mlGeNN: accelerating SNN inference using GPU-enabled neural networks

Article (Accepted Version)

Turner, James Paul, Knight, James Courtney, Subramanian, Ajay and Nowotny, Thomas (2022) mlGeNN: accelerating SNN inference using GPU-enabled neural networks. Neuromorphic Computing and Engineering. ISSN 2634-4386

This version is available from Sussex Research Online: http://sro.sussex.ac.uk/id/eprint/104723/

This document is made available in accordance with publisher policies and may differ from the published version or from the version of record. If you wish to cite this item you are advised to consult the publisher’s version. Please see the URL above for details on accessing the published version.

Copyright and reuse:
Sussex Research Online is a digital repository of the research output of the University.

Copyright and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable, the material made available in SRO has been checked for eligibility before being made available.

Copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational, or not-for-profit purposes without prior permission or charge, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

http://sro.sussex.ac.uk
mlGeNN: accelerating SNN inference using GPU-enabled neural networks

To cite this article before publication: James Paul Turner et al 2022 Neuromorph. Comput. Eng. in press https://doi.org/10.1088/2634-4386/ac5ac5

Manuscript version: Accepted Manuscript

Accepted Manuscript is “the version of the article accepted for publication including all changes made as a result of the peer review process, and which may also include the addition to the article by IOP Publishing of a header, an article ID, a cover sheet and/or an ‘Accepted Manuscript’ watermark, but excluding any other editing, typesetting or other changes made by IOP Publishing and/or its licensors”

This Accepted Manuscript is © 2022 The Author(s). Published by IOP Publishing Ltd.

As the Version of Record of this article is going to be / has been published on a gold open access basis under a CC BY 3.0 licence, this Accepted Manuscript is available for reuse under a CC BY 3.0 licence immediately.

Everyone is permitted to use all or part of the original content in this article, provided that they adhere to all the terms of the licence https://creativecommons.org/licences/by/3.0

Although reasonable endeavours have been taken to obtain all necessary permissions from third parties to include their copyrighted content within this article, their full citation and copyright line may not be present in this Accepted Manuscript version. Before using any content from this article, please refer to the Version of Record on IOPscience once published for full citation and copyright details, as permissions may be required. All third party content is fully copyright protected and is not published on a gold open access basis under a CC BY licence, unless that is specifically stated in the figure caption in the Version of Record.

View the article online for updates and enhancements.
mlGeNN: Accelerating SNN inference using GPU-Enabled Neural Networks

James Paul Turner¹, James C Knight¹, Ajay Subramanian² and Thomas Nowotny¹
¹ Centre for Computational Neuroscience and Robotics, School of Engineering and Informatics, University of Sussex, Brighton, United Kingdom
² Department of Psychology, New York University, New York, New York, 10003, USA
E-mail: J.P.Turner@sussex.ac.uk

Abstract. In this paper we present mlGeNN – a Python library for the conversion of artificial neural networks (ANNs) specified in Keras to spiking neural networks (SNNs). SNNs are simulated using GeNN with extensions to efficiently support convolutional connectivity and batching. We evaluate converted SNNs on CIFAR-10 and ImageNet classification tasks and compare the performance to both the original ANNs and other SNN simulators. We find that performing inference using a VGG-16 model, trained on the CIFAR-10 dataset, is 2.5× faster than BindsNet and, when using a ResNet-20 model trained on CIFAR-10 with FewSpike ANN to SNN conversion, mlGeNN is only a little over 2× slower than TensorFlow.
1. Introduction

For image classification, deep Convolutional Neural Networks (CNNs), trained on large datasets such as ImageNet2012 [1] are the gold standard. However, even after a CNN has been trained, using it for inference requires on the order of millions (if not billions) of multiply-accumulate operations (see Hanif et al. [2], figure 13.5) to propagate real-valued unit activations through many layers of trained weights. This results in computation and energy [3] requirements that make them unsuitable for many mobile and edge-computing applications and raises concerns about the carbon footprint of neural network algorithms. Various approaches have been employed to reduce compute time and energy cost, including simplifying model structure [4], using low-precision data types [5] and even specialised accelerator hardware [6].

Spiking Neural Networks (SNNs) represent a more fundamental change of approach. Taking closer inspiration from biological brains, neurons in SNNs communicate using spatio-temporal patterns of discrete events known as ‘spikes’, rather than the real-valued ‘rates’ exchanged by units in deep networks. Because typically only a small subset of neurons in an SNN are active (spiking) at any given time, simulating an SNN model for a single timestep requires significantly fewer operations. Furthermore, because each spiking neuron’s activation is binary, propagating it through a layer of weights requires only addition rather than multiply-accumulate operations, which may allow much more efficient hardware implementations. A wide variety of specialised neuromorphic hardware has been developed to exploit these properties, using technologies ranging from analogue electronics [7, 8] to standard CMOS ASICs [9–11] and fully-programmable CPUs [12, 13].

Numerous techniques for directly training SNNs have been developed. Notably, back-propagation can be applied to exact spike times [14–16] or the non-differentiable transfer function of each spiking neuron can be replaced with a ‘surrogate gradient’ function, allowing back-propagation through time [17, 18] or more biologically-plausible learning rules [19–21] to be applied to spiking models. However, none of these techniques are yet capable of training deep feedforward SNNs. For these models, an alternative approach is to convert fully-trained traditional deep networks to SNNs. The most common method is to translate the activation of the units in a deep network to firing rates of spiking neurons in an SNN of the same geometry [22–24]. Although good inference performance has been obtained using transfer methods of this kind, the resulting SNNs need to be simulated for hundreds if not thousands of timesteps before a classification is obtained. This overhead cancels out the lower cost of each SNN timestep compared to each ANN step. An alternative approach is to translate the deep network model in such a way that the activations of ANN units are encoded in the spike times of the neurons in the SNN model [25, 26]. By using precise spike times, activations can be represented in many fewer timesteps allowing much more competitive speed of inference.

Spiking neural networks are likely to be ultimately deployed on specialised
mlGeNN

neuromorphic hardware, but researchers still need efficient tools for developing the conversion and training algorithms on standard hardware. There are a large number of SNN simulators [27–30] but most of them are designed for Computational Neuroscience applications and, as such, lack native support for common machine learning building blocks like convolutional layers and batching. Therefore, many ML researchers have chosen to develop alternative spike-based ML libraries [31–34] on top of ML platforms such as PyTorch [35], TensorFlow [36] or JAX [37], so that they can continue to take advantage of the ML infrastructure these platforms provide. These SNN packages each have their own strengths and weaknesses, and many specialise in simulating specific classes of models. For instance, SpykeTorch [34] is a Python package which specialises in training SNN models where each neuron produces at most one spike per stimulus whereas BindsNET [31] aims to be a more general purpose SNN simulation package.

The underlying ML platforms all work by assembling computational graphs from operations, such as matrix multiplications and reductions, operating on a tensor datatype. These operations are implemented as highly-optimised GPU kernels and, to ensure that they fully occupy GPU resources and to counteract the overheads of launching large numbers of kernels, operations are ‘batched’ across an additional tensor dimension. This is a very efficient approach for simulating feedforward ANNs as only a single traversal of the model is required for each input tensor. However, the employed tensor types only have minimal support for sparsity meaning that, to simulate SNNs within these frameworks, the activity of spiking neurons is typically represented as dense tensors with many zeros (no spikes) and a few ones (spikes). These tensors are then propagated through the weights of the network using the standard matrix multiplication based approach. Therefore, each SNN timestep requires the same number of arithmetic operations as a complete traversal of an equivalent ANN. As SNN simulations involve processing multiple timesteps, they will typically be slower than an equivalent ANN by a factor equal to the number of required timesteps when using libraries built on top of standard ML platforms. Nonetheless, due to the highly-optimised kernels and powerful optimization tools provided by modern ML frameworks, this approach has proved popular for simulating small models.

