

Weight Programming in DNN Analog Hardware Accelerators in the Presence of NVM Variability

Charles Mackin,* Hsinyu Tsai, Stefano Ambrogio, Pritish Narayanan, An Chen, and Geoffrey W. Burr

Crossbar arrays of nonvolatile memory (NVM) can potentially accelerate development of deep neural networks (DNNs) by implementing crucial multiply–accumulate (MAC) operations at the location of data. Effective weight-programming procedures can both minimize the performance impact during training and reduce the down time for inference, where new parameter sets may need to be loaded. Simultaneous weight programming along an entire dimension (e.g., row or column) of a crossbar array in the context of forward inference and training is shown to be important. A framework for determining the optimal hardware conditions in which to program weights is provided, and its efficacy in the presence of considerable NVM variability is explored through simulations. This strategy is shown capable of programming 98–99% of weights effectively, in a manner that is both largely independent of the target weight distribution and highly tolerant to variability in NVM conductance-versus-pulse characteristics. The probability that a weight fails to reach its target value, termed P_{fail} , is quantified and the fundamental trade-off between P_{fail} and weight programming speed is explored. Lastly, the impact of imperfectly programmed weights on DNN test accuracies is examined for various networks, including multi-layer perceptrons (MLPs) and long short-term memory (LSTM).

1. Introduction

The confluence of large datasets and advancements in graphics processing units (GPUs) has spurred remarkable advances in deep neural networks (DNNs) in recent years.^[1] DNNs are now routinely used in image classification, speech recognition, and natural language processing. In some instances, the capabilities of these systems have surpassed human capabilities.^[2–4] At their core, DNNs rely extensively on multiply–accumulate (MAC) operations, making them well-matched for GPUs in high-performance applications. GPUs, however, use “stored program” or von-Neumann architectures, meaning memory blocks are physically separated from computational blocks. This exacts


large and unavoidable time and energy penalties for data transport between memory and computational blocks—the so-called “von Neumann” bottleneck.

Analog hardware accelerators for deep learning can avoid this bottleneck by performing MAC operations in memory using a crossbar array structure.^[5–7] In these crossbar arrays, nonvolatile memory (NVM) elements are used to encode synaptic weights. This was recently shown capable of 280× speedup in per area throughput while also providing 100× enhancement in per area energy efficiency over state-of-the-art GPUs.^[8] While the benefits of speed are readily appreciated, enhancement in energy efficiency can be a powerful driver of purchasing decisions in the data center space as well.^[9–12]

Crossbar arrays have been implemented using a variety of analog NVM elements, including resistive RAM (ReRAM),^[13,14] conductive-bridging RAM (CBRAM),^[15] flash,^[16–20] and phase-change memory (PCM).^[21,22] However, most NVM device candidates exhibit varying extents of non-

ideal behavior, including limited resistance contrast, significant nonlinearity in conductance change, as well as strong asymmetry in bidirectional programming. At the application level, this manifests into neural network accuracies that are significantly lower than software-based approaches. Many of these nonidealities, however, can be addressed to an extent through device and circuit-level engineering. One key idea is multiple conductances of varying significance (**Figure 1a**), where each synaptic weight is distributed across at least two conductance pairs using $W = F(G^+ - G^-) + (g^+ - g^-)$. For instance, during training, such a PCM cell could be paired with a capacitive cell to combine complementary features of both and relax the overall device requirements on PCM.^[8] In such a scheme, the majority of DNN weight tuning can occur on a linear but volatile 3T1C memory structure, with periodic but infrequent transfer to the PCM cell for nonvolatile storage. Row- (or column-)wise weight transfer also arises in ex situ training and other training variants. Since PCM is subject to the previously mentioned nonidealities, this weight transfer is best performed with a closed-loop iterative tuning procedure with multiple write-plus-read-verify steps to achieve the overall target weight. To avoid performance degradation, particularly during training, NVM programming

Dr. C. Mackin, Dr. H. Tsai, Dr. S. Ambrogio, Dr. P. Narayanan, Dr. A. Chen, Dr. G. W. Burr
IBM Research–Almaden
650 Harry Road, San Jose, CA 95120, USA
E-mail: charles.mackin@ibm.com

 The ORCID identification number(s) for the author(s) of this article can be found under <https://doi.org/10.1002/aelm.201900026>.

DOI: 10.1002/aelm.201900026

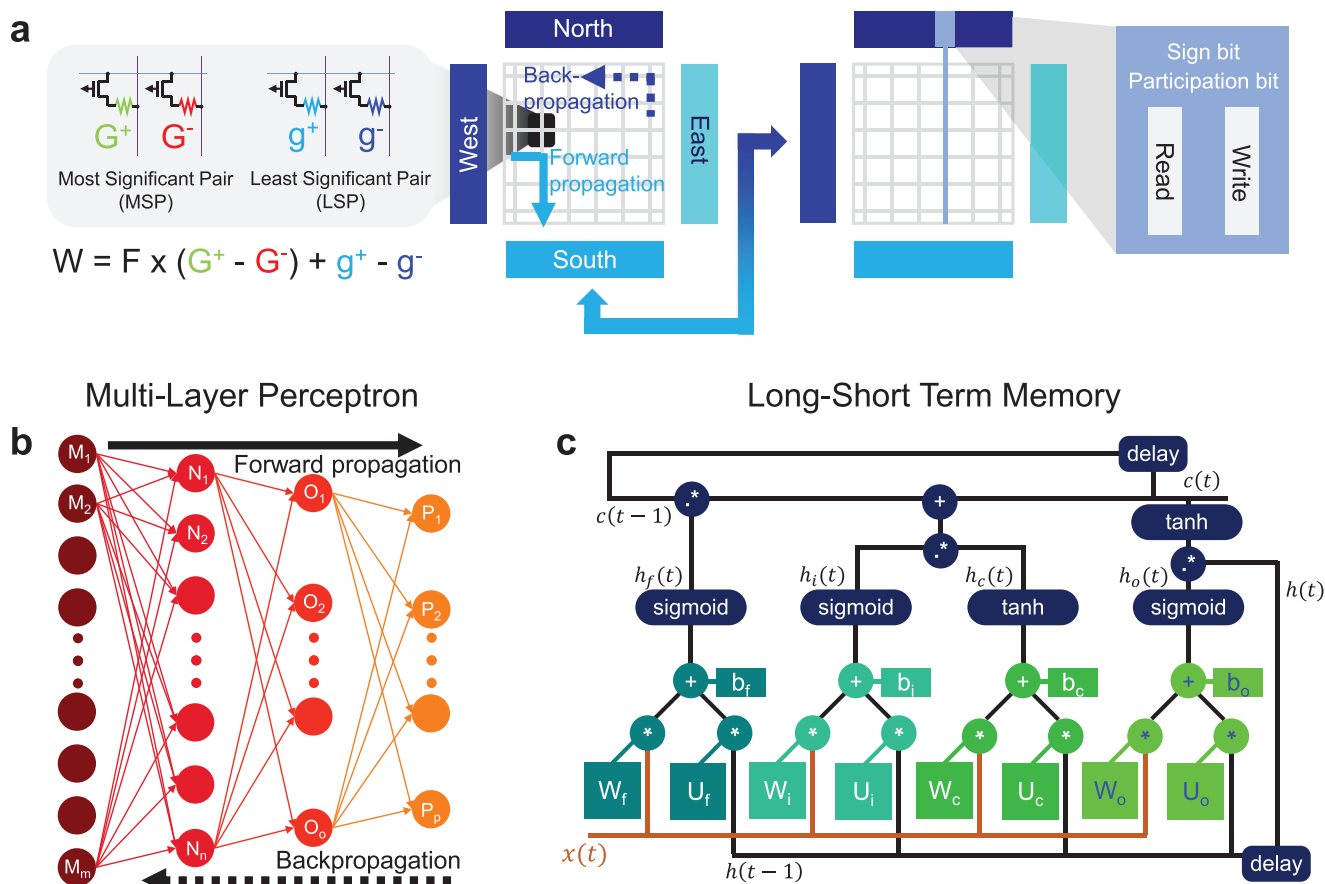


Figure 1. a) Individual synaptic weights are encoded using differential NVM conductance pairs of varying significance and networks are mapped to multiple blocks of crossbar arrays with west and south side peripheral circuitry to handle input and output activations, respectively. Dedicated per-column programming circuitry is located on the north side with row-decode circuitry located on the east side. The impact of weight nonidealities is studied for both b) multilayer perceptron (MLP) and c) long short-term memory (LSTM) networks.

needs to be as fast and effective as possible. Similarly, in an NVM-based DNN forward inference chip, one could envision a need to load a new set of weights (i.e., a new DNN model) to implement new tasks or improve on existing ones. Once again, fast and effective programming of NVM devices is of utmost importance in reducing down-time.

