Precision of bit slicing with in-memory computing based on analog phase-change memory crossbars

Manuel Le Gallo1,* , S R Nandakumar1, Lazar Ciric1, Irem Boybat1, Riduan Khaddam-Aljameh1, Charles Mackin2 and Abu Sebastian1,*

1 IBM Research Europe, Säumerstrasse 4, 8803 Rüschlikon, Switzerland
2 IBM Research Almaden, San Jose, CA 95120, United States of America
* Authors to whom any correspondence should be addressed.
E-mail: anu@zurich.ibm.com and ase@zurich.ibm.com

Keywords: in-memory computing, phase-change memory, bit slicing

Abstract

In-memory computing is a promising non-von Neumann approach to perform certain computational tasks efficiently within memory devices by exploiting their physical attributes. However, the computational accuracy achieved with this approach has been rather low, owing to significant inter-device variability and inhomogeneity across an array as well as intra-device variability and randomness from the analog memory devices. Bit slicing, a technique for constructing a high precision processor from several modules of lower precision, is a promising approach for overcoming this accuracy limitation. However, a systematic study to assess the precision ultimately achieved by bit slicing with analog in-memory computing has so far been lacking. In this work, we assess the computational error from bit slicing when performing in-memory matrix-vector multiplications. Using accurate models of phase-change memory crossbar arrays, we demonstrate that unlike in digital processors where bit slicing is used to extend the dynamic range of the number representation, bit slicing with in-memory computing should aim at minimizing the error from the analog matrix representation through averaging within a given dynamic range. The results are validated using a prototype phase-change memory chip and the impact on the neural network inference accuracy on CIFAR-10 and ImageNet benchmarks is evaluated.

1. Introduction

In-memory computing cores typically consist of a memory array, data converters and a digital control unit [1, 2]. The memory array is organized in a crossbar configuration where memory devices are placed at the intersection of horizontal wordlines and vertical bitlines. The array performs computations across multiple rows in an analog manner, without deciphering the memory content on a per-row basis. The core interfaces with the rest of the computer system through digital signals; digital-to-analog converters (DACs) are responsible for the input while analog-to-digital converters (ADCs) digitize the output of the array. When performing a matrix-vector multiplication (MVM) on such in-memory computing cores in an analog fashion, the elements of the matrix are represented by the conductance values of memory devices and the elements of the vector are represented by the amplitude or duration of the input voltage. The accumulated current (in the case of pulse amplitude-based encoding) or the accumulated charge (in the case of pulse-width encoding) across each bitline is indicative of the dot product between the input vector and one column of the matrix. Owing to the reduction in matrix data transfers, parallelism and analog mode of operation, in-memory computing allows to perform MVMs at high energy efficiency (>10 TOPS/W) [3, 4]. Thus, in-memory computing can be a viable alternative to conventional GPU-based and other digital solutions for MVM acceleration, which is critical for many applications including deep learning.
Encoding each matrix element with a single device is shown to be effective for many applications with relaxed precision requirements [5, 6]. Such an implementation can also provide high area efficiency. The devices can provide either a binary precision with just two levels or multi-level storage with a typical equivalent precision of 4–6 bits [7]. Yet, the precision available at the array level may not be sufficient for a broader range of applications [8, 9]. Furthermore, for applications requiring high precision, the DAC resolution requirements may be prohibitive for allowing a compact and energy efficient design. High resolution for the ADC may be required as well to precisely convert MVM outputs when high input resolution, high precision of the memory devices, and/or large crossbar array sizes are used [10], thereby leading to high energy consumption and area.

The above limitations could, to a certain extent, be remedied by employing a technique called 'bit slicing'. Bit slicing is a rather old approach in computer architecture popular in 1970s–80s for constructing a digital processor from modules of smaller bit width [11]. Each of the modules processes one 'slice' of an operand of larger bit width. The grouped processing components will then have the capability to process, in parallel, an arbitrarily chosen full word-length of a particular task. For instance, two 4 bit slice processors can be combined to process 8 bit words. However, technological advances made it possible to integrate large numbers of transistors on a single chip and develop processors with large word-lengths (such as 32 bit and 64 bit), leading to the decline of bit-slice processors in modern computing systems.

Yet, this concept has been increasingly adopted for increasing the accuracy of the in-place MVM based on in-memory computing [10, 12–21]. According to this technique, an element of the matrix is mapped onto \( n_w \) device conductance values placed on \( n_v \) columns of a crossbar array. Similarly, bit slicing can also be applied to the input vector elements, which are mapped onto \( n_i \) voltage vectors input to the crossbar array one at a time. To perform an in-place MVM, a vector slice is multiplied with a matrix slice, with \( O(1) \) time complexity, and the partial products of these operations are combined outside of the crossbar arrays through a shift-and-add reduction network. This technique has been successfully implemented in some of the recent in-memory computing chips based on binary RRAM and SRAM devices [4, 22, 23].

However, a systematic study to assess the precision ultimately achieved with bit slicing when implemented on analog memory crossbar arrays has so far been lacking. Unlike in digital processors, the errors from the in-place MVM performed with analog in-memory computing are mostly of random nature. Possible factors leading to such errors include 1/f noise from analog memory devices and circuits, temporal conductance drift, device-to-device variations, defective (stuck) devices, and circuit non-idealities (e.g., IR drop). Hence, since random errors perturb the partial results obtained from all the different slices, it is not clear what is the most accurate way to perform bit slicing with in-memory computing.

Previous works devised various strategies to program a matrix element to multiple device conductance values and applied them to inference [20] and training [18, 24] of deep neural networks (DNNs). However, the study done in [20] was restricted to only 2 matrix slices and a single programming algorithm, and no experimental validation of the approach was presented. Experimental results were presented for training DNNs using multiple devices per synaptic weight [18, 24], however the proposed programming schemes cannot be applied as-is for programming pre-trained weights to perform inference only, and the optimal mapping of weight to conductance values for training or inference applications may differ. Other works explored programming strategies to correct the errors from bit slices of higher significance [25], as well as the effect of conductance averaging [26] and various conductance mapping techniques [27] on the MVM error. However, those works focus each on a single aspect of the problem (e.g. only on weight programming algorithms or only on weight slicing techniques), the device models used are simplified and not calibrated to match the behavior of large arrays of devices, and no experimental results are presented to validate the simulations.

