensities across different frames, as a result, significantly reducing the computation time. The main underlying principle of this algorithm is that for a short duration of time there is no change in pixel intensity. Optical flow outputs direction and speed by which an observed object is moving.

To compute the velocities of optical flow, the Lucas-Kanade algorithm was used. Equations 20 and 21 show matrices A and B, respectively. $I_x(qi)$, $I_y(qi)$ and $I_t(qi)$ represent derivatives across x direction, y direction and time, respectively, around pixel qi. For this experiment, we assume a window of size 5-by-5 pixels.

The presented optical flow demonstration is similar to the one reported in [15] where the horizontal bar is continuously moving upwards and the vertical bars are moving to the left of the screen. In the developed prototype it is demonstrated that the direction of the bar's movement can be reported without any error and also the approximate speed by which the two bar's move can be reported. The thickness of the two lines was selected to be 3 cms and the speed of the bar's movements was set to 12 cms per second. The size of the frame that was selected to draw these lines is 50 cm by 50 cm. The velocity of the two line's movement is calculated by solving for X in the equation $AX = B$, where matrix A contains partial derivatives of initial image frame with respect to directions x and y, and matrix B contains partial derivatives of pixel positions between initial image frame and image frame at time t. After implementing matrix division, output matrix X will report the speed and direction of the image pixels, by computing the pseudoinverse of matrix A.

$$A = \begin{bmatrix} I_x(q1) & I_y(q1) \\ I_x(q2) & I_y(q2) \\ \vdots & \vdots \\ I_x(qn) & I_y(qn) \end{bmatrix} \tag{20}$$

$$B = \begin{bmatrix} -I_t(q1) \\ -I_t(q2) \\ \vdots \\ -I_t(qn) \end{bmatrix} \tag{21}$$

VI. Results and Discussion

This section presents the results and discussions for the applications that were tested on TrueNorth using hopfield linear solver. Table I shows the amount of hardware that was utilized to map the relevant hopfield linear solver for the corresponding applications. Note that, optical flow requires considerable number of more neurons than the other two applications. This is because for optical flow, rather than having hard-coded weights on TrueNorth, we chose to implement matrix multiplication by using neurons as multipliers. Since one neuron is being used to multiply two spike based inputs, the amount of required hardware grows quickly. We report the relative error (eqn. 22) and absolute error (eqn. 23) between the output that is obtained from the TrueNorth-based Hopfield linear solver and the output that is obtained using MATLAB's double precision pseudoinverse library function. A lower absolute error would be preferred as it would indicate how close is the predicted value from the ideal value in terms of precision. The result tables also summarize the number of ticks it took to compute the generalized inverse on TrueNorth.

TABLE I
TrueNorth hardware utilization

Application	Number of neurons used	Number of cores	Amount of on-chip cores utilized (in %)
Target tracking	288	2	0.05
Inverse kinematics (Hard-coded weights)	288	2	0.05
Inverse kinematics (Spiking weights)	160	11	0.2
Optical flow	712	11	0.2

$$\delta_{relative} = \frac{2 * (Output_{experiment} - Output_{actual})}{|Output_{experiment}| + |Output_{actual}|} \tag{22}$$

$$\delta_{absolute} = |(Output_{experiment} - Output_{actual})| \tag{23}$$

A. Target Tracking

In this section the precision errors for target tracking application is reported. Matrix A was fixed during the entire simulation, but matrix B changed its values randomly. The simulations were done for 100 different random feature values of matrix B. Table II presents the error that is seen for image scaling parameter, specifically the height and width of the bounding box, table III reports the error in calculated x-coordinates of bounding box and lastly, table IV reports the error in computed y-coordinates of bounding box. It can be inferred from the reported results that if the computations are carried out for longer duration of time ticks, the error reduces.

Note there is one anomaly in the results and that is for computing x-coordinate of bounding box after the simulation runs for 10,000 clock ticks. The relative error for this case increases instead of decreasing. This happens because for the cases where ideal x-coordinate is supposed to be 0, the hopfield solver computes a value of 0.02 or 0.03, as a result, showing a relative of 200% or 300% for just these cases. Therefore, the

TABLE II
ERROR IN REPORTING BOUNDING BOX SCALE (WIDTH AND HEIGHT)

Number of clock ticks	Mean relative error (in %)	Standard deviation of relative error (in %)	Mean absolute error(in cms)	Standard deviation absolute error(in cms)
3000	10.67	18.94	0.1647	0.1943
5000	4.14	9.89	0.0825	0.13
10000	2.96	4.78	0.0736	0.098