One approach to weight programming is to perform simultaneous weight programming along an entire dimension of the crossbar array (e.g., either column-wise or row-wise). Several previous works acknowledge the importance of parallelized weight programming, which is nontrivial in the presence of significant variability in NVM conductance versus pulse characteristics.^[23–27] For instance, if every device, say along a row, were identical and perfectly linear in conductance versus number of pulses, then it is fairly straightforward to determine the number of pulses required on G^+ , G^- , g^+ , and g^- to program the set of target weights. Per-column circuitry could be used to store this information, and then to halt programming once the required number of pulses have been fired for each device. In reality, however, each device along a row may require very different pulse conditions to reach a particular conductance value. Furthermore, given cycle-to-cycle variations and different maximum conductance values, how each weight might end up

distributed across its multiple conductance components cannot be known a priori. This motivates the development of strategies that can minimize programming effort, while providing sufficient tuning precision to achieve software-equivalent accuracies for both training and inference. This is the primary focus of this paper.

For instance, to achieve a positive net weight, one could aim to use the G^+ conductance to reach a threshold value “just below” the target weight, and implement the residual portion on the g^+ conductance. Alternatively, one may choose to program a value “just above” the target weight using G^+ , and implement the residual portion on g^- . As these programming strategies would all involve closed-loop tuning, each column would require both write and read heads. (For clarity only, the subsequent discussion focuses only on row-wise programming, but extensions to column-wise programming or even other N-wise subsets within an N-by-M array should be straightforward.) For area-efficiency, the read circuitry is assumed to be a simple comparator capable of estimating if the measured value is above or below a reference value. Additionally, each column would need to preserve its particular target weight (say in a digital register or an analog capacitor), the currently programmed value of the weight, a sign bit s to denote if the

current value of the weight is above or below its target value, and a participation bit p to determine whether the particular column should continue to participate in the present “phase” of the row-wise programming procedure. Assuming that DNN input and output circuitry resides on the west and south sides of the array, respectively, the dedicated programming column circuitry could be located on the north side, with row decode circuitry on the east side, as shown in Figure 1a.

Given the imperfections in the NVMs and the limited number of programming attempts possible, there exists some nonzero probability, P_{fail} , that the error in the final weight fails to make it inside some acceptable error margin around the final target weight. This paper quantifies P_{fail} given one particular weight- and conductance-programming strategy, and describes the relationship between P_{fail} and programming speed as controlled by choice of the gain factor F and other parameters. A “Jump Table” construct^[28] is used to accurately model the conductance evolution of the NVM devices, using normally distributed random variables to introduce both cycle-to-cycle and device-to-device variability in the size of conductance jumps, as well as device-to-device variability in maximum conductance. The impact on the final accuracy that imperfectly programmed weights might produce in neural networks such as multilayer perceptrons (MLPs) and long short-term memory (LSTMs) is also studied. Figure 1b shows the 3-layer MLP used in this study, with neuron layer sizes of 784-250-125-10 and unbounded ReLU as the activation function between neuron layers, pretrained to high accuracy on the MNIST dataset of handwritten digits.^[29] The fundamental building block for our LSTM model is shown in Figure 1c, where the input vector $x(t)$ and a hidden state vector from the previous time step $h(t-1)$ undergo vector–matrix multiplications to obtain four vectors in an LSTM: the forget gate $h_f(t)$, input gate $h_i(t)$, the output gate $h_o(t)$, and the new context vector $h_c(t)$. The new hidden state vector $h(t)$ is calculated based on Equation (1)

$$\begin{aligned} c(t) &= h_f(t) * c(t-1) + h_i(t) * h_c(t) \\ h(t) &= h_o(t) * \tanh[c(t)] \end{aligned} \quad (1)$$

where $c(t)$, the context vector, is another hidden vector that represents the long-term memory of the LSTM network. The hidden state vector $h(t)$ is then used to generate the output at each time step t through vector–matrix multiplication with an output matrix W_y (not shown). In this study, a character-based language modeling task is chosen for performance evaluation. The model consists of a fully connected embedding layer that converts the ASCII-encoded character (i.e., a one-hot vector of size 256) to an embedding vector x of size 50, then two layers of LSTM, followed by an output layer. To prepare the base-line set of high-quality weights, training was performed on 25-character sequences with each “time step” representing a character’s position within the sequence. Performance of the model is quantified using cross-entropy loss (i.e., negative log likelihood of the ground truth character), with lower loss indicating better performance. The novel Alice in Wonderland, which possesses a total of 150 000 characters, is used as the dataset for the LSTM model with 90% and 10% of the text used for training and test datasets, respectively. The desired objective of weight programming into the modeled NVM conductances is to maintain the

high MLP accuracy and low LSTM cross-entropy loss of the models trained off-line, despite the presence of both cycle-to-cycle and device-to-device variability in the conductances used for weight programming.

The rest of the paper is organized as follows. Section 2 describes the NVM device model and variability. Section 3 details the row-wise NVM programming strategy. Section 4 provides a framework for determining the optimal weight programming conditions in hardware through the minimization of P_{fail} . Section 5 examines the efficacy of the devised weight programming strategy in the presence of NVM variability. Section 6 extends the analysis of hardware programmed weights to neural networks, exploring correlations between target weights and programmed weights, and their impact on MLP and LSTM neural networks. Section 7 provides concluding remarks.

2. PCM Conductance Model

This work employs a previously developed statistical device model that is highly evocative of experimentally observed conductance versus pulse behavior in PCM.^[28] The model captures the “s-shaped” conductance versus pulse profile that is typical in PCM as well as interdevice (device-to-device) and intradevice (cycle-to-cycle) variations. The model, along with the effects of its key components, are depicted in Figure 2. To describe the shape of the conductance increase, a “Jump Table”^[28] approach is employed (see Figure 2a). For any given current conductance G , a Jump Table fully quantifies the probability of any feasible conductance step ΔG that occurs in response to an applied voltage pulse. Each column of the heatmap is a cumulative distribution function (CDF), designed to match the statistical cycle-to-cycle variability observed in experiment.^[30] Intradevice variability is captured through the vertical CDFs using $\sigma_{\text{intra}} = 2.5\%$, since for a given conductance G , different values of ΔG may be extracted from cycle to cycle. (Although a Gaussian distribution is used here, note that the Jump Table concept is inherently capable of implementing much more complicated statistical models as well.) Both axes are represented in percentages to provide a more generalized form of the model. This is achieved by extracting the maximum conductance G_{max} for each device, which scales both the G and ΔG axes of the Jump Table. This allows one Jump Table to apply to an array with a significant amount of device-to-device variability, including both different G_{max} values for each device, and even different S_G scaling coefficients for devices that share the same G_{max} . The latter coefficient effectively stretches the entire Jump Table vertically, so that the average number of pulses required to get from G_{min} to G_{max} can vary even between devices with exactly the same G_{max} values (Figure 2a).