In this work, we perform a precise assessment of the MVM error using bit slicing with in-memory computing. We extend the original digital bit slicing approach proposed in references [13, 14, 21] to support analog weight slices and propose several weight programming algorithms for them. We study the combined influence of the number of slices, base, and weight programming algorithms on the resulting MVM error using accurate statistical models of phase-change memory (PCM) arrays. This allows us to derive insights into what are the most optimal ways to encode the slices in a nonideal analog in-memory computing architecture. The proposed weight slicing approach is implemented experimentally on a prototype multi-level PCM chip to validate the simulation results. Finally, we study the impact of bit slicing with in-memory computing on the DNN inference accuracy on CIFAR-10 and ImageNet benchmarks.

2. Materials and methods

2.1. Bit slicing models

Figure 1 shows an implementation of bit slicing of inputs and weights using an in-memory computing architecture. An input vector \( I \) of size \((1, N)\) and with a value range \( r_I \) is decomposed into \( n_I \) slices. Each slice \( i \) has...
in case a digital input is applied. It is also suitable for spike-based input encoding [28, 29], where an input spike has been used with binary charge-based (e.g., SRAM, Flash) and binary resistive (e.g., MRAM) technologies. However, for weight slicing, it is not clear whether this type of slicing is optimal when weights have a significance of $b_{j}^{r_{i}}$, where $i \in \{1, 2, \ldots n_{I}\}$. Similarly, a weight matrix $W$ of size $(N, N)$ and with value range $r_{W}$ is mapped to $n_{W}$ slices. Each slice $j$ has a significance of $b_{W}^{r_{j}}$, where $j \in \{1, 2, \ldots n_{W}\}$. $b_{i,W}$ are referred to as the base of the input or weight slice. The function $f$ represents the ADC conversion function. The computation of the MVM output $O$ from input vector $I$ and weight matrix $W$ can be thus written as

$$I = b_{i}^{r_{I}}[I]_{1} + b_{i}^{r_{I}}[I]_{2} + \cdots + b_{i}^{r_{I}}[I]_{n_{I}} \quad (1)$$

$$W = b_{W}^{r_{W}}[W]_{1} + b_{W}^{r_{W}}[W]_{2} + \cdots + b_{W}^{r_{W}}[W]_{n_{W}} \quad (2)$$

$$O = \sum_{i=1}^{n_{I}} b_{i}^{r_{I}} \cdot \sum_{j=1}^{n_{W}} b_{j}^{r_{W}} \cdot f([I]_{i} \cdot [W]_{j}). \quad (3)$$

As shown in figure 1(a), each input slice can be applied as a voltage vector input to the crossbar array one at a time. Each weight slice can be programmed on a different column of the array. Upon application of an input, the results from each weight slice are digitized by the ADC, scaled by their respective base $b_{i,W}$, and finally summed up to produce the output $O_{l}$ of a given weight matrix column $l$.

![Figure 1.](image)

**2.1.1. Digital (positional) slicing**

The standard way to perform bit slicing in a digital processor is to construct the slices based on the positional notation used to represent data. For example, in a binary numeral system, an 8 bit number could be decomposed into two 4 bit slices, or four 2 bit slices. The slices are constructed from the digits of the original 8 bit binary number.

When performing input or weight slicing in this mode, the base would be given by

$$b_{i,W} = 2^{[(p_{i,W} - 1)/n_{i,W}]} \quad (4)$$

where $p_{i,W}$ is the full signed bit length used to represent input or weight. The resulting input $[I]_{i}$ or weight $[W]_{j}$ slice values would be elements of $\{0, 1, \ldots, b_{i,W} - 1\}$. Therefore, the range of the input or weight slices $r_{i,l_{i},W}$ is

$$r_{i,l_{i},W} = b_{i,W} - 1. \quad (5)$$

Such digital bit slicing can be used when performing slicing of the input vector applied to the crossbar array, in case a digital input is applied. It is also suitable for spike-based input encoding [28, 29], where an input spike vector at any time point can be viewed as a binary input slice. For weight slicing, this type of representation has been used with binary charge-based (e.g., SRAM, Flash) and binary resistive (e.g., MRAM) technologies [12, 15, 19]. However, for weight slicing, it is not clear whether this type of slicing is optimal when weights...
are programmed as analog conductance values. The noise and variability from the memristive devices will create random errors when programming and reading the conductance values, and these errors will reduce the achievable precision of the bit slicing. In particular, the errors from the slices encoding the most significant bits may dominate over the outputs of the least significant bit slices. Next, we discuss alternative techniques that can be used to perform slicing of analog weights.

2.1.2. Analog weight slicing

In analog resistive memory devices such as PCM and resistive random access memory (RRAM), the weights can be programmed within a continuum of conductance values, as opposed to discrete digital values like in the case described above [6, 30, 31]. For a given base \( b_{W} \), we can thus write \([W] \in [0, r_{W} b_{W}]\), where

\[
\eta = \left\{ \begin{array}{ll}
\eta_{I} \left(1 - b_{W}\right) \left(1 + b_{W}^{n_{W}}\right) / \left(1 + b_{W}\right)
& \text{if } b_{W} > 1 \\
\eta_{I} / \sqrt{n_{W}}
& \text{if } b_{W} = 1.
\end{array} \right.
\]

Thus, as opposed to digital slicing, here it is possible to choose the base \( b_{W} \) independent of the number of slices \( n_{W} \), and find a set of weight slices that will appropriately represent \( W \). The task is then to devise programming algorithms to appropriately derive the set of analog weight slices \([W]\).

2.2. Programming algorithms for analog weight slicing

2.2.1. Equal-fill

The most naive way to program the analog weight slices is simply to set all of them to the same value (equal-fill algorithm). That is,

\[
[W]_{i} = W \frac{r_{i,W}}{r_{W}},
\]

where \( r_{i,W} \) is given by equation (6).

When we follow the equal-fill programming, the MVM error distribution due to weight noise is identical on each slice. The total MVM error resulting from all slices summed with base \( b_{W} \) can be determined analytically. If \( \eta_{I} \) is the MVM error from a single slice, the effective MVM error from \( n_{W} \) slices, \( \eta_{I} \) is given by (see supplementary note I (https://stacks.iop.org/NCE/2/014009/mmedia) for derivation)

\[
\eta = \left\{ \begin{array}{ll}
\eta_{I} \left(1 - b_{W}\right) \left(1 + b_{W}^{n_{W}}\right) / \left(1 + b_{W}\right)
& \text{if } b_{W} > 1 \\
\eta_{I} / \sqrt{n_{W}}
& \text{if } b_{W} = 1.
\end{array} \right.
\]

From equation (8), it is easy to show that the error \( \eta \) is the lowest for \( b_{W} = 1 \), irrespective of the number of weight slices \( n_{W} \) (see supplementary note I for derivation).