TABLE III
ERROR IN REPORTING BOUNDING BOX HORIZONTAL POSITION (x-COORDINATE)

Number of clock ticks	Mean relative error (in %)	Standard deviation of relative error (in %)	Mean absolute error(in cms)	Standard deviation absolute error(in cms)
3000	5.92	21.26	0.1039	0.102
5000	1.13	2.09	0.049	0.0559
10000	6.88	34.16	0.039	0.0379

TABLE IV
ERROR IN REPORTING BOUNDING BOX VERTICAL POSITION (y-COORDINATE)

Number of clock ticks	Mean relative error (in %)	Standard deviation of relative error (in %)	Mean absolute error(in cms)	Standard deviation absolute error(in cms)
3000	1.72	4.83	0.0922	0.0960
5000	1.69	3.23	0.052	0.0703
10000	0.74	2.29	0.0443	0.0392

reported relative error increases, but the absolute error is still small.

B. Inverse kinematics

To evaluate the results of experimental setup for inverse kinematics, we chose to setup two different implementation schemes. In the first implementation technique the feedforward and recurrent weights hopfield linear solver are hard-coded on TrueNorth, whereas in the second implementation technique the weights were represented as spikes.

1) Hopfield network weights hard-coded on TrueNorth: Similar to target tracking application, this experimental setup was tested for 100 different random positions of the arm. The simulations were done for 60,000 clock ticks for each step. Unlike target tracking, here the computations have to be done on high-precision values, as a result, the algorithm takes longer to converge and requires a larger time window to represent very small values as spikes. Also, the matrices A and B change for every set of simulations that were conducted. Table V summarizes the results that were obtained following the experiments with inverse kinematics setup.

Apart from the results shown in table V, we note that the Euclidean distance between the end effector position of the arm and the intended target position has exactly the same value when compared between MATLAB's pseudoinverse function and the Hopfield linear solver. That is, despite using a approximate computing technique with several sources of intermediate error, the effector always reaches the correct position irrespective of whether the computations were done using Hopfield solver or they were done with double precision MATLAB pseudoinverse function. These values were checked for 60 different target positions.

TABLE V
ERROR IN REPORTING END EFFECTOR POSITIONS

Attribute	Mean relative error (in %)	Standard deviation of relative error (in %)	Mean absolute error (in cms)	Standard deviation absolute error (in cms)
Horizontal position of end effector	8.38	34.32	0.0886	0.076
Vertical position of end effector	49.39	83.1	0.357	0.3

2) Hopfield network weights encoded as spikes: This experimental setup is similar to optical flow experiment and it was tested for 100 different random positions of the arm. The simulations were done for over 3 million clock ticks for each step. As the hopfield solver weights are represented as spikes, it requires longer clock ticks to reach an answer that is close enough to the expected result.

In addition to the results shown in table VI, we note that the Euclidean distance between the end effector position of the arm and the intended target position has an average error of 0.0069 cms when compared between MATLAB's pseudoinverse function and the Hopfield linear solver. These values were checked for 20 different target positions. Even though the results show high relative error for vertical position of robotic arm, the end effector reaches very close to the intended position (an average difference of 0.0069 units) irrespective of whether the computations are done using hopfield linear solver or MATLAB's pseudoinverse function.

TABLE VI
ERROR IN REPORTING END EFFECTOR POSITIONS

Attribute	Mean relative error (in %)	Standard deviation of relative error (in %)	Mean absolute error (in cms)	Standard deviation absolute error (in cms)
Horizontal position of end effector	10.81	35.31	0.1263	0.3799
Vertical position of end effector	137.67	82.42	0.0978	0.1885

C. Optical flow

Experiments for optical flow application differ from the previous two applications. As the weight representation scheme chosen for these experiments is stochastically-code spike based, the simulations need to run for a considerably longer duration to converge to a solution. The purpose for such a computation scheme is that the proposed hopfield linear model has the potential to be used as an online pseudoinverse calculator where the TrueNorth neurons could be used as arithmetic operation units. Table VII summarizes the precision error that were obtained when the optical flow matrices were simulated for 3 million clock ticks for 100 different randomly chosen values from frame number 1 and frame number 12. While selecting the matrices A and B we ensured that the ideal output after matrix inverse should not have both the velocities as 0. Otherwise, the results would report incorrect observations for regions where there is no motion present. Similar to inverse kinematics testing, the A and B matrices for this application were tested in every set of simulations that were carried out. The results never gave the wrong prediction in terms of direction of motion or the velocity. In fact, the signs converged to correct values very early during the simulations. Table VII summarizes the results that were obtained following the experiments with optical flow setup.