In this work we present mlGeNN – a framework for converting feedforward ANN models, defined in Keras, to SNN models. It is freely available online at https://github.com/genn-team/ml_genn. Under the hood, rather than using a tensor based ML library to provide GPU acceleration, mlGeNN uses PyGeNN [38] – a Python interface to the GeNN GPU-accelerated SNN simulator [39] which has previously demonstrated excellent performance at accelerating a range of computational neuroscience models [40, 41]. We have extended GeNN with two methods for efficiently implementing convolutional connectivity. Firstly, we have used the ‘procedural connectivity’ approach, previously employed for simulating extremely large computational neuroscience models [41], to directly implement 2D convolutional connectivity. Secondly, we have implemented a new parallelism strategy tailored to the sparse structure of 2D convolutional connectivity. Finally, we have added batching
mlGeNN

| SM | L1 cache | SM | L1 cache | SM | L1 cache |
|----|----------|----|----------|----|----------|
|    |          |    |          |    |          |
|    |          |    |          |    |          |
|    |          |    |          |    |          |
|    |          |    |          |    |          |
|    |          |    |          |    |          |
|    |          |    |          |    |          |
|    |          |    |          |    |          |
|    |          |    |          |    |          |
|    |          |    |          |    |          |
|    |          |    |          |    |          |
|    |          |    |          |    |          |
|    |          |    |          |    |          |

L2 cache

DRAM

**Figure 1.** Organisation of arithmetic and memory compute resources into ‘streaming multiprocessor’ (SM) units on NVIDIA GPUs.

support to GeNN. With these improvements and because GeNN has been optimised for spatial sparseness of connections and temporal sparseness of spiking it can considerably increase the simulation speed of SNNs. However, as we will demonstrate below, in spite of large improvements over using standard ML platforms, the required multiple timesteps in SNN simulations mean that SNN simulations remain slower than ANNs on GPU hardware.

2. Methods

2.1. GPU Architecture

GeNN (and therefore mlGeNN) utilises NVIDIA GPUs to compute neuronal and synaptic updates. As such, GPU SNN simulations using GeNN have a significant performance advantage over equivalent CPU implementations [40]. NVIDIA provides the CUDA language – an extension of the C++ language – for writing programs which will run on NVIDIA GPUs.

An obstacle to CUDA and other GPU programming is the required domain expertise to maximise an application’s GPU resource usage. While CPU architectures are optimised for low-latency computation, with fast local caches and branch prediction but just a few processor cores, GPU architectures are optimised for computational throughput, with vast pools of arithmetic cores and complex hierarchical memory systems to keep them fed with data. At the highest level of the memory hierarchy,
global DRAM memory has the largest capacity but also the lowest memory bandwidth and highest latency. Special care needs to be taken to ensure that global memory reads and writes follow strict patterns (referred to as ‘coalesced’ memory access) to utilise the GPUs’ DMA circuitry most efficiently and minimise access latency. As shown in figure 1, the computational resources of NVIDIA GPUs are arranged into units called ‘streaming multiprocessors’ (SM), where each SM has a pool of relatively fast shared and cache memory, a number of arithmetic cores and a large register file.

Unlike typical parallel CPU programming, where each thread of the application executes different instructions on data from potentially different memory regions, NVIDIA follows the single instruction multiple data (SIMD) paradigm. The concept of the function is replaced with the kernel, in which a single set of instructions is executed concurrently by many processor cores on largely monolithic memory segments. When a CUDA kernel is executed, the computation is organised into sets of threads called ‘warps’, which are divided amongst available SMs. The SM’s warp scheduler then allocates registers and shared memory to warps and schedules them for execution. Execution is fastest when each thread has identical control flow – i.e. follows the same branch in conditional statements – and must be repeated for each divergence in control flow.

The existing GPU-based SNN optimisations of GeNN provide very fast and memory efficient SNN models and hides the complexity of GPU programming from the end user, allowing them to focus entirely on model definition and evaluation.

2.2. Datasets

In our experiments, we use the CIFAR-10 [42] and ImageNet [1] ILSVRC datasets. CIFAR-10 [42] is a dataset of 60,000 low-resolution (32 × 32) RGB colour images divided into 10 image classes, with 6000 images per class. The set is divided into 50,000 training images and 10,000 validation images. CIFAR-10 is a very popular simple image classification dataset, often used when prototyping image classification ANN architectures. The ImageNet [1] ILSVRC dataset is a much larger dataset, consisting of 1281167 training images, 50,000 validation images and 1000 classes. Being more challenging, it is often used to both demonstrate the performance of ANNs on more realistic data and as a basis for transfer learning.

Each dataset is pre-processed before use, following the method described by Sengupta et al. [24]. CIFAR-10 images are pre-processed to have zero mean and unit variance in each colour channel as well as being randomly cropped and randomly horizontally flipped with a probability of 0.5. ImageNet images are pre-processed in the same way but, additionally, each image has AlexNet-style PCA-based noise applied [43].

2.3. Models

In our experiments, our base ANN models follow those used by Sengupta et al. [24], which are in turn based on the VGG [44] and ResNet [45] architectures. Due to the
Figure 2. The architectures of the VGG-like (left) and ResNet-like (right) ANNs used in our experiments. The notation \( \text{/2} \) indicates that downsampling is performed using a two-stride 1x1 convolution layer.

Constraints of the rate-based ANN to SNN conversion methods discussed in section 2.6, all layers in the ANN model must have ReLU activations and not use bias tensors, with max pooling layers replaced by average pooling layers. Because batch normalisation cannot be used without biases, it is replaced with dropout.

The VGG-like ANN, illustrated in figure 2 (left), is essentially VGG-16 [44] modified to meet the above constraints. It is a convolutional neural network (CNN) consisting of several blocks of convolution layers, separated by average pooling layers with a stride of 2, and topped with a final fully connected block. The number of feature maps in each of the five convolution blocks is 64, 128, 256, 512, 512, and the first two layers of the fully-connected block have 4096 units. Each convolution or fully-connected layer not preceded by a pooling layer is, instead, preceded by a dropout layer with a dropout probability of 0.25. When instantiated with an input size of 32 \times 32 (e.g. for CIFAR-10), this model has 287 754 neurons and, with an input size of 224 \times 224 (e.g. for ImageNet), it has 13 707 240 neurons.

For the ResNet-like ANNs, the architectures again follow those described by Sengupta et al. [24]. The ResNet-like architecture for classifying the CIFAR-10 dataset (see section 2.2) is shown in figure 2 (right). It has 224 266 neurons and begins with a block of initial pre-processing convolution layers. These are followed by a series of
mlGeNN

residual blocks [45] which consist of two pathways – a non-identity pathway consisting of two convolution layers, and an identity skip pathway – which converge via add layers. Finally, the network is topped with a global average pooling layer followed by a fully-connected output layer. In each residual block, the first convolution layer of the non-identity path has ReLU activation, as usual, but the ReLU activation of the second convolution layer in the path is deferred until after the junction between the identity and non-identity pathways. This corresponds to ‘version one’ of ResNet, given in the original paper [45]. As in the VGG-like ANNs, the ResNet-like models periodically downsample feature maps while increasing the number of said maps. However, rather than pooling layers, ResNet ANNs use $1 \times 1$ convolutions with a stride of two to downsample layers. This corresponds to ‘type B’ identity pathways described by He et al. [45]. The model has 16 feature maps in the first three residual blocks, followed by 32 in the following three blocks, followed by 64 in the last three residual blocks. Dropout is applied after the first non-identity convolution layer in each residual block, with probability 0.25.

The ResNet-like ANN for classifying the ImageNet dataset is similar, except that the nine residual blocks are replaced as follows: the first three blocks have 64 feature maps, followed by four blocks with 128 feature maps, followed by six blocks with 256 feature maps, and finally three blocks with 512 maps. In addition, a stride-two average pooling layer is added after the initial pre-processing layers. This results in a model with 10 136 552 neurons.

Our training procedure mostly follows that described by Sengupta et al. [24]. ANN models are trained in TensorFlow 2.4.0 with stochastic gradient descent, with batch size 256, weight decay 0.0001, and momentum 0.9. For the CIFAR-10 dataset, the ANN is trained for 200 epochs, with an initial learning rate of 0.05 which is divided by ten after 81 epochs, and again after 122 epochs. For the ImageNet dataset, models are trained for 120 epochs, with an initial learning rate of 0.05 which is divided by ten after epochs 30, 60 and 90.