It is also possible to show that in the specific case where the MVM slice error \( \eta_{I} \) results from pure multiplicative noise on the weights, programming weight slices with equal-fill and \( b_{W} = 1 \) minimizes the MVM error \( \eta \) over all possible programming algorithms (see supplementary note II). However, the conductance noise in actual resistive memory devices such as PCM has far more complex dependencies on the actual conductance value, the programmed state, and time. So, it is not immediately obvious that the equal-fill algorithm with \( b_{W} = 1 \) would perform optimally in hardware.

2.2.2. Max-fill

An alternative to equal-fill is to attempt to fill as many slices as needed towards their maximum possible value \( r_{i,W} \), and set the others to 0 (max-fill algorithm). In other words, this algorithm tries to always use the minimum number of nonzero slices to represent a given weight. This is motivated by the fact that in most resistive memory devices, the high conductance states can generally be programmed with lower (relative) error [5]. Moreover, when the ratio between the highest and lowest conductance state is large \((\gtrsim 10^{3})\), the absolute conductance noise from the lowest conductance state is negligible compared to that of the high conductance states [32]. So, having most slices programmed to 0 and only a minimum number of them programmed to high values might be beneficial to reduce the overall error. The proposed approach is formally described in algorithm 1.

For each slice \( i \), we first check the maximum number that can be encoded in the \( i - 1 \) previous slices (line 3 of algorithm 1). If the absolute weight value to be encoded is smaller than this number, slice \( i \) is programmed to 0 and we move to the next slice of lower significance. Otherwise, slice \( i \) is programmed up to its maximum value, \( r_{i,W} \), and the rest of the weight is carried out to the other slices. This procedure allows to maximize the number of slices programmed to 0, but at the same time prevents encoding a small weight on the slice with the highest significance, which would unnecessarily amplify the noise from the analog weights.
2.2.3. Max-fill with error correction

The max-fill algorithm can be extended to further reduce the overall error arising from weight programming inaccuracies. This extension is referred to as max-fill with error correction (max-fill with EC). When programming a slice according to algorithm 1, the error measured between the desired and programmed slice value is added to the remaining weight magnitude \( W^r \) and is used for the further slicing. Hence, the idea is to compensate the error from programming one slice using subsequent slices. The proposed approach is formally described in algorithm 2.

However, the programming of subsequent slices will also encounter similar programming inaccuracies. Hence, it is more beneficial if the error from one slice is compensated by slices of lower significance. Indeed, if the programming errors from all the slices can be fully compensated by this method, the remaining error on the total weight will be that of the least significant slice, \([\varepsilon]_1\). It is clear that the higher \( b_W \) is, the lower will be \([\varepsilon]_1\) compared with the total weight dynamic range. But then, using a high \( b_W \) could also prevent errors from the high significant slices to be fully corrected by slices of lower significance if they are too large. These different tradeoffs will be analyzed in more detail in section 3.

2.3. Simulation model

2.3.1. Setup

In order to evaluate the accuracy of bit slicing on PCM crossbars, the simulation setup shown in figure 2 is used. The bit slicing schemes are first evaluated using random input vectors and weight matrices with normal distribution, \( \mathcal{N}(0, 1) \). Input vectors \( I \) of size \((1, N)\) and weight matrices \( W \) of size \((N, N)\), with \( N = 128 \), are used. The bit resolution of \( I \) is INT-\( p_I \), \( W \) is INT-\( p_W \) and that of quantized results from ADCs is INT-\( p_O \), where \( p_I = p_W = 9 \) and \( p_O = 8 \) are assumed by default. The input is sliced into \( n_I\) slices, and the weights into \( n_W\) slices. The MVM results from each input/weight slice combination are individually digitized by the ADC, then scaled and added using digital arithmetic. Weight and input slices are always signed in the simulations. The accuracy of the MVM results are characterized using a relative error computed as: \( ||O - I_{FP} \cdot W_{FP}||_2 / ||I_{FP} \cdot W_{FP}||_2 \), where \( O \) is the computed MVM result from the model, and \( I_{FP} \cdot W_{FP} \) is the exact result computed in 64 bit floating-point.

2.3.2. ADC model

It is important to determine the expected range of analog MVM results from the individual slices to set the ADC input range properly to minimize ADC quantization loss and clipping. We expect to cover \( m_I \) and \( m_W \), number of standard deviations of original input and weight distributions and \( m_O \) number of standard deviations of
Figure 2. Simulation setup for computing MVM accuracy on PCM crossbars with bit slicing. Floating point Gaussian distributed inputs and weights are first converted to (by default) INT-9 format. Resulting weights $W$ are then sliced according to a given algorithm, and then the weight slices are linearly mapped to differential conductance values $G^+, G^-$. The computed MVM result on inputs slices times weight slices are scaled by $s_O$, which represents the hardware ADC gain, before being digitized to (by default) INT-8. The partial results are then shift-and-added, multiplied by the drift compensation factor $\alpha(t)$, and finally rescaled to obtain the final output $O$. The drift compensation factor $\alpha(t)$ is computed using the same simulation setup by applying an all-constant input $I_{cal}$ to obtain the output $O_{cal}(t)$.

ADC input. Therefore, we determined an output scaling factor, $s_O$, to map the ADC input to output as a function of input/weight ranges, crossbar size $N$, and number of input/weight slices through the following empirical formula:

$$s_O = r_O \times \frac{1}{m_O} \times \frac{(m_l - 1.5)/n_l + 1.5}{r_{l,j}} \times \frac{(m_W - 1.5)/n_W + 1.5}{r_{W,l}}$$

(9)

where $r_O = 2^{p_O-1} - 1$ corresponds to the output range of a $p_O$-bit ADC. Due to slicing, the distribution of the weight and input slices changes from Gaussian to a categorical distribution with sample space of $\{-1, 0, 1\}$ when increasing the number of slices. The empirical factor (the two rightmost terms) adapts the output scale to take into account the changes in the ADC input distributions due to this (see supplementary note III). Each partial dot product resulting from an input/weight slice combination is scaled by $s_O$, quantized, and clipped at $r_O$ to determine the final ADC output.