TABLE VII
ERROR IN REPORTING MAGNITUDE OF VELOCITIES FOR OPTICAL FLOW

Attribute	Mean relative error (in %)	Standard deviation of relative error (in %)	Mean absolute error(in cms/sec)	Standard deviation absolute error(in cms/sec)
Magnitude of horizontal velocity	18.39	36.56	0.1649	0.5485
Magnitude of vertical velocity	7.65	18.27	0.2057	1.0

VII. Conclusions and Future Work

To the best of our knowledge, this paper is the first attempt to implement a generalized matrix inverse calculator on a limited precision neural substrate. We validate the mathematical model using a Hopfield network-based linear solver that has been implemented on the IBM TrueNorth spiking neural substrate. Our empirical results show that it is plausible to have a Hopfield linear solver that can do on-chip computations of pseudoinverse without off-loading these computation intensive calculations to a CPU. Though the mathematical operations take a relatively long time to reach a solution, considering the amount of nominal resources that this Hopfield linear solver consumes there is high possibility that such algorithms would perform very well on substrates that operate at much higher frequency when compared with 1000 Hz frequency of TrueNorth. Large scaling factors (described as parameter β) may end up resulting in consuming more resources and/or longer computation time, since we end up requiring a greater number of ticks to represent the values accurately. Thus, having a framework where we can optimize the value of this scaling factor parameter is very important.

In future work we will look into speeding up the computation by using a population coding scheme for encoding values to spikes. Currently our implementation uses a rate coding technique for encoding values with a single neuron. Considering the resources that we have available on TrueNorth board, we could use population coding scheme that would do the computations in parallel. We expect that the extensions to our mathematical framework to encompass population coding will be straightforward. In contrast, determining an optimal assignment of population resources to jointly minimize area, energy, and computation time will be a difficult challenge.

As part of our future work we would like to develop a thorough mathematical framework and rigorous analysis of the computations that are happening in the recurrent pathway.

Acknowledgment

This work was supported in part by NSF grant CCF-1628384. Mikko Lipasti has a financial interest in Thalchemy Corp. Authors would like to thank IBM SyNAPSE team for providing the software and hardware platforms for TrueNorth.

References

[1] M. Riesenhuber and T. Poggio, "Hierarchical models of object recognition in cortex," *Nature Neuroscience*, vol. 2, pp. 1019–1025, 1999.
[2] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," in *Biological Cybernetics*, vol. 36, 1980, pp. 193–202.
[3] D. W. Arathorn, *Map-Seeking Circuits in Visual Cognition: A Computational Mechanism for Biological and Machine Vision*. Stanford, CA, USA: Stanford University Press, 2002.
[4] E. Rolls, "Invariant visual object and face recognition: Neural and computational bases, and a model, visnet," *Frontiers in Computational Neuroscience*, vol. 35, Jun 2012.
[5] C. Mead, "Neuromorphic electronic systems," *Proceedings of the IEEE*, vol. 78, pp. 1629–1636, 1990.
[6] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, and et. al., "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.
[7] Y. Mroueh, S. Voinea, and T. Poggio, "Learning with group invariant features: A kernel perspective," in *Advances in Neural Information Processing Symposium*, 2015.
[8] A. Nere, "Computing with hierarchical attractors of spiking neurons," Ph.D. dissertation, University of Wisconsin - Madison, 2013.
[9] G. Lendaris, K. Mathia, and R. Saeks, "Linear hopfield networks and constrained optimization." *IEEE Trans. Systems, Man and Cybernetics, Part B*, vol. 91, pp. 114–118, Feb 1999.
[10] A. Ben-Israel and A. Charnes, "Contributions to the theory of generalized inverses," *J. Soc. Indust. Appl. Math.*, vol. 11, no. 3, pp. 55–60, 1963.
[11] P. J. Olver, "Numerical analysis lecture notes," 2017.
[12] A. S. Cassidy, P. Merolla, J. V. Arthur, S. K. Esser, and et. al., "Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*, Aug 2013, pp. 1–10.
[13] "https://en.wikipedia.org/wiki/Affine_transformation," 2016.
[14] R. Shukla and M. Lipasti, "A self-learning map-seeking circuit for visual object recognition," in *2015 International Joint Conference on Neural Networks (IJCNN)*, July 2015, pp. 1–8.
[15] S. K. Esser, A. Andreopoulos, R. Appuswamy, and et. al., "Cognitive computing systems: Algorithms and applications for networks of neurosynaptic cores," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*, Aug 2013, pp. 1–10.