2.4. Sparse connectivity for efficient convolutions

Convolutional layers are a key component of almost all modern deep network architectures [43–45]. Therefore, providing efficient means to propagate activity through these layers is vital for any ML framework. The multi-channel 2D convolution operation used in these layers can be described as follows:

$$y(l, m, n) = \sum_{i} K_{\text{height}} \sum_{j} K_{\text{width}} \sum_{k} I_{\text{chan}} k(i, j, k, n) x(l + i, m + j, k)$$

where $x$ is a $I_{\text{height}} \times I_{\text{width}} \times I_{\text{chan}}$ input tensor, $k$ is a $K_{\text{height}} \times K_{\text{width}} \times I_{\text{chan}} \times O_{\text{chan}}$ kernel and $y$ is a $O_{\text{height}} \times O_{\text{width}} \times O_{\text{chan}}$ output tensor. Many different approaches have been taken to efficiently implement convolution operations on GPUs. The first ML libraries [46] directly implemented equation 1 but, in order to do this in efficient library code, there are lot of corner cases which need optimising and tuning for ever-evolving GPU architectures. Alternatively, convolution can be ‘lowered’
into a matrix multiplication by a doubly-blocked Toeplitz matrix generated from the convolution kernel. Memory can be temporarily allocated to hold these matrices and then the multiplication can be performed using highly-optimised general purpose matrix multiplication libraries such as cuBLAS [47]. However, for large models, the memory requirements can be prohibitive so, instead, convolution kernel weights can be expanded on the fly within the matrix multiplication kernel [48]. However, although the doubly-blocked Toeplitz matrix is very sparse, this approach still has the same computational complexity as general matrix multiplication. One approach for reducing this complexity is to compute convolutions using an equivalent Fast Fourier Transform [49]. Alternatively, in the common case where $K_{\text{width}} = K_{\text{height}} = 3$ or $K_{\text{width}} = K_{\text{height}} = 5$, algorithms based on Winograd’s minimal filtering algorithms can reduce the complexity of convolutions by up to $4 \times$ [50]. These algorithms divide each input channel into small tiles (typically $4 \times 4$ when $K_{\text{width}} = K_{\text{height}} = 3$). Each of these tiles is combined with the kernel elements from each output channel to obtain a minimal filter for each tile and these are summed across all output channels. Finally, this flattened representation is transformed back into output space by multiplying it by a much smaller matrix. TensorFlow and PyTorch both use the cuDNN library [51] to provide highly-optimised implementations of all of these algorithms for NVIDIA GPUs and switch between them based on each convolutional layer’s parameters.

As discussed previously, propagating spikes through a connection matrix is a rather different problem because only a (small) subset of the input neurons will be active during a given timestep and no multiplications are required. Existing GPU SNN simulators have numerous efficient algorithms for propagating spikes through sparse connectivity [40, 52, 53]. However, using a general purpose sparse data-structure to store convolutions is rather wasteful of memory as the same weights from the convolution kernel are duplicated in many locations in the sparse matrix. For example, implementing a VGG-16 architecture for ImageNet in this way would require over 100 GB of GPU memory.

2.4.1. Procedural connectivity In our previous work [38] we introduced ‘procedural connectivity’, a technique where neurons’ outgoing sparse random connectivity is regenerated on the fly when they spike, rather than being stored in memory. Using procedural connectivity we demonstrated how a model – so large it could previously only be simulated on a supercomputer – could be simulated on a single GPU. Here we apply the same technique to directly implement equation 1. One thread handles each incoming spike using the following algorithm (support for padding and stride has been omitted for clarity):

```plaintext
function PROCESS_SPIKE(spikeID)
    // Convert incoming spike ID to row, column and channel
    inRow ← spikeID / (Ichan * Iwidth)
    inCol ← (spikeID / Ichan) % Iwidth
    inChan ← spikeID % Ichan
```
Calculate corresponding range of output rows and columns
\[
\begin{align*}
\minOutRow & \leftarrow \min(O_{\text{height}}, \max(0, 1 + \text{inRow} - K_{\text{height}})) \\
\maxOutRow & \leftarrow \min(O_{\text{width}}, \text{inRow}) \\
\minOutCol & \leftarrow \min(O_{\text{width}}, \max(0, 1 + \text{inCol} - K_{\text{width}})) \\
\maxOutCol & \leftarrow \min(O_{\text{height}}, \text{inCol})
\end{align*}
\]

Loop through output rows, columns and channels
\[
\text{for outRow} \leftarrow \minOutRow \text{ to maxOutRow do}
\]
\[
\text{kernRow} \leftarrow \text{inRow} - \text{outRow}
\]
\[
\text{for outCol} \leftarrow \minOutCol \text{ to maxOutCol do}
\]
\[
\text{kernCol} \leftarrow \text{inCol} - \text{outCol}
\]
\[
\text{for outChan} \leftarrow 0 \text{ to } O_{\text{chan}} - 1 \text{ do}
\]
\[
\text{neuronID} \leftarrow \left( (\text{outRow} \times O_{\text{width}} \times O_{\text{chan}}) + (\text{outCol} \times O_{\text{chan}}) + \text{outChan} \right)
\]
\[
\text{genSynapse(neuronID, kernRow, kernCol, inChan, outChan)}
\]
\[
\text{end for}
\]
\[
\text{end for}
\]
\[
\text{end for}
\]
\[
\text{end function}
\]

where genSynapse will extract the correct weight from the kernel and apply it to the input of the postsynaptic neuron selected by neuronID using an atomic add operation. Because this algorithm is instantiated using GeNN’s code generator, constant terms will be hard-coded into the kernel, allowing the CUDA compiler to transform divisions by constants into more efficient instructions, unroll loops and optimise for special cases such as \(O_{\text{chan}} = 1\). Furthermore, unlike with the probabilistic connectivity investigated in our previous work [38], combining procedural convolutions with learning is entirely possible as the algorithm to back-propagate through the convolutional connectivity is very similar and equally efficient.

2.4.2. Toeplitz connectivity As discussed previously, 2D convolution is equivalent to multiplying by a doubly-blocked Toeplitz matrix. In this section, we show how convolutions are ‘lowered’ into such matrices. This explanation is based on [54] but updated to use machine-learning rather than signal-processing conventions. A Toeplitz matrix is one where the values along all diagonals are constant, i.e.
\[
\begin{pmatrix}
a_0 & a_{-1} & a_{-2} & \cdots & \cdots & \cdots & a_{-(N-1)} \\
a_1 & a_0 & a_{-1} & a_{-2} & \cdots & \cdots & \cdots \\
a_2 & a_1 & a_0 & a_{-1} & \cdots & \cdots & \cdots \\
\vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots \\
\vdots & \vdots & \vdots & \ddots & a_0 & a_{-1} & a_{-2} \\
\vdots & \vdots & \vdots & \ddots & a_1 & a_0 & a_{-1} \\
a_{M-1} & \cdots & \cdots & \cdots & a_2 & a_1 & a_0
\end{pmatrix}
\]

Furthermore, if we build a matrix \(A\) out of Toeplitz sub-matrices \(A_k\) and the structure
Global memory

| Shared memory | 0 | 2 | 4 | 5 |
|---------------|---|---|---|---|
|               | k22 | k21 | k12 | k11 |
|               | 0 | 0 | 0 | 0 |

Global memory

Figure 3. Parallelising spike propagation through a doubly-blocked Toeplitz matrix using one thread block with 4 threads. Snaking lines indicate CUDA threads.

of $A$ with respect to these submatrices is also Toeplitz:

$$A = \begin{pmatrix} A_0 & A_{-1} & \cdots & A_{-(L-1)} \\ A_1 & A_0 & \cdots & A_{-(L-2)} \\ \vdots & \vdots & \ddots & \vdots \\ A_K & A_{K-1} & \cdots & A_0 \end{pmatrix},$$

then, this matrix is called a doubly-blocked Toeplitz matrix. A standard way to generate a Toeplitz matrix from a vector $v$ is to use $v$ as the first column vector, then make one cyclic permutation and use it as the second column vector and so on.

2D convolution operations can be expressed as multiplication by a doubly-blocked Toeplitz matrix. In brief, to convolve an $I_{\text{height}} \times I_{\text{width}}$ input with a $K_{\text{height}} \times K_{\text{width}}$ kernel $K$, e.g.

$$K = \begin{pmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{pmatrix}$$

we flip $K$ across the horizontal and vertical axis and pad it to the output size $(I_{\text{height}} + K_{\text{height}} - 1) \times (I_{\text{width}} + K_{\text{width}} - 1)$ of the convolution. For instance, a $2 \times 3$ input convolved by $K$ above, leads to output size $3 \times 4$. Depending on the padding mode used by the convolution, typically, only part of this output is actually required. The flipped and padded kernel from above would be

$$K_{\text{pad}} = \begin{pmatrix} k_{22} & k_{21} & 0 & 0 \\ k_{12} & k_{11} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

We then convert each row vector of this matrix into Toeplitz matrices $F_i$ as described above:

$$F_0 = \begin{pmatrix} k_{22} & 0 & 0 \\ k_{21} & k_{22} & 0 \\ 0 & k_{21} & k_{22} \\ 0 & 0 & k_{21} \end{pmatrix}, \quad F_1 = \begin{pmatrix} k_{12} & 0 & 0 \\ k_{11} & k_{12} & 0 \\ 0 & k_{11} & k_{12} \\ 0 & 0 & k_{11} \end{pmatrix}, \quad F_2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$
and, finally, assemble these into a doubly blocked Toeplitz matrix $F$:

$$F = \begin{pmatrix}
F_0 & F_2 \\
F_1 & F_0 \\
F_2 & F_1
\end{pmatrix}$$

The convolution of, e.g.,

$$I = \begin{pmatrix}
1 & 0 & 1 \\
0 & 0 & 1
\end{pmatrix}$$

is then given by turning the matrix $I$ into a column vector by stacking up its row vectors,

$$I_{\text{col}} = \begin{pmatrix}
1 \\
0 \\
1 \\
0 \\
1 \\
1
\end{pmatrix}$$

and multiplying $F$ from the left,

$$O_{\text{col}} = F \cdot I \iff O_{\text{col},j} = \sum_i F_{ji} I_i$$

Finally, $O_{\text{col}}$ can be reinterpreted as the output matrix $O$ by arranging its entries row-wise in a $3 \times 4$ matrix.