Note, however, that the mapping between ADC input and output according to equation (9) may not be the most optimal when using slices with low bit-width and small crossbar sizes. In this particular case, it may be possible to set the ADC resolution to $p_O = \left\lceil \frac{(p_l - 1)/n_l + (p_W - 1)/n_W + \log_2(N)}{1 + \left(\frac{p_l - 1}{n_l} + \frac{p_W - 1}{n_W} + \log_2(N)\right)} \right\rceil$ if $(p_l - 1)/n_l > 1$ and $(p_W - 1)/n_W > 1$, or $p_O = \left\lfloor \frac{(p_l - 1)/n_l + (p_W - 1)/n_W + \log_2(N)}{1 + \left(\frac{p_l - 1}{n_l} + \frac{p_W - 1}{n_W} + \log_2(N)\right)} \right\rfloor$ otherwise, and obtain lossless digitization of the sliced MVM outputs [13]. To achieve this, one should use $s_O = 1$. However, with this approach, the required number of bits for the ADC becomes impractically large when dealing with large crossbar sizes and multi-bit input and weight slices. Since such large crossbars and multi-bit implementations can be advantageous with respect to throughput and energy efficiency [3], we refrained from using the latter approach and always used equation (9) to map the ADC input to its output, to obtain consistent results between the different slicing modes.

2.3.3. PCM model

For the evaluations, we use a statistical model of PCM developed based on the characterization of doped-Ge$_2$Sb$_2$Te$_5$ (d-GST) mushroom-type PCM from a million device array integrated in 90 nm CMOS technology [33]. A mushroom-type PCM device consists of a volume of phase-change material sandwiched between two metal electrodes. The phase-change material can be found either in the orderly high-conductance crystalline phase, or in the disordered low-conductance amorphous phase. A continuum of conductance values can be achieved in a PCM device by changing the ratio of amorphous and crystalline material through the application of suitable electrical pulses [5]. Amorphizing the material is referred to as a ‘RESET’ operation while crystallization is described as a ‘SET’ operation.

For mapping the sliced weight matrix $W$ to the PCM crossbar array, each weight slice is mapped to a differential PCM pair [34]. We assume a closed loop iterative programming with a read delay $b_t$ of 25 s and maximum...
conductance range $G_{\text{max}}$ of 25 $\mu$S [35]. Hence, each weight slice range $r_{c,W}$ is linearly mapped to the conductance range to obtain target conductance values $G_T$. Depending on the sign of the weight, either the positive or negative PCM device is programmed to $G_T$, and the other is RESET to 0 $\mu$S. The actual conductance values $G(t)$ at time $t$ used for the MVMs are obtained from $G_T$ by incorporating programming noise, conductance drift, and read noise using the following equations

$$G_{\text{prog}} = G_T + \mathcal{N}(0, \sigma^2_{\text{prog}})$$

$$G_{\text{drift}}(t) = G_{\text{prog}}(t_0)(t/t_0)^{-\nu}$$

$$G(t) = G_{\text{drift}}(t) + \mathcal{N}(0, \sigma^2_{\text{nc}}(t))$$

where $\sigma_{\text{prog}}(G_T)$ is the standard deviation of the non-linear state-dependent programming noise, $\nu(G_T) = \mathcal{N}(\mu, G_T), \sigma^2(G_T))$ is the state-dependent and device-to-device stochastic drift exponent, and $\sigma_{\text{nc}}(G_T, t)$ is the estimated standard deviation of 1/f noise measured in the PCM devices. Additional details on the model, PCM characterization data, and a description of the experimental platform can be found in supplementary note IV.

2.3.4. Global drift compensation

The PCM conductance drift described in equation (11) has a detrimental impact on the MVM accuracy because it makes the weights programmed in PCM reduce over time. Global drift compensation (GDC) has been shown to be an effective tool to mitigate the effect of conductance drift in PCM on the MVM accuracy [5, 7, 36]. The idea of GDC is to compute a single scaling factor that can be applied to the output of the entire crossbar in order to compensate for a global conductance shift. The proposed implementation of GDC with bit slicing is shown in figure 2. Here, we compute the GDC scaling factor at time $t$, $\alpha(t)$, from the one-norm of the output $O_{\text{cal}}(t)$ obtained by applying an all-constant input $I_{\text{cal}}$. Hence,

$$\alpha(t) = \frac{\|O_{\text{cal}}(t_0)\|_1}{\|O_{\text{cal}}(t)\|_1}$$

where $O_{\text{cal}}(t_0)$ is a reference output obtained by applying $I_{\text{cal}}$ at a defined time $t_0$ after programming the weights. The MVM output at time $t$ can be then multiplied by the resulting $\alpha(t)$ scalar in order to compensate for conductance drift. Note, however, that GDC can compensate only for a global conductance shift across the array, but the variability of the drift exponent across devices will still cause errors in the MVM results over time. The impact of these errors on the MVM accuracy and how bit slicing can mitigate them will be analysed in the next section.

3. Results

3.1. Matrix-vector multiplication accuracy

First, we evaluate the impact of the ADC resolution on the MVM error when using input and weight slicing. To characterize solely the error coming from the ADC, the PCM model that simulates non-ideal analog weight slices is turned off. For simplicity, we use digital positional slicing on both inputs and weights. The results are shown in figure 3 for INT4 to INT12 ADC resolution. The error reduces by a factor of approximately 2 for every added bit of ADC resolution, as expected from a uniform quantization model. Moreover, no systematic reduction of error is observed by using bit slicing on inputs or weights. It shows that, assuming ideal input generation from the hardware peripheral circuitry, no significant MVM accuracy decrease or improvement is expected from performing bit slicing on inputs. Input bit slicing, however, could be used to improve the energy efficiency and throughput, and simplify the DAC. It also shows that performing bit slicing does not allow to use a lower ADC resolution without having an increase in MVM error, despite the lower bit resolution of weight and input slices. This is a consequence of the scaling performed in equation (9), which results in covering only $m_O$ standard deviations of ADC input within the range $r_O$, instead of its full bit-width. We also observe that the relative MVM error due to ADC is less than 2% when the resolution is 8 bit or higher. Therefore, we expect that for high resolutions, the MVM error will be dominated by errors from analog weights, which are expected to be on the order of 10% or higher for existing resistive memory technologies [5].