Interdevice statistics were extracted based on experimental data from over 100 000 mushroom-cell 1T1R PCM devices fabricated in the 90 nm geometry. Figure 2b describes the distribution of G_{max} , which is similar to a normal distribution, but scaled and clipped so as to constrain the distribution to positive-only values. The spread is captured by $\sigma_{G_{\text{max}}} = 30.5 \mu\text{S}$, with a mean around $\mu_{G_{\text{max}}} = 50 \mu\text{S}$. Figure 2c describes the normal distribution of slope coefficients S_G , with a mean value of $\mu_{S_G} = 1$ and $\sigma_{S_G} = 0.16$. To gain a better insight into the

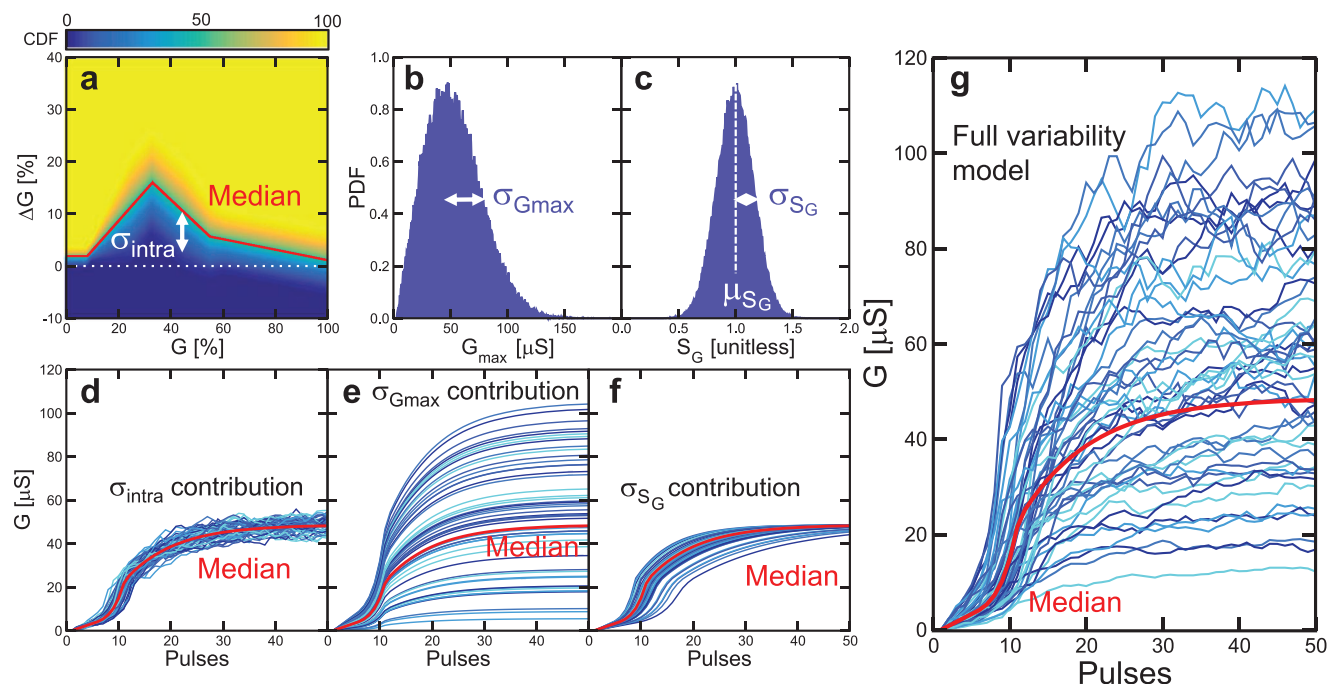


Figure 2. Description of the adopted PCM conductance versus pulse model. The “Jump Table” a) links the current G with ΔG , providing the “s-shaped” trajectory that is characteristic of incremental same-pulse programming of mushroom-cell PCM.^[28,30] Intra-device variability is captured by $\sigma_{\text{intra}} = 2.5\%$ on the vertical axis CDF, while the interdevice variability is described by distributions in maximum conductance G_{max} b) and slope S_G c). (d–f) depict fifty conductance traces to illustrate the isolated contributions of intra-device d), G_{max} e), and slope S_G f) variations. g) An example of the complete model including all variability contributions. Experimental results (b,c) were obtained using 20 ns pulses at a constant current of 40 μA for a sample size of 100 000.

different variability contributions, Figure 2d shows 50 modeled programming traces in the absence of interdevice variability, thus showing only the contributions of intra-device variability as defined by σ_{intra} . Similarly, Figure 2e,f explores the impact of $\sigma_{G_{\text{max}}}$ and σ_{S_G} , respectively, each in the absence of the other forms of variability. Conductance trajectories using the full model, with all variability contributions combined, are shown in Figure 2g.

Based on the model, it is interesting to extract the number of pulses needed to saturate a conductance to G_{max} , given an initial hard reset of $G = 0$ μS . **Figure 3** depicts the CDFs and the probability density functions (PDFs) as a function of the number of applied pulses, which range from 2 to 30. The CDFs

can be seen to saturate around ≈ 30 pulses as individual PCMs have largely reached their respective G_{max} at this point. The effects of variability are also clear from the progressive broadening of distributions, since different devices follow different conductance trajectories, producing a broadening of conductance values as the number of programming pulses increases. This model of conductance evolution is highly evocative of experimental PCM devices. However, additional considerations for real devices, such as resistance drift after programming^[31] and the possibility that circuit noise or $1/f$ noise fluctuations within the physical device might lead to erroneous verify-reads, are not included here.

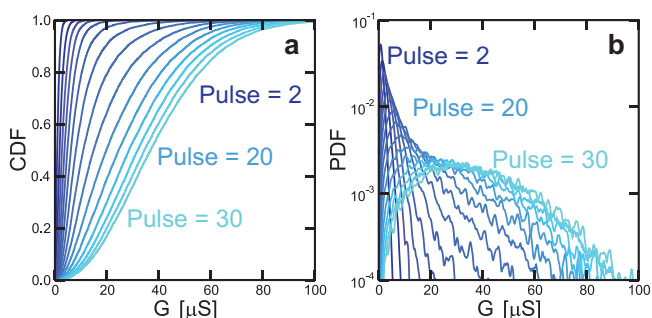


Figure 3. a) Cumulative distribution function (CDF) of PCM model conductances as a function of pulse number and b) probability density function (PDF) of PCM model conductances as a function of pulse number. The distribution of PCM conductances saturates after ≈ 30 pulses as the majority of individual PCMs have neared their G_{max} .

3. Four-Phase Weight Programming Algorithm

Variability in NVM conductance versus pulse characteristics requires weights to be programmed in a “closed-loop” fashion using some form of write-verify method. The requirement of a verify step represents an inherent disadvantage relative to write only (i.e., “open-loop”) methods. In the absence of a write only option, however, it becomes imperative to mitigate the additional time and hardware resources involved in implementing a write-verify algorithm. As a result, the programming algorithm is designed to favor simplicity in order to limit the complexity of the circuitry at the north side of the crossbar array, which contributes to the overall circuit area and power footprint. The algorithm requires the north side circuitry to store only the target weight and two control bits—one sign bit s and one participation bit p —for each weight along a row.

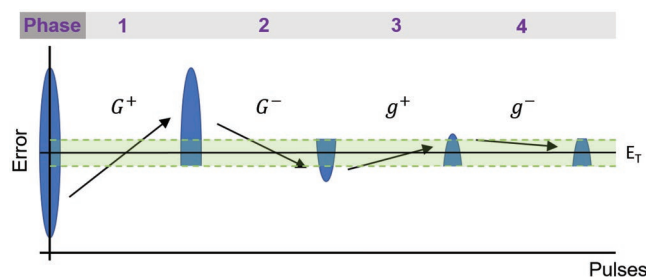


Figure 4. Four-phase algorithm used for row-wise weight programming by sequentially tuning G^+ and G^- in the MSP, followed by g^+ and g^- in the LSP. The algorithm minimizes circuit area and power costs associated with physical implementation by requiring only two bits per weight: one participation bit and one sign bit.

Weights are programmed in a row-wise fashion into NVM using the four-phase algorithm illustrated in **Figure 4**. Weights are implemented using two conductance pairs according to $W = F(G^+ - G^-) + (g^+ - g^-)$, where F represents a gain factor between the most significant pair (MSP) and least significant pair (LSP) of conductances.^[8,28] The algorithm selects a row and stores the corresponding target weights in the peripheral circuitry at the north side. Weights from the row are read out in parallel by sensing the resulting currents along the columns. The weights are then compared to their target values to calculate the participation bit p and sign bit s for each weight. The sign bit s indicates whether a weight is above or below its target value and the participation bit p indicates if the weight is within the error tolerance. A pulse is then applied to NVMs in the row based on the algorithm phase, sign bit, and participation bit. This process is repeated until all four phases of the programming algorithm have been traversed. So long as the number of write-verify steps in each phase is smaller than the number of devices along the row, the effective number of write-verify steps per device is <1 , making this algorithm an attractive and efficient choice when compared to a serialized, conductance-by-conductance write-verify approach. Once a particular row completes programming, the target weight storage and control bits in the north side circuitry are reused (i.e., overwritten) to program the subsequent row.