However, if instead of treating $I$ as a vector of ones and zeros, we represent it as a set of the indices of neurons (with indices starting at 0) we can reframe this operation as follows:

$$I_{\text{ind}} = \{0, 2, 4, 5\}$$

$$O_{\text{col},j} = \sum_{i \in I_{\text{ind}}} F_{ji}$$

This new operation not only removes the need for multiplication entirely but its computational cost will also decrease with sparse spiking. In GeNN, this is implemented in a similar manner to the parallelism strategy for sparse connectivity described by Knight and Nowotny [40] where each block of $N_{\text{block}}$ threads (in figure 3 $N_{\text{block}} = 4$) is responsible for processing $N_{\text{block}}$ kernel entries. Processing begins by using $N_{\text{block}}$ threads to copy the indices $I_{\text{ind}}$ of the $N_{\text{block}}$ presynaptic spikes – which were written to global memory by the presynaptic population’s neuron kernel – into shared memory so that they can be accessed by all threads in the block during the next phase. The threads are then synchronised and loop through the $N_{\text{block}}$ spikes with each thread processing the kernel entries in their column. The index of each thread is then converted into a row, column and output channel within the kernel and the following simple algorithm is used to process a kernel entry,

```python
function processKernelEntry(spikeID, kernRow, kernCol, kernOutChan)
    ▷ Convert incoming spike ID to row, column and channel
    inRow ← spikeID / (I_{chan} * I_{width})
```
$mlGeNN$

\[
\text{inCol} \leftarrow (\text{spikeID} / \text{I}_{\text{chan}}) \% \text{I}_{\text{width}}
\]

\[
\text{inChan} \leftarrow \text{spikeID} \% \text{I}_{\text{chan}}
\]

\> Calculate corresponding output row and column

\[
\text{outRow} \leftarrow \text{inRow} + \text{kernRow}
\]

\[
\text{outCol} \leftarrow \text{inCol} + \text{kernCol}
\]

\> Convert output row, column and channel back into neuron ID

\[
\text{neuronID} \leftarrow ((\text{outRow} \times \text{O}_{\text{width}} \times \text{O}_{\text{chan}}) + (\text{outCol} \times \text{O}_{\text{chan}}) + \text{kernOutChan})
\]

\[
\text{GENSYNAPSE}(\text{neuronID}, \text{kernRow}, \text{kernCol}, \text{inChan}, \text{kernOutChan})
\]

end if

end function

where GENSYNAPSE plays the same role as it did in the algorithm for procedural connectivity described in section 2.4.1.

2.5. Batching

In order to fully occupy highly-parallel GPU devices, machine learning tools typically simultaneously apply batches of data to multiple copies of models. This reduces the detrimental effects of kernel launch latency on overall compute time. Furthermore, weights and other variables can be shared across all instances within a batch, improving the locality of memory accesses.

Because of the regular memory access patterns of the matrix multiplication operations used to propagate activity between ANN layers, the kernels employed by modern ML libraries are optimised to minimise global memory access across the batch. However, when propagating spikes, different neurons will be spiking across the instances of the batch so different weights will be accessed. This makes a software-based approach for reducing global memory accesses to weights difficult so, in GeNN, we implement batching by simply duplicating our neuron and spike propagation kernels across another thread block dimension and rely on the GPU caches to reduce global memory traffic to the shared weights.

2.6. Rate-based conversion

Rate-based conversion of ANNs to SNNs aims to emulate the transfer function of the ANN unit with that of a spiking model. Typically the transfer function of rectified linear units (ReLU) with no bias tensors is emulated using simple integrate and fire (IF) neurons [22, 24]:

\[
V_j(t + 1) = V_j(t) + \sum_{i=0}^{N} w_{ji}z_i(t)
\]

where $V_j$ represents the membrane voltage of neuron $j$, into which the input spikes $z_i$ from $N$ presynaptic neurons are accumulated via a weight matrix $w_{ji}$. When $V_j$ crosses
a threshold $V_{\text{thr}}$, a spike is emitted and $V_j$ is reset to zero. The weights of each layer in
the ANN model are then transplanted into the corresponding layers of the SNN model.

In order to maximise the classification accuracy of the converted SNN model, it is important to ensure that the available dynamic range of its spiking activity is representative of the range of activations occurring in the ANN model. If the integration step is too coarse, or the input current to the integrate and fire (IF) neurons is too strong, then it is possible for the membrane potential to far exceed the firing threshold value before being reset in the next timestep. The difference $V - V_{\text{thr}}$ of membrane potential $V$ and threshold $V_{\text{thr}}$ in such a situation can be interpreted as the amount of lost information and needs to be minimised. Conversely, if the stimulation of a layer is unexpectedly weak, it can take a long time before neurons are stimulated enough to fire, and neurons that do not fire within the sample presentation time again represent lost information. This necessitates increasing the amount of sample presentation time for accurate classification. The solution to this issue is threshold balancing, where the threshold of IF neurons in each layer are set to their maximum expected activation but not higher, such that the amount each IF neuron’s $V$ exceeds its threshold at firing is minimised throughout the simulation. An equivalent method of dividing the layers’ incoming weights by the maximum activation may also be used. Various methods have been proposed for computing the expected activation of the IF neurons. data-norm [22] and spike-norm [24] are two such threshold balancing methods.

Here we use the data-norm method, where a random subset of real training data is propagated through the ANN, and the activation of each unit in each layer of the ANN is measured. While the maximum activation of each layer alone can be used as the corresponding SNN layer’s threshold, Diehl et al. [22] further propose using the highest of either the maximum activation or maximum incoming weight of that layer, to prevent any single incoming spike from instantly raising the neuron’s membrane potential above threshold (hence losing information). The following algorithm demonstrates this method for a feedforward ANN model.

```plaintext
function DataNorm(ann, snn, data)

▷ Set SNN thresholds with the data-norm method
previousFactor ← 1
for $i ← 1$ to ann.nLayers do
  maxActivation ← 0
  for all $x ∈$ data do
    maxActivation ← MAX(ann.layers[i].forward($x$))
  end for
  maxWeight ← MAX(ann.layers[i].weights)
  scaleFactor ← MAX(maxActivation, maxWeight)
  snn.layers[i].threshold ← scaleFactor/previousFactor
  previousFactor ← scaleFactor
end for
end function
```
mlGeNN

2.7. Few-spike conversion

While the conversion of ReLU-based ANNs to rate-based SNNs can produce high classification accuracy [23, 24], in order to accurately represent analogue activations, a large number of spikes are required. Therefore, either large numbers of timesteps need to be simulated for each stimulus (2500 is typical [24]) or very high rates are required. Either way, the potential computation and energy efficiency gains of using spikes is likely to be lost.

Thorpe et al. [55] analysed a number of different spike coding schemes and found that coding schemes which consider the timing of individual spikes – rather than just their rate – can increase the amount of information which can be transmitted in a fixed time by up to 10×. Therefore, it is unsurprising that several techniques have been proposed for converting ANNs to SNNs with a spike-timing based encoding scheme [25, 26]. Here we use the FewSpike approach [26] which aims to encode each stimulus using an average of two spikes per neuron and many fewer timesteps (typically ≈ 10) and has already been shown capable of converting large-scale ANNs to SNNs. In GeNN, we perform inference using these neurons in a ‘pipelined’ manner so that, during one K timestep stimulus presentation, each neuron j emits spikes representing the previous stimulus and accumulates spikes representing the current stimulus. To do this, each neuron has one state variable ̂Fj into which spikes z are accumulated from N input neurons i via a weight matrix wji:

\[ ̂F_j(t) = d(t) \sum_{i=0}^{N} w_{ji} z_i(t) \]

where d(t) is a function common to all neurons with the same activation function. At the start of each stimulus presentation, the neuron’s second state variable Vj is set to the value of ̂Fj that was accumulated during the previous K timesteps and ̂Fj is reset to zero. Vj then evolves for the next K timesteps as follows:

\[ V_j(t + 1) = V_j(t) - h(t)z_j(t) \]
\[ z_j(t) = H(V_j(t) - T(t)) \]