Next, we evaluate the impact of the different weight slicing modes on the error obtained when using the PCM model. In contrast to the digital hardware with precise bit representation, where bit slicing is used to extend the dynamic range available to the number representation, slicing the analog weights aims to minimize the error from the analog representation within a given dynamic range. In the crossbar array, each weight slice $[W]_i$ is programmed to the same conductance range with identical programming noise model. Hence,
Figure 3. Relative MVM error as a function of ADC resolution for different positional slicing configurations of inputs and weights. In this simulation the PCM model is disabled in order to characterize solely the error coming from ADC conversion. The relative MVM error here is computed as \(|O - I \cdot W|/|I \cdot W|\), where \(I\) and \(W\) are the integer inputs and weights as opposed to their floating point version (see figure 2). This is done so that only the relative MVM error coming from ADC conversion is captured in this simulation. The values of \(m_0\) (number of standard deviations of ADC input covered by ADC range) were optimized empirically for each ADC resolution \(p_0\) to minimize the MVM error. Values of \(m_0 = 2, 3, 4, 5, 5\) for \(p_0 = 4, 6, 8, 10, 12\) were used, respectively.

the errors from the individual slices get averaged based on their respective significance factor at the array periphery.

When using equal-fill programming, the error resulting from analog weight noise is predicted by equation (8). Since the total MVM error is dominated by the weight error compared to those from ADC and input slicing, the MVM error is expected to follow the same trend. Our simulation results for equal-fill with \(b_W = 1, 2\) in figure 4(a) suggests this indeed is true. When the noise on individual weight slices is similar, the best strategy is to average them with equal significance. In comparison, for the digital slicing with two slices \((n_W = 2)\), the significance ratio between the slices is 1:16. In this case, the noise is dominated by that from the most significant slice, and the effective MVM error is close to that from the no-slicing case \(n_W = 1\). On the other hand when \(n_W = 8\), digital slicing becomes equivalent to the case of \(b_W = 2\) with max-fill, at which point their significance factors and programming algorithm become identical. As discussed earlier, max-fill maximizes the number of zero slices and non-zero slices are programmed close to \(G_{\text{max}}\). The zero slices are assumed to be programmed with zero noise (fully RESET) and at \(t_0\) higher conductance values are programmed with high signal to noise ratio [35]. This enables lower error with max-fill relative to equal-fill algorithms at \(t_0\) (figure 4(a)). Max-fill with error correction achieves further reduction in MVM error. Error correction can be more effective when the error from a higher significance slice is corrected by a lower significance slice, such as for \(n_W = 2, 4\) with \(b_W = 2\). All these trends are clearly visible at \(t_0\) in figure 4(a). However, at one month, the difference between the different programming schemes reduces and equal significance \(b_W = 1\) with equal-fill begins to show lower error. We interpret this as an effect of conductance drift becoming the dominant noise source at later points in time. Drift acts as a multiplicative noise and causes the programmed states to diverge. The drift divergence is higher at higher conductance values [35] reducing the advantages of max-fill. Since the optimum weight slicing for multiplicative noise is equal significance with equal-fill (supplementary note II), it is expected that it leads to the lowest error when the drift divergence dominates over the other sources of errors.

Next, we evaluate in more detail the effect of the base of the weight slices on MVM accuracy when using analog weight slicing. Figure 4(b) shows the relative error from max-fill relative to equal-fill algorithms at \(t_0\) (figure 4(a)). Max-fill with error correction achieves further reduction in MVM error. As discussed earlier, the error correction of programming noise can be more effective when the correction is performed by a slice of lower significance. However, as the base increases, we lose the advantage of averaging and the error from the most significant slice starts to dominate. As a result, the base increases, the error starts to increase towards that of the \(n_W = 1\) scenario.

The full time evolution of the error for the best performing combinations of base and weight programming algorithms is shown in figure 4(c). Due to drift variability, the relative error increases by approximately 8% over one year when using only one weight slice. Increasing the number of slices reduces both the initial relative error as well as the rate of increase of the error over time when \(b_W = 1\). Max-fill with EC using \(b_W = 2\) reduces the initial error further for \(n_W = 2\), but the rate of error increase over time is higher than for \(b_W = 1\). Therefore, the benefits of the error correction vanish over time, and using \(b_W > 1\) for \(n_W > 2\) becomes detrimental.
Finally, we experimentally verify the effectiveness of analog weight slicing on a prototype multi-level PCM chip. Our experiments are conducted on a chip containing more than 1 million PCM devices. Using a Gaussian weight matrix of size (128, 128), weight slices are first defined in software using the max-fill algorithm, and then programmed to the PCM chip as conductance values using a program-and-verify algorithm [5]. The programmed conductance values from the chip are serially read over time and then reported to software for computing the MVM results. Figure 4(d) shows the relative MVM error as a function of number of weight slices $n_W$ for $b_W = 1, 2$, obtained with 1000 randomly generated input vectors. The trends observed in simulations match the experimentally observed behaviour. The relative error reduces consistently with increasing number of weight slices, similar to the simulations in figure 4(a). The base $b_W$ outperforms $b_W = 2$ for $n_W = 4, 8$ in both the simulations and experiments, with an error reduction of 4.9% and 16.9% for $n_W = 4, 8$, respectively. The reduction in error from $b_W = 1$ with respect to $b_W = 2$ also increases over time, consistent with the simulation results. The slight discrepancies in the relative error values observed between the experiment and the simulations could be attributed to devices for which the iterative programming algorithm did not converge. While the simulation model assumes 100% convergence, the average convergence for all the experiments is 98.9%, with lower convergence recorded (approximately 97%) for larger absolute weight values, which are
more prominent when using higher $n_W$. Moreover, only a single set of weight slices was programmed to the PCM hardware, although in the simulations statistics were gathered from 1000 random weight matrices.

### 3.2. DNN inference accuracy

The effectiveness of bit slicing is further evaluated when employing multiple PCM crossbar arrays to perform DNN inference. Slicing the inputs and weights alters the distribution of the analog currents from the crossbar columns that represent the MVM results. Therefore, it is necessary to properly set the DAC/ADC ranges when performing DNN inference. While it is possible to optimize the DAC ranges through hardware-aware training of the DNNs, for this study the ranges were set post training.