The initial distribution of weight errors is inversely related to the distribution of target weights, since all conductances are initially assumed hard reset at $0 \mu\text{S}$. This assumption is a slight divergence from the behavior of real PCM devices, which will typically exhibit a small nonzero conductance even when fully RESET. However, because of the very large resistance contrast of PCM devices, distributions of RESET conductance tend to be fairly tightly distributed so long as sufficient RESET programming current is available.

In phase one of the weight-programming algorithm, G^+ devices in need of programming receive pulses until the read-verify step indicates that they have exceeded their target value, or fall within a constant error tolerance E_T zone around the target value. Similarly, in phase two, G^- devices in need of programming (e.g., above the E_T zone) receive programming pulses until they fall below the target value or into the small error tolerance E_T zone around the target value. Finally, in phases three and four, similar write-verify pulse sequences are

applied to all g^+ and then all g^- pulses in need of programming. At each step, the sign and participation bits control which columns of weights should participate on any particular pulse. Any weights that fail to converge within the error tolerance by the end of the programming process are classified as weight program fails. Since the g^- device is tuned in the final phase, the algorithm sometimes terminates just after having stepped well below the target value. This makes negative weight errors more likely than positive weight errors.

Extensions of this algorithm that make use of different E_T values during different phases could readily be implemented, without incurring much additional cost in terms of circuitry. This provides added flexibility, and enables programming strategies in which varying error tolerances might be advantageous. One might imagine programming strategies in which the MSP is programmed to a more relaxed error tolerance than the LSP, in order to compensate for the amplification of errors through the gain factor F . In many cases, however, a single error tolerance value is perfectly capable of converging on the distribution of target weights, providing adequately low failure rates for any given weight configuration (e.g., the particular capabilities of its various physical conductances given device-to-device variability) and the target weight value.

4. Optimized Weight Programming

This section provides a framework for determining the optimal hardware conditions in which to program weights. This is useful for understanding the weight programming capabilities (and limitations) of PCM cells for weight transfer during training. It is also useful for determining the scale factor for best mapping software-trained floating-point weights into a range of hardware attainable weights (in units of microSiemens) for inference-only applications. For instance, weights trained in software may be rescaled to an optimal hardware range using a constant scale factor. A constant scale factor was chosen to preserve the entirety of the weight distribution and to avoid clipping. Intentional clipping of target values, or strategies for training in order to similarly limit weight extents, may be beneficial and impose reasonable trade-offs in some cases, but are beyond the scope of this work. The same scale factor can be applied to rescale hardware-programmed weights to their software equivalents in order to simulate the effects of weight nonidealities on neural network accuracy.

There are two free variables in determining optimal hardware conditions for weight programming: the MSP gain factor F and W_{range} , which describes the span of allowable weights on an interval symmetric about zero. For instance, $W_{\text{range}} = 100 \mu\text{S}$ represents a distribution of weights spanning from -50 to $50 \mu\text{S}$. Fine tuning these parameters can help minimize one of the critical performance metrics of interest: P_{fail} , the probability that an individual weight fails to converge. P_{fail} is defined for an individual weight, so as to report a metric that is independent of architecture (i.e., the number of weights that may be programmed in parallel). In addition, because P_{fail} is independent and identically distributed for each weight, more complex and architecture-dependent failure metrics such as the probability of having a programming fail within a row or even within

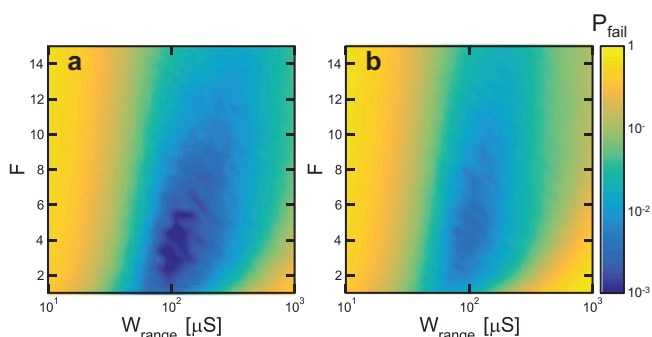


Figure 5. Probability of programming failure P_{fail} for any given weight as a function of F and W_{range} for a) normally distributed weights and b) uniformly distributed weights. When W_{range} is very low, the conductance versus pulse response of PCM is too coarse for convergence. When W_{range} is very high, P_{fail} increases due to the limited conductance range of PCM. Values of F at the high end represent extreme imbalances between MSP and LSP, hampering weight convergence. Values of F toward the low end of the spectrum (e.g., $F < 1$) effectively reverse the roles of MSP and LSP, also resulting in higher P_{fail} . Sample size is $N = 10000$ at each point in the heatmap and $E_T = 1.25\%$ of the W_{range} .

an entire array may be readily constructed. The gain factor F directly influences the ability of weights to converge on their target values by impacting the coarseness of MSP tuning relative to the LSP. Similarly, given the limited capabilities of the NVM in achieving certain conductances (Figure 3), weight distributions spanning different hardware ranges W_{range} will impact how well the programming algorithm implements a given distribution of target weights. Due to the statistical nature of the conductance versus pulse model combined with the weight programming algorithm, it is difficult to solve for an optimal F and W_{range} analytically.

The impact of each parameter on P_{fail} is simulated and depicted in **Figure 5** for both normal and uniform weight distributions. These plots show that the optimal combination of F and W_{range} is largely independent of the particular target weight distribution. In the first case (Figure 5a), normally distributed weights are used to mimic the frequently encountered case where DNN weight distributions have the highest density near zero and very few weights need to be programmed to a target value that is large in magnitude. Weights are generated using a standard normal distribution, bounded to three standard deviations to eliminate the possibility of extreme outliers. In the second case (Figure 5b), weights are generated using a uniform distribution to mimic neural network layers where the weights may be more uniformly dispersed. Optimizing P_{fail} for uniformly distributed weights has the added benefit of providing a set of parameters F and W_{range} that converge on all weights within the interval W_{range} equally well. It also provides an unbiased estimate of error density as a function of the target weight. The P_{fail} values shown in Figure 5 are calculated using an error tolerance E_T of 1.25%, which is defined as a percentage of the overall weight range. As such, the absolute magnitude of E_T in units of microSiemens is changing as W_{range} changes along the horizontal dimension of these plots. This relative error tolerance value was found to be the maximum error that can be tolerated without negative effects on DNN performance (Section 6).

Figure 5 depicts a clear central minimum or “sweet spot” in P_{fail} . If either F or W_{range} are too low or too high, weight programming becomes less effective in reaching target values. Although there exists some interplay between F and W_{range} , each parameter can be examined separately to some extent, to provide an understanding of the impact on P_{fail} . For instance, values of W_{range} at the low end of the spectrum correspond to weight distributions that are so narrow (in terms of microSiemens) that the conductance versus pulse response of PCM is too coarse to effectively converge on the target weight distribution. On the other hand, W_{range} values toward the high end of the spectrum, when combined with a low F , represent a distribution of weights that is too broad based on the limited capabilities of PCM. In this case, even the maximum conductances achievable by the PCM devices cannot reach the higher weight values. For instance, when $W_{\text{range}} = 1000 \mu\text{S}$ and $F = 1$, the majority of PCM conductances are simply unable to reach weights anywhere near $\pm 500 \mu\text{S}$. This is evidenced by the elevated P_{fail} and also becomes apparent from examining the statistical conductance data in Figure 3—very few PCMs are capable of conductances in excess of $100 \mu\text{S}$. Large values of W_{range} cannot be effectively compensated by simply increasing F , because the relative strength between MSP and LSP eventually becomes unreasonable. In this case, a single pulse on the MSP might produce a step larger than the entire range of the corresponding LSP. This renders the LSP ineffective at compensating errors introduced during MSP programming.