where H represents the Heaviside step function. T(t) and h(t) are respectively the spiking threshold and the amount subtracted from the membrane potential when the neuron produces a spike, and are common to all neurons with the same activation function. In this paper we use ReLU neurons where d(t) = h(t) = T(t) = α2(K−t) which essentially encode activations in a binary K bit fixed-point format. In a similar manner to the rate-based normalisation, we optimise the encoding of activations by calculating a per-layer scaling factor α based on the maximum activation value encountered in a subset of training data. In order to support ResNet models with multiple pathways such as those described in section 2.3, a synaptic delay of K + 1 timesteps is added to the identity pathway to match the additional pipeline stage and extra timestep of delay in the non-identity pathway. These ensure that all inputs to the neurons at the junction points are correctly synchronised.
| Network       | ANN accuracy [%] | Best rate-based SNN accuracy [%] | Best few-spike SNN accuracy [%] |
|--------------|------------------|----------------------------------|-------------------------------|
| VGG-16       | 91.2             | 90.9                             | 91.4                          |
| VGG-16 (Sengupta et al. [24]) | 91.7             | 91.6                             |                               |
| ResNet-20    | 89.4             | 84.3                             | 89.4                          |
| ResNet-20 (Sengupta et al. [24]) | 89.1             | 87.5                             | –                             |
| ResNet-20 (Stöckl and Maass [26]) | 91.6             | –                                | 91.5                          |

Table 1. Comparison of CIFAR-10 accuracy between our best models and those trained by Sengupta et al. [24] and; Stöckl and Maass [26]. Bold rows indicate our own results.

2.8. mlGeNN

mlGeNN is our new open source Python package which uses the PyGeNN library [38] to accelerate the simulation of SNN models using graphics processing units (GPUs).

It is used by first constructing and training an ANN model in Keras, keeping the constraints discussed in sections 2.6 and 2.7 in mind. mlGeNN currently supports dense, 2D convolution and 2D average pooling layers, but could be extended with other layer types, such as 3D convolution and recurrent layers. mlGeNN also supports models with multiple input and output layers and, models constructed using both the Keras sequential and functional APIs. The fully trained ANN model is then automatically translated by mlGeNN into a description suitable for GeNN. Connectivity (dense, convolution) layers and subsequent activation layers are translated into synapse groups and neuron groups using the models required by the conversion scheme. Pooling layers are treated a little differently since they are not typically followed by activation functions. Inserting additional neuron groups here would introduce additional nonlinearities in the SNN model which would degrade classification accuracy so, rather than creating new neuron groups, pooling operations are instead merged with the connectivity immediately downstream of them. Furthermore, there are constraints on the type of pooling layer one can use. While several implementations of spiking max pooling layers have been proposed [23, 56], these are all tied to the coding scheme used by the network so, here, we prefer using average pooling layers.

The model conversion is performed through an instance of a model converter, corresponding to the chosen conversion method (see above). mlGeNN provides the DataNorm [22], SpikeNorm [24] and FewSpike [26] converter classes. When constructing each converter, one passes the signed_input argument, indicating whether the input data is signed or all positive, and the norm_data argument, which should be a list containing a subset of training data (one entry for each model input) used to optimise the model conversion. Signed input is implemented by using GeNN’s normal spiking mechanism for positive inputs and GeNN’s spike-like-event mechanism (see Knight et al. [38]) to emit ‘negative spikes’, i.e. spike events that trigger updates by $-w$ for synapses of
Finally, one calls the `Model.convert_tf_model` method of the mlGeNN `Model` class, passing the TensorFlow model and converter instance, and optionally passing the synapse connectivity type ("sparse", "procedural", "toeplitz" as described in section 2.4), simulation timestep, batch size and random number generator seed (passing 0 yields a random seed). This method returns an mlGeNN SNN model corresponding to the given ANN model. This call also instructs the GeNN backend to generate, compile and load the GPU code of the model. Finally, the user evaluates the mlGeNN model by calling its `evaluate` method, passing a list of input data lists (one entry per input layer), a list of output label lists (one entry per output layer), and the presentation time of each stimulus in milliseconds. An example of this process, using the rate-based data-norm conversion method, is given below.

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import *
tf_model = Sequential([
    Conv2D(16, 3, padding='same', activation='relu',
           use_bias=False, input_shape=(32, 32, 3)),
    AveragePooling2D(2),
    Flatten(),
    Dense(10, activation='relu', use_bias=False)])