For the DNN simulations, a single input slice is used. We computed the 99.995th percentile of the input for each weight layer based on 10 000 training images. For inference simulations, the input to each crossbar array is linearly scaled such that the 99.995th percentile value maps to the maximum DAC range, and quantized according to the DAC precision (9 bit). For the weight mapping, the value range of each weight slice is linearly mapped to the conductance range $[-G_{\text{max}}, G_{\text{max}}]$ of a differential PCM pair. The weight values are simulated using the full PCM model described in section 2.3.3. Based on the number of slices, a single ADC range for all crossbars is determined as described in section 2.3.2. The MVMs from each weight slice and input slice are scaled using equation (9) and quantized to integers based on the ADC bit resolution (8 bit). The ADC outputs from the slices are then processed as shown in figure 2 to determine the actual weight layer response.

The different weight slicing schemes were evaluated for inference with ResNet-32 performing classification of CIFAR-10 dataset, and ResNet-34 performing classification of ImageNet dataset. The network was trained in a hardware-aware manner by injecting noise and weight clipping [5]. The weights for the ResNet-32 and ResNet-34 DNNs were obtained by additive noise training which adds zero mean random noise of standard deviation 3.8% of the weight range and perform clipping to the weights during forward propagation [5]. The ResNet-32 weights were clipped to two standard deviation and ResNet-34 weights were clipped to 2.5 standard deviations of their respective weight distributions after every training batch. The DNN weights were unrolled into a 2D matrix [37], sliced, and mapped to PCM crossbar arrays of fixed size, $N$. We used $N = 128$ for ResNet-32 and $N = 512$ for ResNet-34. When the weight matrix size was larger than the crossbar array, the matrix was divided equally into multiple crossbar arrays. Since the size of the weight matrix changes from layer to layer,
and between DNNs, some of the crossbar memory arrays were only partially filled. However, we used a fixed ADC range based on the $N$ for all the crossbar arrays in the simulation. The effect of ADC was less than 0.05% in CIFAR-10 classification accuracy and less than 0.3% for ImageNet top-1 classification accuracy (see supplementary note V). GDC was applied as described in section 2.3.4, by applying an all-constant calibration input along with each batch of input. The MVM result from the calibration input immediately after programming the weights to the array was used as the reference calibration vector $O_{\text{cal}}(t_0)$.  

Figures 5(a) and (b) show the temporal evolution of test accuracy when the models were mapped to the bit slicing simulator on CIFAR-10 and ImageNet for increasing number of weight slices of equal significance with max-fill algorithm. As for the previous MVM results, a larger number of slices helps to increase the accuracy retention and reduces the variability across different hardware instances. In figures 5(c) and (d), we show the test accuracies at $t_0$ and at one month for different slice programming schemes for equal ($b_W = 1$) and varying ($b_W = 2$) significance. For CIFAR-10 at $t_0$, max-fill or max-fill with EC for $b_W = 1$ work best. The relative benefit between them in terms of test accuracy score is less than the error introduced by the ADCs. However, after one month, the equal-fill with $b_W = 1$ seems to perform at least equally well as observed previously. For ImageNet, even though the relative advantage in terms of classification accuracy for the different slicing schemes seems to be less than the ADC error, $b_W = 1$ with equal-fill seems to perform best at later time points as well.

4. Discussion

In this work, we evaluated the precision of the bit slicing approach on an in-memory computing architecture. In contrast to the bit slicing in a digital processor, whose aim is mainly to extend the dynamic range of the number representation, the primary challenge here is stochasticity arising from the analog memory devices used to represent the weights. Our work focuses on inference applications, where the weight matrix is expected to be constant. A gradient descent based training, typically employed for DNNs, could benefit from a synapse representation with high dynamic range for finer weight updates. However during inference, which employs a linear summation of input features in each layer, a lower dynamic range is sufficient. As shown in this study, what is critical during inference is to minimize the error arising from the analog memory devices.

Although we performed the evaluations using a model of PCM devices, the results can be generalized to other multi-level analog memory devices considering the following aspects. The weight slices in general can be programmed in two ways, either each slice is programmed independently, or using an error correction scheme where the subsequent slice values are determined such that they correct the error from previous slices. For independent programming, averaging between multiple slices reduces the error, therefore $b_W = 1$ works best. As $b_W$ increases above 1, the weight error is dominated by that from the most significant slice, reducing any averaging effect. Within equal significant slices, whether we should follow an equal-fill or max-fill programming approach depends on the state-dependence of the programming error. If the programming error is proportional to the conductance value (multiplicative noise), the MVM error will be minimized for $b_W = 1$ with equal-fill (supplementary note II). However, when the programming error is independent of conductance value, it is better to program as low a number of devices as possible with max-fill. The state-dependence of noise in our PCM devices at $t_0$ is in-between to these two cases, and the minimal error is obtained for max-fill when $b_W = 1$. However, when we include the error correction, $b_W = 2$ minimizes the MVM error at $t_0$ (when there is no drift) as shown in figure 4(b). This is because the error correction is more effective when the error is corrected by a slice of lower significance. However, as $b_W$ increases, the error starts to be outside the value range of subsequent slices, causing the MVM error to increase further for $b_W > 2$. As time elapses, the PCM exhibits conductance drift and at larger time scales, say a month, the error caused by the drift begins to dominate. Since the drift appear as a multiplicative noise, $b_W = 1$ with equal-fill begins to minimize the MVM error as expected.

This paper focuses on the MVM accuracy of analog in-memory computing through input and weight slicing techniques. Although architectural optimization is not the primary focus of this work, the proposed schemes do have an impact on design of the analog and digital components of the in-memory accelerator. Bit-slicing on memristive crossbar arrays allows DNN accuracies to be maintained over a longer period in time, which further strengthens the case for using non-volatile memories for in-memory computing. The much slower accuracy degradation relaxes the requirements on how frequently the non-volatile memory arrays should be re-programmed to maintain high accuracy. Moreover, employing bit-slicing offers advantages for the array periphery. Input slicing enables reducing the resolution of DAC circuitry, allowing even 1 bit inverters to be used. This can be especially beneficial for integration with tight memory-cell pitches. On the other hand, weight slicing increases the number of memory devices per weight and thus increases the crossbar size, unless the technology permits vertical stacking [38]. Weight slicing will also increase the energy consumption because more power will be dissipated with higher number of memory devices per weight. However, this overhead can
be reduced by maximizing the number of zero slices. Although the number of required ADCs remains the same, each additional weight or input slice necessitates extra A/D conversions that will negatively impact throughput and energy-efficiency. To sum up, the minimum required computational precision as well as the time, energy and area budget will all impact the core design and implementation choices.