Values at the extreme low end of the spectrum, where $F < 1$, approach a point in which the roles of the MSP and LSP are effectively reversed. This results in an algorithm that is effectively programming the LSP before the MSP. This negates the benefit of having an LSP and results in higher P_{fail} . Figure 5 also shows some interdependence between F and W_{range} . As W_{range} increases, so does the optimal value of F for that value of W_{range} . This stems from the fact that the conductance versus pulse behavior of PCM is fixed, so that the bulk of conductances are easy to tune only to a specific range within some moderate number of pulses. As the target weight range W_{range} increases, the gain factor F should increase as well to enable the PCM to more effectively reach the now broader distribution of weights.

The optimal combination of F and W_{range} (i.e., location of the P_{fail} “sweet spot”) is largely independent of the target weight distribution. This allows the same F and W_{range} parameters to be used regardless of the weight distribution. If this were not the case, a different set of F and P_{fail} would have to be optimized for each subset of weights in a network exhibiting different characteristics (e.g., each layer). However, while the optimal combination of F and W_{range} is largely independent of the target weight distribution, the actual values of P_{fail} are not. This becomes apparent when the target weight distribution is switched from a normal distribution to a uniform distribution. The higher density of weights at the ends of the uniform distribution is less forgiving on the P_{fail} metric than normally distributed weights where the weight density quickly falls off at higher values. Because P_{fail} is more tightly constrained for the case of uniformly distributed weights, one can select values for F and W_{range} that are optimal for uniformly distributed weights

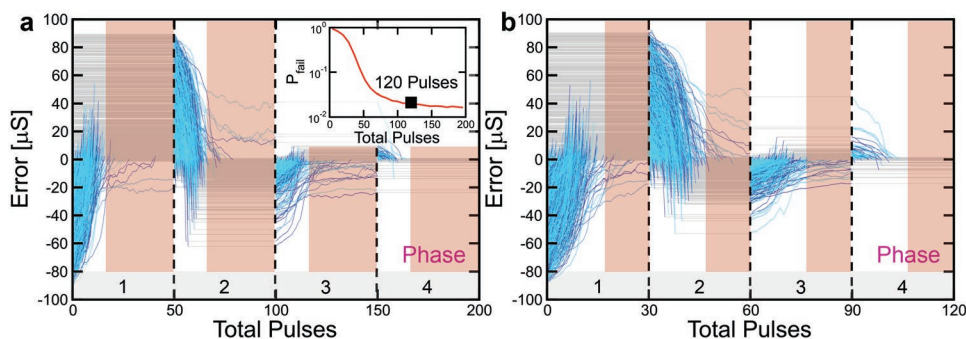


Figure 6. Error convergence trajectories during weight programming using a) 200 total pulses and b) 120 total pulses. Highlighted regions show time periods for which the majority of weights are no longer participating in programming. Allotting more than ≈ 30 pulses per phase produces diminished returns in terms of P_{fail} as PCM conductances are largely saturated. The trade-off between P_{fail} and programming speed is depicted in the inset. Programming conditions are $F = 5$, $W_{\text{range}} = 180 \mu\text{S}$, and $E_T = 1.25\%$ ($\pm 1.125 \mu\text{S}$).

with the knowledge that they will also work well for normally distributed weights.

5. Simulated Weight Programming

With a framework established for determining optimal weight programming conditions in hardware as well as software-to-hardware weight rescaling, attention can be turned toward simulating the complete weight programming process and assessing the overall efficacy of programming weights in NVM. Optimal parameters $F = 5$ and $W_{\text{range}} = 180 \mu\text{S}$ were selected based on a more fine-grained exploration of the “sweet spot” depicted in Figure 5. The programming of one million uniformly distributed weights was then simulated, with weight error convergence results depicted in Figure 6. Unless otherwise noted, NVM parameters correspond to those shown in Figure 2, implying that each individual conductance trajectory resembles those shown in Figure 2g. Although weights evolve to a variety of different values based on the target weight distribution, the error corresponding to each weight evolves toward zero. For this reason, the time evolution of weight errors are shown as opposed to the evolution of the programmed weights. The weight error is defined as the difference between the programmed weight and target weight, such that positive and negative errors correspond to the overshooting and undershooting the target weights, respectively. A uniform distribution of target weights was selected to provide an unbiased assessment of the ability of weights to converge to any location within in the optimal hardware W_{range} . This also provides an unbiased assessment of error density as a function of target weight. For the less forgiving case of uniformly distributed weights, $\approx 97.9\%$ of weights successfully converge on their target values within an error tolerance of 1.25% of W_{range} (i.e., $\pm 1.125 \mu\text{S}$). For reference, when using a standard normal distribution bounded by three sigma and rescaled to W_{range} , 99.4% of weights successfully converge to the same $E_T = 1.25\%$ of W_{range} . The reported simulation results in Figure 6 illustrate the success of the four-phase weight programming algorithm introduced in Figure 4 in minimizing programmed weight error—despite the presence of significant yet realistic levels of device variability in NVM—thanks to

the optimization of the programming strategy (e.g., using the “sweet spot” at the center of Figure 5b).

Example weight convergence trajectories are depicted in Figure 6 for two different programming durations, to highlight the trade-off between programming speed and P_{fail} . For 200 total and 120 total pulses evenly distributed among the four phases of weight programming, the majority of weights participating in programming (blue) can be seen to overshoot their targets and stop participating in programming (gray) until the subsequent phase. After four phases, and using the appropriate MSP gain factor $F = 5$, the majority of weights have converged to within E_T of their target values. In the case of 200 total pulses, the bulk of the weights are inactive for a significant portion of time during each phase (purple highlighted bars). This suggests the total programming time could be reduced with minimal effect on P_{fail} . The inset plot in Figure 6 quantifies this trade-off between P_{fail} and programming speed, revealing two distinct regimes each of which is characterized by a different slope. When the total pulse budget is large, P_{fail} is relatively flat and the total number of pulses can be reduced with minimal impact on P_{fail} . This is akin to eliminating the largely inactive regions (purple). The second regime, however, corresponds to the case where so few pulses are used per phase that active programming for the majority of weights (blue) becomes truncated. This causes P_{fail} to increase at a much higher rate as the number of pulses is reduced.

Due to the level of variability in PCM and a preference for higher fidelity programming over speed, 120 total pulses was selected as a reasonable trade-off on the P_{fail} versus speed curve. Use of 120 pulses is also reasonable given the fact that individual conductances largely saturate after ≈ 30 pulses (Figure 3). Applying pulses beyond this point provides little benefit, as can be seen in the P_{fail} versus speed trade-off. Even when programming with 120 total pulses, however, the bulk of the weights are inactive for ≈ 15 pulses per phase. Those 15 pulses are spent reducing P_{fail} primarily by attempting to converge a handful of outlier PCMs. Alternatively, if a higher P_{fail} is tolerable, weights can still be programmed reasonably well using only 60 total pulses. This is consistent with the inset characterizing the programming speed versus P_{fail} trade-off, where 60 total pulses marks the slope transition point. Decreasing the total pulses beyond this point, however, results in an accelerated rate of increase in P_{fail} .

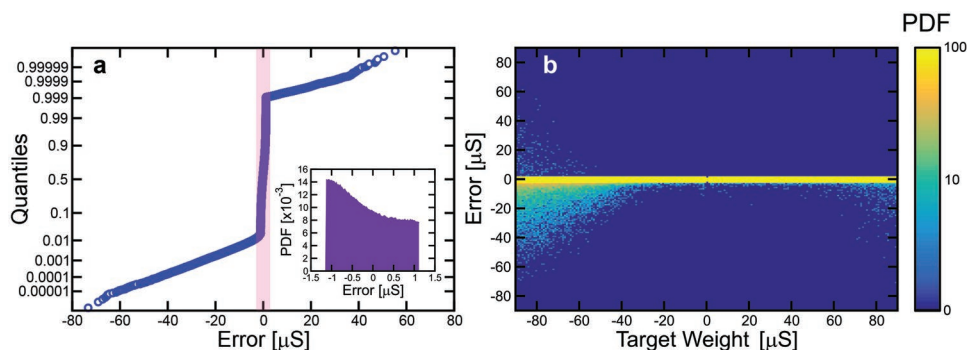


Figure 7. a) Normal quantile plot of weight errors, with the region of successfully converged weights highlighted (purple). The negative tail region of the distribution is larger than the positive tail region indicating that weight programming is more likely to undershoot than overshoot target weights. This stems from the fact that g^- is tuned in the final programming phase. The empirical probability density function of weight errors for converged weights (inset) corresponding to the purple highlighted region. b) Scatterplot heatmap showing high correlation between programmed weights and target weights (yellow) with an increasing likelihood of programming failures for higher magnitude weights (cyan). Programmed weights near the origin are zero for a region of $\pm 1.125 \mu\text{S}$, corresponding to the error tolerance E_T , since under the specified weight programming algorithm, the participation bit is immediately turned off and no pulses are ever fired to program this weight. $N = 10^6$.