# compile and train tf_model with TensorFlow
...

from ml_genn import Model
from ml_genn.converters import DataNorm
from ml_genn.layers import InputType
converter = DataNorm([x_subset], signed_input=True,
                      input_type=InputType.POISSON)
tg_model = Model.convert_tf_model(tf_model, converter=converter,
                                  connectivity_type='toeplitz',
                                  dt=1.0, batch_size=50,
                                  rng_seed=0)
tg_model.evaluate([x], [y], 300)
```

3. Results

In this section we explore the effect on accuracy and inference latency of SNNs converted using the algorithms described in sections 2.6 and 2.7. We also compare the latency of SNNs to both the original TensorFlow ANNs and, for the VGG-16 model, to SNNs simulated using BindsNet [31] using the modifications developed by Lu and Sengupta [57] to support average pooling.
3.1. CIFAR-10

Table 1 compares the accuracy on the CIFAR-10 dataset of the VGG-16 and ResNet-20 models described in section 2.3 to prior work by Sengupta et al. [24] as well as Stöckl and Maass [26]. As both models are based on the descriptions provided by Sengupta et al. [24], unsurprisingly, both ANN models reach similar levels of accuracy. While mlGeNN does support the ‘spike-norm’ rate-based conversion algorithm used by Sengupta et al. [24], here we use the simpler data-norm conversion algorithm of Diehl et al. [22], modified to support ResNet architectures. This results in slightly lower rate-based conversion accuracies than those reported by Sengupta et al. [24] but, the few-spike models perform much better. To support rate-based conversion, we do not use batch normalization and bias when training our models. However, Stöckl and Maass [26] were not constrained by these restrictions so were able to train a rather more accurate ResNet-20 model and this improved performance was transferred to the SNN model.

Both of the conversion algorithms described in sections 2.6 and 2.7 allow speed of inference to be ‘traded off’ for accuracy by varying the number of simulation timesteps each stimulus is presented for. When using rate-based conversion, more timesteps \((T)\) allows for a rate code with a finer granularity and, when using few-spike conversion, additional timesteps \((K)\) add extra ‘bits’ to the binary fixed-point format and thus reduce the quantisation error. Figure 4 shows the accuracy of VGG-16 and ResNet-20 models trained on the CIFAR-10 dataset and converted to rate-based and few-spike SNNs using different \(T\) and \(K\) values. When using the VGG-16 model, classification accuracy is significantly degraded with values of \(T < 2500\) or \(K < 10\) (Figure 4a) whereas, when using the ResNet-20 model classification performance is only significantly degraded with values of \(T < 1000\) or \(K < 8\) (figure 4b). This reduced ‘convergence time’ for the ResNet-20 model matches the findings presented by Sengupta et al. [24].
Using these optimal $T$ and $K$ values, we simulated both models using the different connection algorithms described in section 2.4 across a range of batch sizes. Figure 5 shows the resultant time to perform inference on the entire CIFAR-10 test set using the VGG-16 and ResNet-20 models. At batch size 1, the procedural connectivity approach described in section 2.4.1 performs best across all conversion algorithms and models – with the ResNet-20 SNN actually performing more than $3.5 \times$ faster than the TensorFlow ANN. However, as batch size increases, the Toeplitz matrix algorithm described in section 2.4.2 performs the best with the fastest FewSpike SNNs being only a little over $2 \times$ slower than the TensorFlow ANN when using the ResNet-20 model and around $12.5 \times$ slower when using the VGG-16 model. The procedural connectivity algorithm scales poorly because, while at small batch sizes, using one thread per-neuron (rather than per-matrix diagonal) allows the GPU’s parallelism to be better exploited, as batch sizes increase, memory latency worsens due to high contention between the atomic operations used to update the input to the postsynaptic neuron and badly coalesced reading of weights.

Because the models converted using the rate-based algorithm need to be simulated for many more timesteps than those converted using the few-spike algorithm, they are slower across the board – with even the fastest model being over $100 \times$ slower than the TensorFlow ANN. However, the encoding scheme used by the few-spike model [26] actually results in approximately $10 \times$ more spikes per timestep than the rate-based
### Table 2. Comparison of ImageNet top-1 accuracy between our best models and those trained by Sengupta et al. [24].

| Network          | ANN accuracy [%] | Best rate-based SNN accuracy [%] | Best few-spike SNN accuracy [%] |
|------------------|------------------|----------------------------------|---------------------------------|
| VGG-16           | 70.3             | -                                | 70.2                            |
| VGG-16 (Sengupta et al. [24]) | 70.5             | 69.96                            |                                 |
| ResNet-34        | 69.3             | -                                | 68.4                            |
| ResNet-34 (Sengupta et al. [24]) | 70.7             | 65.47                            |                                 |

Figure 6. ImageNet inference time using VGG-16 (A) and ResNet-34 (B) models. SNNs were simulated using optimal configuration of \( K = 10 \) for few-spike conversion. The VGG-16 SNNs were simulated with a batch size of 32 when using procedural connectivity and 48 when using Toeplitz connectivity. The ResNet-34 SNNs were all simulated with a batch size of 32. ANNs were simulated with optimal batch size of 256. A 12 GB Titan V GPU is used for all experiments.

model. Therefore, the spiking activity in each timestep is much sparser in the rate-based models, allowing GeNN to simulate individual timesteps significantly faster than an ANN step. Note that the total number of spikes per evaluation with few-spike conversion is still much lower than the total spikes per evaluation with rate-based conversion, since fewer timesteps are required per evaluation.

Finally, comparing the VGG-16 model simulated using mlGeNN against BindsNET [31] we can see that because, under the hood, BindsNET uses the same dense tensor operations as the ANN, as discussed in the introduction, its performance is very similar to that of the ANN multiplied by \( T \) and over \( 2.5 \times \) slower than the fastest rate-based GeNN VGG-16 model.
Table 2 compares the accuracy on the ImageNet dataset of the VGG-16 and ResNet-34 models described in section 2.3 to the results presented by Sengupta et al. [24]. Again, these models are based on the descriptions provided by Sengupta et al. [24] so the ANN models reach similar levels of accuracy. Based on our CIFAR-10 results, presented in the previous section, we have not evaluated our rate-based conversion algorithm on ImageNet but both of our few-spike converted models out-perform the rate-based performances reported by Sengupta et al. [24].

Figure 6 shows the time taken to perform inference on the entire ImageNet test set using the VGG-16 and ResNet-34 models. The simulation with the Toeplitz connectivity is approximately twice as fast as the procedural algorithm in both models but still over $100 \times$ slower than the ANN simulated using TensorFlow. While at first maybe a bit surprising, this result can be understood in the context of how TensorFlow processes convolutions. As discussed in section 2.4, TensorFlow switches between multiplying by Toeplitz matrices or employing Winograd or FFT based approaches based on the geometry of each convolutional layer. Lavin et al. [50] derived the following expressions for the total number of arithmetic operations required to convolve an $I_{\text{width}} \times I_{\text{height}} \times I_{\text{chan}}$ input with a $K_{\text{height}} \times K_{\text{width}} \times I_{\text{chan}} \times O_{\text{chan}}$ kernel using these approaches:

$$L = \alpha \left(1 + \frac{\beta}{I_{\text{chan}}} + \frac{\gamma m^2}{N_{\text{batch}} I_{\text{width}} I_{\text{height}}} + \frac{\delta}{O_{\text{chan}}}ight) N_{\text{batch}} I_{\text{width}} I_{\text{height}} I_{\text{chan}} O_{\text{chan}}$$

where $N_{\text{batch}}$ is the batch size, $m$ is a function of the size of the tiles the input is divided into when using the FFT or Winograd algorithms and the factors $\alpha$, $\beta$ and $\gamma$ relate to the operations incurred within different parts of the three algorithms. For straightforward matrix multiplication-based approaches, $\alpha = K_{\text{width}} K_{\text{height}}$ and $\beta = \gamma = \delta = 0$, but, for Winograd and FFT approaches, $\alpha$, $\beta$ and $\gamma$ are constants with values between 1 and 10, depending on the exact configurations. In models simulated with a large batch size and with large $I_{\text{width}}$, $I_{\text{height}}$, $I_{\text{chan}}$ and $O_{\text{chan}}$ such as the VGG and ResNet models operating on ImageNet considered here, the $\beta$, $\gamma$ and $\delta$ terms are relatively small and the number of operations required for the Winograd and FFT based approaches are essentially only

$$L \approx \alpha N_{\text{batch}} I_{\text{width}} I_{\text{height}} I_{\text{chan}} O_{\text{chan}}$$

By exploiting sparsity, GeNN reduces the number of operations ($L$) compared to straightforward matrix multiplication by the fraction $P_{\text{spike}}$ of neurons which spike each timestep. Additionally, as propagating spikes requires no multiplications, there should be some additional advantage. However, if $\alpha < P_{\text{spike}} K_{\text{width}} K_{\text{height}}$ for the Winograd and FFT based approaches, an ANN step with one of these algorithms will be more efficient than a single timestep of GeNN’s current spike propagation algorithms. Accordingly, as $\alpha = 2.25$ for the Winograd and 1.81 for the FFT configurations used by TensorFlow in the models considered, and $K_{\text{width}} = K_{\text{height}} = 3$, a sparsity of $P_{\text{spike}} < 0.2$ would be required for an SNN timestep to be faster than the ANN step. For both of the model implementations reported here, $P_{\text{spike}} \approx 0.4$ and if we also consider that we