An interesting avenue would be to design in-memory computing chips such that the number of input and weight slices could be dynamically changed, enabling design optimization across applications. This flexibility can be exploited within an application too, for example by performing the precision critical part of DNN inference with increased number of input and weight slices. Such a mixed-precision implementation can be executed on multi-core in-memory-based accelerators where each convolution layer of a ResNet is mapped to an in-memory computing core and the activations are propagated from layer-to-layer [39]. The precision of the first and last layers are critical for high accuracy [5], and therefore could be performed with higher number of input and weight slices in comparison to the other layers.

This paper focused on bit slicing strategies for performing MVMs when the elements of the matrix are stationary. Yet, there are applications such as DNN training where the weight matrix is frequently updated [40]. For DNN training, a higher precision (typically 8 bits and above) is critical to achieve competitive accuracies [41, 42]. Reference [16] employs input slicing with $b_I = 1$ and $n_I = 16$ and weight slicing with $b_W = 2$ and $n_W = 4$ with 4 bit RRAM, to achieve an effective precision of 16 bits for both the inputs and weights. In such implementations with $b_W > 1$, the current weight has to be read, added to the weight gradient and the new weights are reprogrammed in the array at each weight update phase to avoid any overflow. This incurs time and energy, which can be to some extent remedied by employing batch training [16, 43–46] or other mixed-precision training schemes [47] where device updates are performed less frequently. Another approach is to allocate a portion of the weight slices to overflows [24, 17], but this requires periodically reading and reprogramming the whole array during training. It has been shown that implementations with $b_W = 1$ can avoid reading the array before every weight update (i.e. perform blind updates without any verification) [18]. Yet, the achieved dynamic range is lesser than in implementations with $b_W > 1$. In general, the best performing bit slicing strategies and their respective energy/latency tradeoffs differ significantly for inference and training, hence separate studies are needed to assess their relevance for each application.

5. Conclusion

In summary, we evaluated the precision of bit slicing when implemented on analog PCM crossbars for in-memory computing, focusing on DNN inference applications where the weight matrix is constant. We showed that the proper setting of ADC quantization ranges, which need to be different when using different number of slices, is critical for obtaining precision benefits from bit slicing. Moreover, we show that performing bit slicing on the input vector applied to the crossbar array does not significantly affect the overall computational accuracy. However, weight slicing is effective in reducing the random errors due programming inaccuracies, conductance noise and drift. Importantly, using weight slices of equal significance (base of 1) is best for mitigating the errors from noise and drift through averaging, especially when a high number of slices is used. In the specific case of PCM, optimized weight programming algorithms were shown to lead to lower error just after programming, but over time the error was mostly determined by the base of the slices and was the lowest with a base of 1. This shows that in general, bit slicing should be done differently on analog in-memory computing hardware compared with conventional digital processors to obtain optimal precision benefits. The next steps will be to derive programming strategies for weight slicing that give the optimal accuracy for any given network, and to demonstrate them on fully-integrated DNN inference PCM chips [3, 48].

Acknowledgments

We thank our colleagues at IBM Research - Europe, IBM Research - Almaden and IBM TJ Watson Research Center, in particular, U Egger, N Papandreou and A Petropoulos for experimental help, M BrightSky for help with fabricating the PCM prototype chip used in this work, and M Rasch for technical input.

Data availability statement

The data that support the findings of this study are available upon reasonable request from the authors.
References