Although errors for successfully programmed weights are known to be less than 1.25% of W_{range} (i.e., $\pm 1.125 \mu\text{S}$), it is interesting to examine the characteristics of the error distribution produced by the weight programming strategy. In the case of one million programmed weights, the vast majority fall within the tolerable error range as shown in **Figure 7**. There exist two tail regions, however, with roughly 2 in 100 weights undershooting and 1 in 1 000 weights overshooting their target values. This asymmetry in weight failures stems from the fact that the final phase of programming tunes g^- , which gives rise to a greater tendency to undershoot as opposed to overshoot target weights. The inset in **Figure 7** shows a relatively uniform distribution of errors for successfully converged weights. Here too, however, successfully converged weights can be found to aggregate slightly toward the lower end of the error tolerance interval, stemming again from g^- tuning in the final phase. This aggregation of weight near the lower bound is a somewhat undesirable effect as it is preferable for errors to aggregate near the center, where the absolute error is zero. The span of the weight error distribution, however, can always be reduced through manipulation of the error tolerance E_T parameter. The scatterplot heatmap in **Figure 7** shows a high correlation between programmed weights and target weights with noticeable asymmetry due to undershoot bias and an increasing likelihood of programming failures at higher magnitudes. With an optimized weight programming strategy in place and an understanding of the resulting weight error distributions, weight programming can now be simulated for exemplar neural networks to evaluate the impact of weight nonidealities on accuracy.

6. Impact on DNN Accuracy and Loss

This section examines the effects of nonidealities in programmed weights on DNN accuracy. Parameters influencing the weight nonidealities are classified into two groups: parameters governing weight programming (e.g., F , W_{range} , total pulses, E_T) and parameters describing NVM device-level vari-

ability (e.g., $\mu_{G_{\text{max}}}$, $\sigma_{G_{\text{max}}}$, μ_{S_c} , σ_{S_c}). This section reports the effects of one parameter from each group. Of the programming conditions, E_T is reported as it bears the most direction relation to weight nonidealities and is the only variable not to have been optimized previously. The second parameter μ_{S_c} , which represents the average rate at which PCM approaches its maximum conductance, is reported because it has greatest adverse effect on DNN accuracy. Impact of weight nonidealities on network performance are studied for two different types of networks. Network performance is evaluated for image classification accuracy using a three-layer MLP network trained on the MNIST dataset, and for character-based language modeling through cross-entropy loss (i.e., loss) from a two-layer LSTM network using Alice in Wonderland as the dataset.

Five sets of weights are trained for both MLP and LSTM in software using different random seeds. Floating-point software weights for each layer are then rescaled to the previously identified optimal hardware range of $\pm 90 \mu\text{S}$ for the simulation of weight programming in NVM. After simulating the weight programming process, weights are re-scaled back to their software equivalents for use in forward inference to assess the impact of weight nonidealities on network accuracy (and loss). The effects of variable E_T and slope coefficient μ_{S_c} on MLP and LSTM accuracies are shown in **Figure 8**.

Figure 8 shows that MLP accuracy and LSTM loss are both close to ideal when E_T is low and when μ_{S_c} falls within a certain range. The LSTM network is more sensitive to non-idealities in programmed weights, and the red dashed line at $E_T = 1.25\%$ in **Figure 8c** indicates where loss begins to rise. This is the origin of the E_T criteria that was adopted in the earlier sections of the paper. The variable μ_{S_c} represents the average rate at which the conductances increase in response to applied pulses during programming. A low μ_{S_c} leads to low accuracy (high loss) because only a small portion of the device conductance range can be reached with the limited number of programming pulses, and a high μ_{S_c} leads to significant overshoot of conductance values within the first steps of weight programming, to an extent that cannot be recovered in later phases.

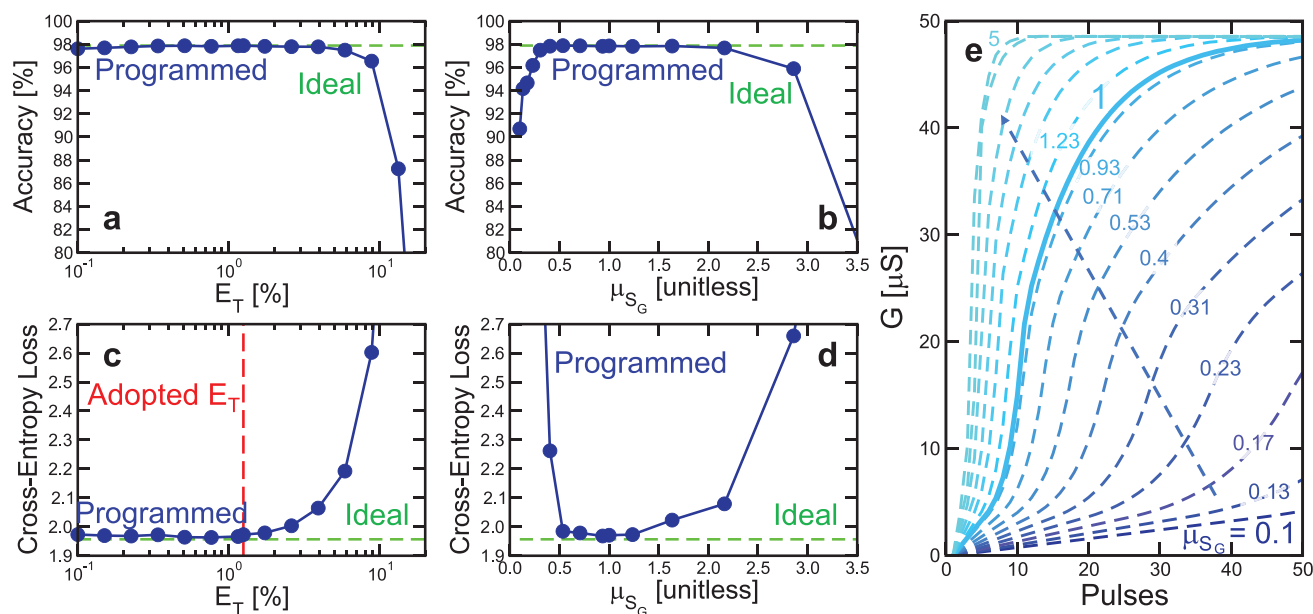


Figure 8. Accuracy as a function of a) E_T and b) μ_{S_g} for the MLP model. Cross-entropy loss of the LSTM model as a function of c) E_T and d) μ_{S_g} . The red dashed line in (c) represents $E_T = 1.25\%$. e) The impact of μ_{S_g} on the shape of the conductance versus pulse curves in the absence of cycle-to-cycle variability (removed here only for clarity of plotting).

In order to take a closer look at the programmed weights and their relation to E_T , forward inference results between the programmed weights and weights constructed by adding uniform noise bounded by E_T to the ideal target values are compared. It was previously shown that weight errors after programming are roughly uniformly distributed. **Figure 9a,b** shows that for the MLP network, simulated weight programming conforms very closely to distortion of the ideal target weights by additive uniform noise, up to E_T values of $\approx 10\%$. **Figure 9c** compares the weight errors as a function of target weight for the LSTM network with $E_T = 1.25\%$, and shows that the programmed weight distribution closely resembles the ideal target weight distribution with additive uniform noise. As a result, constructed weights with additive uniform noise may provide a reasonable estimation of programming error in network loss estimations. This in turn may be used to select an error tolerance condition, as was done previously with $E_T = 1.25\%$, for the further optimization of weight programming.