are presenting each stimulus for $K = 10$ timesteps, TensorFlow has approximately a $16 \times$ algorithmic advantage in the number of arithmetic operations. Furthermore, the number of arithmetic operations is only half the story. cuDNN is a highly-optimised library, where key components are written directly in SASS assembly language for each GPU architecture whereas GeNN is based on CUDA and there remains significant scope for further optimisation, specifically to remove the need for atomic operations in global memory and to further optimise index calculations.

4. Discussion

In this paper we have introduce mlGeNN, a library for machine learning with spiking neural networks that makes use of the GeNN GPU accelerated SNN simulator. We have described how mlGeNN can be used to translate trained ANN architectures into different SNN implementations, using standard algorithms [22, 24, 26], and how the speed and accuracy of inference compares to both the original ANN models and SNN models simulated in other libraries. We found that accuracy is as described in the original publications reporting on ANN to SNN conversions while speed of inference presents a very mixed picture. While mlGeNN is faster than BindsNET [31] on most of the considered examples, and comparable to ANN speeds for some, it is orders of magnitudes slower for others.

As discussed in section 3.2, the gap in performance between ANN and SNN models, running on the same hardware comes from the number of timesteps SNN models have to run per-stimuli, the algorithmic advantage of specialised ANN convolution algorithms and the degree of optimisation and maturity of cuDNN [51] when compared to SNN libraries such as GeNN [39].

In this work – aside from the restrictions described in section 2.6 and 2.7 – we have only considered the conversion of models using standard 32 bit floating point weights and activations. However, representing ANN activations in as few SNN timesteps as possible – using either a rate code or the fixed-point spike code used by the few-spike method – is a similar problem to quantizing ANN activations for inference with low-precision data types. Therefore, applying some techniques from quantization-aware training [58] or fine-tuning [5] to our ANN activations could allow the number of SNN timesteps per stimulus to be minimised, reducing the gap between ANN and SNN performance.

While FewSpike representations require fewer SNN timesteps per stimulus, in the ImageNet-scale models we evaluate in section 3.2, they result in very high firing rates which make each timestep slower than ANN updates. One solution would be to ‘sparsen’ the neuron activations by training the ANN with L1 regularisation so more activations are zero which is represented by the absence of spikes. Alternatively, binary codes where more common activation values are represented with fewer bits could be employed although this would add additional decoding complexity to the neurons or synapses of the model.

The new ‘Toeplitz’ connectivity scheme we describe in section 2.4.2 is significantly
faster than both the general purpose sparse connectivity described by Knight and Nowotny [40] and the general purpose ‘procedural’ connectivity described by Knight et al. [38] but, significant levels of sparsity are still required for it to match the number of operations required for an ANN update. This property not only affects software-based SNN simulators like GeNN, but is also an important consideration for designers of spike-based neuromorphic chips aiming to compete with non-spiking ML accelerators. However, beyond the theoretical differences in the number of operations required to update an ANN which we previously discussed, there is significant scope to further optimise our implementation. Our current implementation accumulates the output from each layer into global memory using ‘fire-and-forget’ atomic add operation. Although these operations are handled entirely within the L2 cache of modern GPUs and thus have relatively low latency, by accumulating updates into shared memory using the approach described by Bautembach et al. [53], this could be further reduced. Furthermore, profiling using the NVIDIA Nsight compute profiler suggests that while the algorithm is currently primarily compute bound, a large proportion of the arithmetic operations involved index calculations rather than actually processing of convolutions. This is because our single program multiple data code generator (described in Knight et al. [38]) will reuse the same generated code to simulate all layers with Toeplitz connectivity, meaning that the CUDA compiler will be unable to optimize away the divide operations within the index calculations required by the algorithm presented in section 2.4.2. Current GPU architectures do not have hardware support for integer division so this results in a large number of superfluous operations to implement these in software but, as the divisors are all known at build-time, these could be replaced with a single integer multiply-add operation [59].

In conclusion, the mlGeNN framework adds a new element to the GeNN ecosystem, allowing GeNN to be used for machine learning workflows that translate fully trained deep ANNs into SNNs for inference. This is a first step towards a fully SNN-based machine ecosystem built on GeNN that, in the future, will also allow direct training of SNNs. Ultimately, we hope that this will become the ideal tool for developing SNN based algorithms for deployment on both standard and neuromorphic hardware.

Funding

This work was funded by the EPSRC (Brains on Board project, grant number EP/P006094/1 and ActiveAI project, grant number EP/S030964/1), the European Union’s Horizon 2020 research and innovation program under Grant Agreement 945539 (HBP SGA3), by Huawei Technologies Co and a Google Summer of Code grant to AS. Additionally we gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS Supercomputer JUWELS at Jülich Supercomputing Centre (JSC) and the JADE2 consortium funded by the EPSRC (EP/T022205/1) for compute time on their systems.
REFERENCES

Acknowledgments

We would like to thank Malin Sandström and everyone else at the International Neuroinformatics Coordinating Facility (INCF) for their hard work running the Google Summer of Code mentoring organisation every year. Without them, this and many other exciting projects would not be possible.

References

[1] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition, pages 248–255. Ieee, 2009.

[2] Muhammad Abdullah Hanif, Muhammad Usama Javed, Rehan Hafiz, Semeen Rehman, and Muhammad Shafique. Hardware–Software Approximations for Deep Neural Networks, pages 269–288. Springer International Publishing, Cham, 2019. ISBN 978-3-319-99322-5. doi: 10.1007/978-3-319-99322-5_13. URL https://doi.org/10.1007/978-3-319-99322-5_13.

[3] Eva García-Martín, Crefeda Faviola Rodrigues, Graham Riley, and Håkan Grahn. Estimation of energy consumption in machine learning. Journal of Parallel and Distributed Computing, 134:75–88, 2019. ISSN 07437315. doi: 10.1016/j.jpdc.2019.07.007. URL https://doi.org/10.1016/j.jpdc.2019.07.007.

[4] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. apr 2017. URL http://arxiv.org/abs/1704.04861.

[5] Jeffrey L. McKinstry, Steven K. Esser, Rathinakumar Appuswamy, Deepika Bablani, John V. Arthur, Izzet B. Yildiz, and Dharmendra S. Modha. Discovering Low-Precision Networks Close to Full-Precision Networks for Efficient Embedded Inference. pages 1–11, 2018. URL http://arxiv.org/abs/1809.04191.

[6] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Canin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Guillard, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham,
REFERENCES

Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture, volume Part F1286, pages 1–12, New York, NY, USA, jun 2017. ACM. ISBN 9781450348928. doi: 10.1145/3079856.3080246. URL https://dl.acm.org/doi/10.1145/3079856.3080246.

[7] Ning Qiao, Hesham Mostafa, Federico Corradi, Marc Osswald, Fabio Stefanini, Dora Sumislawsk, and Giacomo Indiveri. A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses. Frontiers in Neuroscience, 9(APR):1–17, 2015. ISSN 1662453X. doi: 10.3389/fnins.2015.00141.

[8] Johannes Schemmel, Laura Kriener, Paul Muller, and Karlheinz Meier. An accelerated analog neuromorphic hardware system emulating NMDA- and calcium-based non-linear dendrites. Proceedings of the International Joint Conference on Neural Networks, 2017-May:2217–2226, 2017. doi: 10.1109/IJCNN.2017.7966124.

[9] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D Flickner, William P Risk, Rajit Manohar, and Dharmendra S Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. Science, 345(6197):668–673, 2014. doi: 10.1126/science.1254642. URL http://www.sciencemag.org/content/345/6197/668.abstract.

[10] Mike Davies, Narayan Srinivasa, Tsung-han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutt, Steve Mccoy, Arnab Paul, Jonathan Tse, Guruguhanthan Venkataramanan, Yi-hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang. Loihi : a Neuromorphic Manycore Processor with On-Chip Learning. IEEE Micro, 30(1):82–99, 2018. doi: 10.1109/MM.2018.112130359.

[11] Charlotte Frenkel, Jean-didier Legat, and David Bol. A 65-nm 738k-Synapse/mm 2 Quad-Core Binary-Weight Digital Neuromorphic Processor with Stochastic Spike-Driven Online Learning. In 2019 IEEE International Symposium on Circuits and Systems (ISCAS), pages 1–5. IEEE, may 2019. ISBN 978-1-7281-0397-6. doi: 10.1109/ISCAS.2019.8702793. URL https://ieeexplore.ieee.org/document/8702793/.

[12] Stephen B Furber, Francesco Galluppi, S Temple, and Luis A Plana. The SpiNNaker Project. Proceedings of the IEEE, 102(5):652–665, may 2014. ISSN 0018-9219. doi: 10.1109/JPROC.2014.2304638.

[13] Sebastian Höppner, Yexin Yan, Andreas Dixius, Stefan Scholze, Johannes Partzsch, Marco Stolba, Florian Kelber, Bernhard Vogginger, Felix Neumärker, Georg
REFERENCES

Ellguth, Stephan Hartmann, Stefan Schiefer, Thomas Hocker, Dennis Walter, Genting Liu, Jim Garside, Steve Furber, and Christian Mayr. The SpiNNaker 2 Processing Element Architecture for Hybrid Digital Neuromorphic Computing. pages 1–9, 2021. URL http://arxiv.org/abs/2103.08392.