[1] Yu S 2018 Neuro-inspired computing with emerging nonvolatile memories Proc. IEEE 106 260–85
[2] Sebastian A et al 2020 Memory devices and applications for in-memory computing Nat. Nanotechnol. 15 529–44
[3] Khaddam-Aljehani R et al 2021 Hermes core—a 14 nm CMOS and PCM-based in-memory compute core using an array of 300 ps/LSB linearized CCO-based ADCs and local digital processing 2021 Symp. VLSI Circuits pp 1–2
[4] Jia H et al 2021 15.1 a programmable neural-network inference accelerator based on scalable in-memory computing 2021 IEEE Int. Solid-State Circuits Conf. (ISSCC) vol 64 pp 236–8
[5] Joshi V et al 2020 Accurate deep neural network inference using computational phase-change memory Nat. Commun. 11 2473
[6] Yao P, Wu H, Gao B, Tang J, Zhang Q, Zhang W, Yang J and Qian H 2020 Fully hardware-implemented memristor convolutional neural network Nature 577 641–6
[7] Le Gallo M, Sebastian A, Cherubini G, Giefer H and Eleftheriou E 2018 Compressed sensing with approximate message passing in using-in-memory computing IEEE Trans. Electron. Devices 65 4304–12
[8] Le Gallo M, Sebastian A, Mathis R, Manica M, Giefer H, Tutma C, Bokus C, Curioni A and Eleftheriou E 2018 Mixed-precision in-memory computing Nat. Electron. 1 246
[9] Giannopoulos I et al 2018 8 bit precision in-memory multiplication with projected phase-change memory Proc. IEEE Int. Electron Devices Meeting (IEDM) pp 27.7.1–27.7.4
[10] Zidan M A and Lu W D 2020 Vector multiplications using memristive devices and applications thereof Memristive Devices for Brain-Inspired Computing: From Materials, Devices, and Circuits to Applications—Computationa...pp 1–9
[11] Mick J and Brick J 1980 1980 Bit-Slice Microprocessor Design (New York: McGraw-Hill)
[12] Boinondi M N and Ipek E 2016 Memristive Boltzmann machine: a hardware accelerator for combinatorial optimization and deep learning. 2016 IEEE Int. Symp. High Performance Computer Architecture (HPCA) (Piscataway, NJ; IEEE) pp 1–13
[13] Shafiee A, Nag A, Muralimohanor N, Balasubramonian R, Strachan J P, Hu M, Williams R S and Srikumar V 2016 ISAAC: a convolutional neural network accelerator with in situ analog arithmetic in crossbars 2016 ACM/IEEE 43rd Annual Int. Symp. Computer Architecture (ISCA) pp 14–26
[14] Chi P et al 2016 PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA) pp 27–39
[15] Yu S et al 2016 Binary neural network with 16 mb ReRAM macro chip for classification and online training 2016 IEEE Int. Electron Devices Meeting (IEDM) 16.2 pp 1–4
[16] Song L, Qian X, Li H and Chen Y 2017 Pipelayer: a pipelined ReRAM-based accelerator for deep learning 2017 IEEE Int. Symp. High Performance Computer Architecture (HPCA) pp 541–52
[17] Agarwal S et al 2017 Achieving ideal accuracies in analog neuromorphic computing using periodic carry 2017 Symp. VLSI Technology pp T174–5
[18] Boybat I et al 2018 Neuromorphic computing with multi-memristive synapses Nat. Commun. 9 2514
[19] Feinberg B, Vengalam U K R, Whitehair N, Wang S and Ipek E 2018 Enabling scientific computing on memristive accelerators 2018 ACM/IEEE 45th Annual Int. Symp. Computer Architecture (ISCA) (Piscataway, NJ; IEEE) pp 367–82
[20] Mackin C, Tasi H, Ambrogio S, Narayanan P, Chen A and Burr G W 2019 Weight programming in DNN analog hardware accelerators in the presence of NVM variability Adv. Electron. Mater. 5 1900026.
[21] Zidan M A, Jeong Y, Lee J, Chen B, Huang S, Kushner M J and Lu W D 2018 A general memristor-based partial differential equation solver Nat. Electron. 1 411
[22] Yoon J H et al 2021 29.1 a 40 nm 64 kb 56.67 TOPS/W read-disturb-tolerant compute-in-memory/digital ReRAM macro with active-feedback-based read and in situ write verification 2021 IEEE Int. Solid-State Circuits Conf. (ISSCC) vol 64 pp 404–6
[23] Xue C X et al 2021 16.1 a 22 nm 4 mb 8b-precision ReRAM computing-in-memory macro with 11.91 to 195.7 TOPS/W for tiny AI edge devices 2021 IEEE Int. Solid-State Circuits Conf. (ISSCC) vol 64 pp 245–7
[24] Ambrogio S et al 2018 Equivalent-accuracy accelerated neural-network training using analogue memory Nature 558 60
[25] Gonugondla S K, Patil A D and Shanbhag N R 2020 Swipe: enhancing robustness of ReRAM crossbars for in-memory computing 2020 IEEE/ACM Int. Conf. Computer Aided Design (ICCAD) pp 1–9
[26] Zhang W et al 2021 Array-level boosting method with spatial extended allocation to improve the accuracy of memristor based computing-in-memory chips Sci. China Inf. Sci. 64 1–9
[27] Pedretti G, Ambrosi E and Lelmi D 2021 Conductance variations and their impact on the precision of ReRAM with switching resistance memory (RRAM) 2021 IEEE Int. Reliability Physics Symp. (IRPS) pp 1–8
[28] Pfeiffer M and Pfel T 2018 Deep learning with spiking neurons: opportunities and challenges Front. Neurosci. 12 774
[29] Rajendran B, Sebastian A, Schmucker M, Srinivasra N and Eleftheriou E 2019 Low-power neuromorphic hardware for signal processing applications: a review of architectural and system-level design approaches IEEE Signal Process. Mag. 36 97–110
[30] Sebastian A, Le Gallo M, Burr G W, Kim S, BrightSky M and Eleftheriou E 2018 Tutorial: brain-inspired computing using phase-change memory devices J. Appl. Phys. 124 111101
[31] Chen J, Wu H, Gao B, Tang J, Hu X S and Qian H 2020 A parallel multibit programming scheme with high precision for RRAM-based neuromorphic systems IEEE Trans. Electron Devices 67 2213–7
[32] Bruce R et al 2021 Mushroom-type phase change memory with projection liner: an array-level demonstration of conductance drift and noise mitigation IEEE Int. Reliability Physics Symp. (IRPS) pp 1–6
[33] Nandakumar S R et al 2019 Phase-change memory models for deep learning training and inference 26th IEEE Int. Conf. Electronics, Circuits and Systems (ICECS) pp 727–30
[34] Suri M et al 2011 Phase change memory as synapse for ultra-dense neuromorphic systems: application to complex visual pattern extraction IEEE Int. Electron Devices Meeting (IEDM) (Piscataway, NJ): IEEE 4.4 pp 1–4
[35] Nandakumar S R et al 2020 Precision of synaptic weights programmed in phase-change memory devices for deep learning inference IEEE Int. Electron Devices Meeting (IEDM) pp 29.4.1–29.4.4
[36] Ambrogio S et al 2019 Reducing the impact of phase-change memory conductance drift on the inference of large-scale hardware neural networks 2019 IEEE Int. Electron Devices Meeting (IEDM) 6.1 pp 1–4
[37] Gokmen T, Onen M and Haensch W 2017 Training deep convolutional neural networks with resistive cross-point devices Front. Neurosci. 11 1–22
[38] Adam G C, Hoskins B D, Prezioso M, Merrikh-Bayat F, Chakrabarti B and Strukov D B 2017 3D memristor crossbars for analog and neuromorphic computing applications IEEE Trans. Electron Devices 64 312–8
[39] Dazzi M, Sebastian A, Benini L, Eleftheriou E 2021 Accelerating inference of convolutional neural networks using in-memory computing Frontiers in Computational Neuroscience 15 674154
[40] Li J et al 2021 Reduction 93.7% time and power consumption using a memristor-based imprecise gradient update algorithm Artif. Intell. Rev. 55 657–77
[41] Wang N, Choi J, Brand D, Chen C-Y and Gopalakrishnan K 2018 Training deep neural networks with 8 bit floating point numbers Advances in Neural Information Processing Systems pp 7675–84
[42] Sun X et al 2020 Ultra-low precision 4 bit training of deep neural networks Advances in Neural Information Processing Systems vol 33 ed H Larochelle, M Ranzato, R Hadsell, M F Balcan, H Lin pp 1796–807
[43] Prezioso M, Merrikh-Bayat F, Hoskins B D, Adam G C, Likharev K K and Strukov D B 2015 Training and operation of an integrated neuromorphic network based on metal-oxide memristors Nature 521 61–4
[44] Yao P et al 2018 Face classification using electronic synapses Nat. Commun. 8 15199
[45] Li C et al 2018 Efficient and self-adaptive in situ learning in multilayer memristor neural networks Nat. Commun. 9 2385
[46] Wang Z et al 2019 In situ training of feed-forward and recurrent convolutional memristor networks Nat. Mach. Intell. 1 434–42
[47] Nandakumar S et al 2020 Mixed-precision deep learning based on computational memory Front. Neurosci. 14 406
[48] Narayanan P et al 2021 Fully on-chip MAC at 14 nm enabled by accurate row-wise programming of PCM-based weights and parallel vector-transport in duration-format 2021 Symp. VLSI Technology pp 1–2