When E_T is set to a value smaller than that of the intrinsic weight programming noise, as shown in **Figure 9d**, the accuracy remains similar to the ideal value, but the distributions of errors become highly asymmetric (virtually all negative). Because most weights fail to converge within the narrow error tolerance and negative fails are more probable, most programmed weights fall below the target weights when E_T becomes very small (e.g., $E_T = 0.1\%$). When E_T is increased to a large value, all weights within the error tolerance receive no programming pulses and remain hard reset. As mentioned earlier, this has the effect of pruning all weights within the error tolerance to zero. Pruning significant portions of the weight distribution causes the accuracy or loss to degrade faster than adding uniform noise—in the case of additive uniform noise,

the constructed weights still have some positive correlation with the target weights. The regime of very high noise is not of much interest in this study, however, due to poor performance in accuracy and loss.

To reemphasize the importance of choosing the right conductance range for weight mapping, we study the effects of high weight clipping in MLP and LSTM models. Weights are artificially clipped in the models by lowering the maximum weights, setting all weights above to the maximum value, and counting the number of weights clipped. **Figure 10b** shows that the LSTM model in this study is very sensitive to weight clipping, especially the output bias weights. Therefore, optimizing weight mapping to a hardware conductance range that minimizes the probability of failure for high magnitude weights is critical for achieving low cross-entropy loss. The impact of these programming fails on network accuracy, however, is mitigated by the fact that the probability of having a large magnitude weight in combination with a weak G^+ or G^- is relatively low. On the other hand, MLP weights and non-output weights in the LSTM model can tolerate higher levels of weight clipping, but the fraction of inadvertently clipped weights should still be limited to about 1% of the total number of weights.

7. Conclusions

This work constructed a conductance versus pulse model based on experimental observations of PCM, which captures the considerable interdevice (device-to-device) and intradevice (cycle-to-cycle) variations present in maximum conductance as well as the conductance slope. A computationally inexpensive yet highly effective four-phase algorithm was then developed to

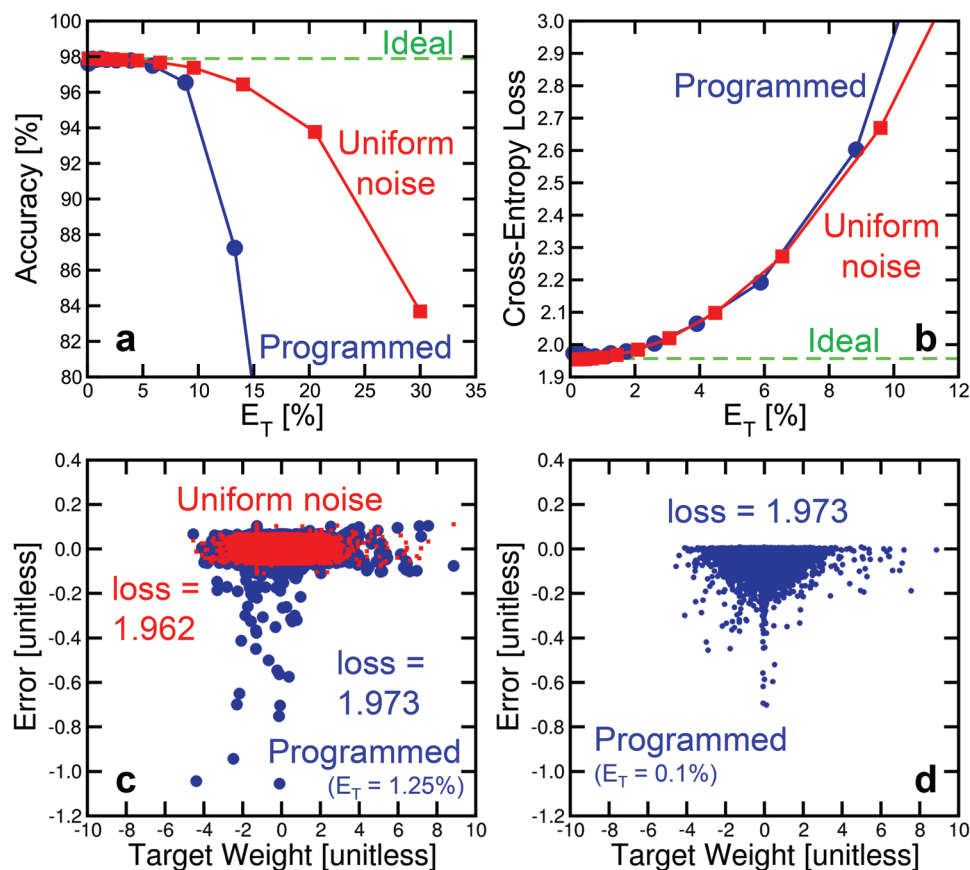


Figure 9. a) Accuracy of the MLP model and b) cross-entropy loss of the LSTM model as a function of E_T for simulated weight programming (blue) and ideal target weights with uniform additive noise (red). c) Weight error versus target weights using $E_T = 1.25\%$ for simulated programmed weights (blue circles) and for constructed weights with uniform noise (red dots). d) Weight error versus target weights for simulated weight programming when $E_T = 0.1\%$.

enable row-wise programming of weights through sequential tuning of G^+ , G^- , g^+ , and g^- . The algorithm requires only two bits per weight: one participation bit and one sign bit, which imposes minimal cost in terms of circuitry required to implement the programming algorithm.

A framework was developed for determining the optimal hardware conditions under which to program weights. This in turn was used in rescaling software-trained floating-point weights to a range optimized for NVM in units of microSiemens. This is critical for enabling software simulations to

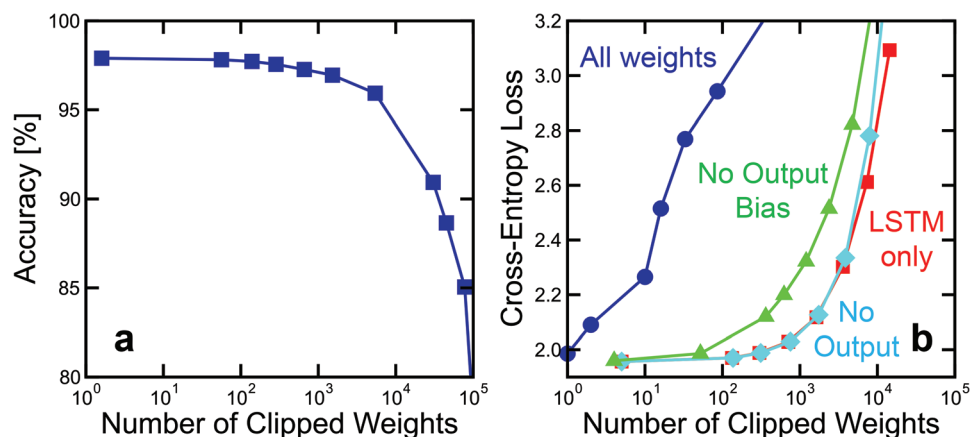


Figure 10. a) MNIST classification accuracy for MLP network and b) language modeling cross-entropy loss for LSTM model as a function of number of clipped weights. Total number of weights in the models are 228 885 for MLP and 66 256 for LSTM.

determine the impact of weight nonidealities on neural network accuracies. It is also particularly important for the use of DNN analog hardware accelerators in inference-only applications. Hardware conditions were optimized by choosing the combination of F and W_{range} which minimize the programming fail rate P_{fail} across a uniform target weight distribution. This optimization procedure finds a good optimal combination even if the actual target weight distribution is normal or otherwise nonuniform. In the case of uniformly distributed weights, $\approx 97.9\%$ of weights could be programmed to within an error tolerance of 1.25% of W_{range} , as measured with a sample size of one million weights. This increased to $\approx 99.4\%$ when weights with a truncated normal distribution are rescaled to an optimal hardware range of $\pm 90 \mu\text{S}$. Errors associated with successfully converged weights exhibited a relatively uniform distribution across the error tolerance interval, with slight aggregation toward the lower end of the error tolerance interval. For cases in which weights failed to converge on their target values (e.g., program failures), the weights were seen to exhibit a systematic undershoot bias, which stems from the fact that the algorithm tunes g^- during the final phase. Larger magnitude weights were found to possess a higher probability of program failure relative to weights with smaller magnitudes. The impact of these programming failures in actual neural networks, however, is mitigated by the fact that the density of weights tends to decline with increasing magnitude.