[14] Sander M. Bohte, Joost N. Kok, and Han La Poutré. Error-backpropagation in temporally encoded networks of spiking neurons. Neurocomputing, 48(1):17–37, 2002. ISSN 0925-2312. doi: https://doi.org/10.1016/S0925-2312(01)00658-0. URL https://www.sciencedirect.com/science/article/pii/S0925231201006580.

[15] Iulia-Maria Comșa, Krzysztof Potempa, Luca Versari, Thomas Fischbacher, Andrea Gesmundo, and Jyrki Alakuijala. Temporal coding in spiking neural networks with alpha synaptic function: Learning with backpropagation. IEEE Transactions on Neural Networks and Learning Systems, pages 1–14, 2021. doi: 10.1109/TNNLS.2021.3071976.

[16] Timo C Wunderlich and Christian Pehle. Event-based backpropagation can compute exact gradients for spiking neural networks. Scientific Reports, 11(1):12829, dec 2021. ISSN 2045-2322. doi: 10.1038/s41598-021-91786-z. URL https://doi.org/10.1038/s41598-021-91786-z http://www.nature.com/articles/s41598-021-91786-z.

[17] Guillaume Bellec, Darjan Salaj, Anand Subramoney, Robert Legenstein, and Wolfgang Maass. Long short-term memory and learning-to-learn in networks of spiking neurons. In Advances in Neural Information Processing Systems, volume 2018-Decem, pages 787–797, 2018.

[18] Friedemann Zenke and Tim P. Vogels. The Remarkable Robustness of Surrogate Gradient Learning for Instilling Complex Function in Spiking Neural Networks. Neural Computation, 33(1):899–925, 2021. ISSN 1530888X. doi: 10.1162/neco_a_01367.

[19] Guillaume Bellec, Franz Scherr, Anand Subramoney, Elias Hajek, Darjan Salaj, Robert Legenstein, and Wolfgang Maass. A solution to the learning dilemma for recurrent networks of spiking neurons. Nature Communications, 11(1):3625, dec 2020. ISSN 2041-1723. doi: 10.1038/s41467-020-17236-y. URL http://www.nature.com/articles/s41467-020-17236-y.

[20] Jacques Kaiser, Hesham Mostafa, and Emre Neftci. Synaptic Plasticity Dynamics for Deep Continuous Local Learning (DECOLLE). Frontiers in Neuroscience, 14 (May):1–11, may 2020. ISSN 1662-453X. doi: 10.3389/fnins.2020.00424. URL https://www.frontiersin.org/article/10.3389/fnins.2020.00424/full.

[21] Friedemann Zenke and Surya Ganguli. Superspike: Supervised learning in multilayer spiking neural networks. Neural Computation, 30(6):1514–1541, 06 2018. ISSN 0899-7667. doi: 10.1162/neco_a_01086. URL https://doi.org/10.1162/neco_a_01086.

[22] Peter U. Diehl, Daniel Neil, Jonathan Binas, Matthew Cook, Shih-Chii Liu, and Michael Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through
REFERENCES

weight and threshold balancing. In 2015 International Joint Conference on Neural Networks (IJCNN), volume 2015-Septe, pages 1–8. IEEE, jul 2015. ISBN 978-1-4799-1960-4. doi: 10.1109/IJCNN.2015.7280696. URL http://ieeexplore.ieee.org/document/7280696/.

[23] Bodo Rueckauer, Iulia Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih Chii Liu. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. Frontiers in Neuroscience, 11(DEC):1–12, 2017. ISSN 1662453X. doi: 10.3389/fnins.2017.00682.

[24] Abhronil Sengupta, Yuting Ye, Robert Wang, Chiao Liu, and Kaushik Roy. Going Deeper in Spiking Neural Networks: VGG and Residual Architectures. Frontiers in Neuroscience, 13(1998):1–16, mar 2019. ISSN 1662-453X. doi: 10.3389/fnins.2019.00095. URL https://www.frontiersin.org/article/10.3389/fnins.2019.00095/full.

[25] Bodo Rueckauer and Shih Chii Liu. Conversion of analog to spiking neural networks using sparse temporal coding. Proceedings - IEEE International Symposium on Circuits and Systems, 2018-May:1–5, 2018. ISSN 02714310. doi: 10.1109/ISCAS.2018.8351295.

[26] Christoph Stöckl and Wolfgang Maass. Optimized spiking neurons can classify images with high accuracy through temporal coding with two spikes. Nature Machine Intelligence, 3(3):230–238, 2021.

[27] Nicholas T Carnevale and Michael L Hines. The NEURON book. Cambridge University Press, 2006.

[28] Marc-Oliver Gewaltig and Markus Diesmann. NEST (NEural Simulation Tool). Scholarpedia, 2(4):1430, 2007.

[29] Nora Abi Akar, Ben Cumming, Vasileios Karakasis, Anne Kusters, Wouter Klijn, Alexander Peyser, and Stuart Yates. Arbor — A Morphologically-Detailed Neural Network Simulation Library for Contemporary High-Performance Computing Architectures. In 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pages 274–282. IEEE, feb 2019. ISBN 978-1-7281-1644-0. doi: 10.1109/EMPDP.2019.8671560. URL http://arxiv.org/abs/1901.07454https://ieeexplore.ieee.org/document/8671560/.

[30] Marcel Stimberg, Romain Brette, and Dan F.M. Goodman. Brian 2, an intuitive and efficient neural simulator. eLife, 8:1–41, 2019. ISSN 2050084X. doi: 10.7554/eLife.47314.

[31] Hananel Hazan, Daniel J. Saunders, Hassaan Khan, Devdhar Patel, Darpan T. Sanghavi, Hava T. Siegelmann, and Robert Kozma. BindsNET: A Machine Learning-Oriented Spiking Neural Networks Library in Python. Frontiers in Neuroinformatics, 12(December):1–18, dec 2018. ISSN 1662-5196. doi: 10.3389/fninf.2018.00089. URL http://arxiv.org/abs/1806.01423https://www.frontiersin.org/article/10.3389/fninf.2018.00089/full.
REFERENCES

[32] Christian Pehle and Jens Egholm Pedersen. Norse - A deep learning library for spiking neural networks, January 2021. URL https://doi.org/10.5281/zenodo.4422025. Documentation: https://norse.ai/docs/.

[33] Dylan Muir, Felix Bauer, and Philipp Weidel. Rockpool documentation, September 2019. URL https://doi.org/10.5281/zenodo.4639684.

[34] Milad Mozafari, Mohammad Ganjtabesh, Abbas Nowzari-Dalini, and Timothée Masquelier. SpykeTorch: Efficient Simulation of Convolutional Spiking Neural Networks With at Most One Spike per Neuron. Frontiers in Neuroscience, 13(Mm):1–16, jul 2019. ISSN 1662-453X. doi: 10.3389/fnins.2019.00625. URL https://www.frontiersin.org/article/10.3389/fnins.2019.00625/full.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, Advances in Neural Information Processing Systems 32, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[36] TensorFlow Developers. Tensorflow, November 2021. URL https://doi.org/10.5281/zenodo.5645375. Specific TensorFlow versions can be found in the "Versions" list on the right side of this page.<br>See the full list of authors <a href="https://github.com/tensorflow/tensorflow/graphs/contributors">on GitHub</a>.

[37] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

[38] James C Knight, Anton Komissarov, and Thomas Nowotny. PyGeNN: A Python Library for GPU-Enhanced Neural Networks. Frontiers in Neuroinformatics, 15(April), apr 2021. ISSN 1662-5196. doi: 10.3389/fninf.2021.659005. URL https://www.frontiersin.org/articles/10.3389/fninf.2021.659005/full.

[39] Esin Yavuz, James Turner, and Thomas Nowotny. GeNN: a code generation framework for accelerated brain simulations. Scientific reports, 6(November 2015):18854, 2016. ISSN 2045-2322. doi: 10.1038/srep18854. URL http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=4703976&tool=pmcentrez&rendertype=abstract.

[40] James C. Knight and Thomas Nowotny. GPUs Outperform Current HPC and Neuromorphic Solutions in Terms of Speed and Energy When Simulating a
REFERENCES

Highly-Connected Cortical Model. Frontiers in Neuroscience, 12(December):1-19, 2018. ISSN 1662-453X. doi: 10.3389/fnins.2018.00941. URL https://www.frontiersin.org/article/10.3389/fnins.2018.00941/full.

[41] James C Knight and Thomas Nowotny. Larger GPU-accelerated brain simulations with procedural connectivity. Nature Computational Science, 1(2):136–142, feb 2021. ISSN 2662-8457. doi: 10.1038/s43588-020-00022-7. URL https://www.biorxiv.org/content/10.1101/2020.04.27.063693v2http://www.nature.com/articles/s43588-020-00022-7.

[42] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. PhD thesis, University of Toronto, 2009.

[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, NIPS’12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[44] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[46] Alex Krizhevsky. cudaconvnet2, 2014. URL https://code.google.com/p/cuda-convnet2/.

[47] NVIDIA Corporation. cuBLAS, 2007. URL https://developer.nvidia.com/cublas.

[48] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. Fast implementation of DGEMM on Fermi GPU. Proceedings of 2011 SC - International Conference for High Performance Computing, Networking, Storage and Analysis, 2011. doi: 10.1145/2063384.2063431.

[49] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through FFTS. 2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings, pages 1–9, 2014.

[50] Andrew Lavin and Scott Gray. Fast Algorithms for Convolutional Neural Networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 4013–4021, Las Vegas, 2016.

[51] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. pages 1–9, 2014. URL http://arxiv.org/abs/1410.0759.

[52] Dennis Bautembach, Jason Oikonomidis, Nikolaos Kyriazis, and Antonis Argyros. Faster and Simpler SNN Simulation with Work Queues. In 2020 International Joint Conference on Neural Networks (IJCNN), pages 1–8. IEEE, jul 2020. ISBN 978-1-7281-6926-2. doi: 10.1109/IJCNN48605.2020.9206752. URL https://ieeexplore.ieee.org/document/9206752/.
REFERENCES

[53] Dennis Bautembach, Iason Oikonomidis, and Antonis Argyros. Even Faster SNN Simulation with Lazy+Event-driven Plasticity and Shared Atomics. jul 2021. URL http://arxiv.org/abs/2107.04092.

[54] Ali Salehi. 2D convolution as matrix multiplication using Toeplitz matrices. Technical report, 2018. URL https://raw.githubusercontent.com/alisaaalehi/convolution_as_multiplication/master/ConvAsMulExplained.pdf.

[55] Simon Thorpe, Arnaud Delorme, and Rufin Van Rullen. Spike-based strategies for rapid processing. *Neural Networks*, 14(6-7):715–725, 2001. ISSN 08936080. doi: 10.1016/S0893-6080(01)00083-1.

[56] Timothée Masquelier and Simon J. Thorpe. Unsupervised learning of visual features through spike timing dependent plasticity. *PLoS Computational Biology*, 3(2):0247–0257, 2007. ISSN 1553734X. doi: 10.1371/journal.pcbi.0030031.

[57] Sen Lu and Abhronil Sengupta. Exploring the Connection Between Binary and Spiking Neural Networks. *Frontiers in Neuroscience*, 14(June):1–13, 2020. ISSN 1662453X. doi: 10.3389/fnins.2020.00535.

[58] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[59] Arch D. Robison. N-bit unsigned division via N-bit multiply-add. *Proceedings - Symposium on Computer Arithmetic*, pages 131–139, 2005.