Implications of weight nonidealities were simulated for various neural networks including MLPs trained on MNIST as well as LSTMs trained on Alice and Wonderland. Weight clipping was found to be highly detrimental to neural network accuracy, which supports the case for software-to-hardware weight remapping using a constant scale factor. Weights programmed in NVM using the parallel row-wise weight programming strategy introduced here were found capable of achieving software-equivalent accuracies for both types of networks. This was achieved despite the considerable degree of variability present in PCM device characteristics. This suggests that the appropriate use of multiple conductances of varying significance can allow DNN analog hardware accelerators—already capable of significant improvements in throughput per area and energy efficiency over their digital counterparts—to also achieve the full generalization accuracies offered by DNN models pretrained in software. Future work will be needed to address the additional impact of resistance drift and read noise during the read-verify step of weight programming.

Supporting Information

Supporting Information is available from the Wiley Online Library or from the author.

Conflict of Interest

The authors declare no conflict of interest.

Keywords

analog hardware accelerator, crossbar array, deep learning, phase change memory

Received: January 4, 2019

Revised: February 22, 2019

Published online: April 23, 2019

- [1] T. Qureshi, P. Chingambam, S. Shafaq, Large-scale deep unsupervised learning using graphics processors, **2009**.
- [2] Y. LeCun, Y. Bengio, G. Hinton, *Nature* **2015**, 521, 436.
- [3] K. He, X. Zhang, S. Ren, J. Sun, *Proc. of the IEEE Int. Conf. on Computer Vision*, IEEE, USA **2015**, pp. 1026–1034.
- [4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis, *Nature* **2016**, 529, 484.
- [5] G. W. Burr, R. M. Shelby, S. Sidler, C. Di Nolfo, J. Jang, I. Boybat, R. S. Shenoy, P. Narayanan, K. Virwani, E. U Giacometti, B. N. Kurdi, H. Hwang, *IEEE Trans. Electron Devices* **2015**, 62, 3498.
- [6] G. W. Burr, R. M. Shelby, A. Sebastian, S. Kim, S. Kim, S. Sidler, K. Virwani, M. Ishii, P. Narayanan, A. Fumarola, L. L. Sanches, I. Boybat, M. Le Gallo, K. Moon, J. Woo, H. Hwang, Y. Leblebici, *Adv. Phys.: X* **2017**, 2, 89.
- [7] I. Boybat, M. Le Gallo, S. R. Nandakumar, T. Moraitis, T. Parnell, T. Tuma, B. Rajendran, Y. Leblebici, A. Sebastian, E. Eleftheriou, *Nat. Commun.* **2018**, 9, 1.
- [8] S. Ambrogio, P. Narayanan, H. Tsai, R. M. Shelby, I. Boybat, C. di Nolfo, S. Sidler, M. Giordano, M. Bodini, N. C. P. Farinha, B. Killeen, C. Cheng, Y. Jaoudi, G. W. Burr, *Nature* **2018**, 558, 60.
- [9] Y. Cui, C. Ingaz, T. Gao, A. Heydari, in *Intersociety Conf. on Thermal and Thermomechanical Phenomena in Electronic Systems*, IEEE, USA **2017**, pp. 936–942.
- [10] M. Wiboonrat, in *Int. Conf. on Ecological Vehicles and Renewable Energies*, IEEE, USA **2014**, pp. 1–6.
- [11] B. Grot, D. Hardy, P. Lotfi-Kamran, B. Falsafi, C. Nicopoulos, Y. Sazeides, *IEEE Micro* **2012**, 32, 52.
- [12] N. Ahuja, C. W. Rego, S. Ahuja, S. Zhou, S. Shrivastava, in *29th IEEE Semiconductor Thermal Measurement and Management Symp.*, IEEE, USA **2013**, pp. 1–5.
- [13] J.-W. Jang, S. Park, G. W. Burr, H. Hwang, Y.-H. Jeong, *IEEE Electron Device Lett.* **2015**, 36, 457.
- [14] J.-W. Jang, S. Park, Y.-H. Jeong, H. Hwang, in *IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, USA **2014**, pp. 1054–1057.
- [15] S. Lim, M. Kwak, H. Hwang, *IEEE Trans. Electron Devices* **2018**, 65, 3976.
- [16] M. Tadayoni, S. Hariharan, S. Lemke, T. Pate-Cazal, B. Bertello, V. Tiwari, N. Do, in *IEEE International Conference on Microelectronic Test Structures (ICMTS)*, IEEE, USA **2018**, pp. 27–30.
- [17] X. Guo, F. Merrikh-Bayat, M. Bavandpour, M. Klachko, M. R. Mahmoodi, M. Prezioso, K. K. Likharev, D. B. Strukov, in *IEEE International Electron Devices Meeting (IEDM)*, IEEE, USA **2017**, pp. 151–154.
- [18] F. Merrikh-bayat, X. Guo, M. Klachko, M. Prezioso, K. K. Likharev, D. B. Strukov, *IEEE Trans. Neural Networks Learn. Syst.*, **2018**, 29, 4782.
- [19] F. Merrikh-Bayat, X. Guo, M. Klachko, M. Prezioso, K. K. Likharev, D. B. Strukov, *IEEE Trans. Neural Networks Learn. Syst.* **2016**, 29, 4782.
- [20] L. Fick, D. Blaauw, D. Sylvester, S. Skrzyniarz, M. Parikh, D. Fick, in *IEEE Custom Integrated Circuits Conference (CICC)*, IEEE, USA **2017**, pp. 1–4.

- [21] N. Papandreou, H. Pozidis, A. Pantazi, A. Sebastian, M. Breitsch, C. Lam, E. Eleftheriou, in *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*, IEEE, USA **2011**, pp. 329–332.
- [22] R. M. Shelby, G. W. Burr, I. Boybat, C. Di Nolfo, in *IEEE Int. Reliability Physics Symp. Proc.*, IEEE, USA **2015**, pp. 1–6.
- [23] M. J. Marinella, S. Agarwal, A. Hsia, I. Richter, R. Jacobs-Gedrim, J. Niroula, S. J. Plimpton, E. Ipek, C. D. James, *IEEE J. Emerging Sel. Top. Circuits Syst.* **2018**, 8, 86.
- [24] M. V. Nair, P. Dudek, in *Proc. of the Int. Joint Conf. on Neural Networks*, IEEE, USA **2015**, pp. 1–7.
- [25] J. Alspector, R. Meir, B. Yuhas, A. Jayakumar, D. Lippe, in *Advances in Neural Information Processing Systems*, NIPS, USA **1993**, pp. 836–844.
- [26] S. Agarwal, R. B. Jacobs-Gedrim, A. H. Hsia, D. R. Hughart, E. J. Fuller, A. A. Talin, C. D. James, S. J. Plimpton, M. J. Marinella, in *2017 Symp. on VLSI Technology*, JSAP, USA **2017**, pp. 174–175.
- [27] H. Wu, P. Yao, B. Gao, W. Wu, Q. Zhang, W. Zhang, N. Deng, D. Wu, H.-S. P. Wong, S. Yu, H. Qian, in *International Electron Devices Meeting (IEDM)*, IEEE, USA **2017**, pp. 274–277.
- [28] G. Cristiano, M. Giordano, S. Ambrogio, L. P. Romero, C. Cheng, P. Narayanan, H. Tsai, R. M. Shelby, G. W. Burr, *J. Appl. Phys.* **2018**, 124, 151901.
- [29] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *Proc. IEEE* **1998**, 86, 2278.
- [30] G. W. Burr, R. M. Shelby, C. di Nolfo, J. Jang, R. S. Shenoy, P. Narayanan, K. Virwani, E. U. Giacometti, B. N. Kurdi, H. Hwang, in *International Electron Devices Meeting (IEDM)*, IEEE, USA **2014**.
- [31] A. Pirovano, A. L. Lacaita, F. Pellizzer, S. A. Kostylev, A. Benvenuti, R. Bez, *IEEE Trans. Electron Devices* **2004**, 51, 714.