NITI: Training Integer Neural Networks Using Integer-Only Arithmetic

Maolin Wang, Seyedramin Rasoulinezhad, Philip H. W. Leong, *Senior Member, IEEE*, and Hayden K.-H. So, *Senior Member, IEEE*

Abstract—Low bitwidth integer arithmetic has been widely adopted in hardware implementations of deep neural network inference applications. However, despite the promised energy-efficiency improvements demanding edge applications, the use of low bitwidth integer arithmetic for neural network training remains limited. Unlike inference, training demands high dynamic range and numerical accuracy for high quality results, making the use of low-bitwidth integer arithmetic particularly challenging. To address this challenge, we present a novel neural network training framework called NITI that exclusively utilizes low bitwidth integer arithmetic. NITI stores all parameters and accumulates intermediate values as 8-bit integers while using no more than 5 bits for gradients. To provide the necessary dynamic range during the training process, a per-layer block scaling exponentiation scheme is utilized. By deeply integrating with the rounding procedures and integer entropy loss calculation, the proposed scaling scheme incurs only minimal overhead in terms of storage and additional computation. Furthermore, a hardware-efficient pseudo-stochastic rounding scheme that eliminates the need for external random number generation is proposed to facilitate conversion from wider intermediate arithmetic results to lower precision for storage. Since NITI operates only with standard 8-bit integer arithmetic and storage, it is possible to accelerate it using existing low bitwidth operators originally developed for inference in commodity accelerators. To demonstrate this, an open-source software implementation of end-to-end training, using native 8-bit integer operations in modern GPUs is presented. In addition, experiments have been conducted on an FPGA-based training accelerator to evaluate the hardware advantage of NITI. When compared with an equivalent training setup implemented with floating point storage and arithmetic, NITI has no accuracy degradation on the MNIST and CIFAR10 datasets. On ImageNet, NITI achieves similar accuracy as state-of-the-art integer training frameworks without relying on full-precision floating-point first and last layers.

Index Terms—Neural network training, integer arithmetic, NITI, GPU, tensor core, hardware accelerator

1 Introduction

Training deep neural networks (DNNs) from scratch is a lengthy and computationally demanding process that requires a notoriously large number of arithmetic and memory operations. This forms a substantial barrier for rapid deployment and development of new applications and models. Since the success of a DNN implementation depends heavily on the numerical accuracy of the training process, the use of 32-bit single precision floating point arithmetic (`fp32`) has been the default standard for many DNN training frameworks and hardware systems. However, as a continuous quest to improve performance, power and energy efficiency of training DNNs, both in edge scenarios and in cost-sensitive datacenters, there is significant interest in DNN training frameworks and accelerators that can operate with lower bitwidth or non-standard number systems [1], [2]. For instance, modern GPU-accelerated training frameworks already provide facilities to perform mixed-precision DNN training where half precision floating point numbers (`fp16`) are used for most of the training process [3]. In these cases, reducing the data bitwidth from 32 to 16 alone, allows larger models to be trained with better memory bandwidth utilization, and the smaller and more efficient hardware compute units enable improved power and more parallelism.

The challenge, however, is that the reduced dynamic range and accuracy of half precision floating point arithmetic can bring non-trivial degradation to the quality of DNN training. This effect is initialization and training set dependent. As a result, alternative number representations, such as the brain floating point number (`bfloat16`) used in the tensor processing unit (TPU) [4], and other mixed-precision schemes have been proposed [5], [6]. To reduce bitwidth to below 16, Fox *et al*. recently proposed an 8-bit mini block floating point scheme that they have demonstrated to successfully train a range of common neural networks with minimal accuracy loss [7]. In a similar vein, using an 8-bit Posit number system, Lu *et al*. have also demonstrated minimal degradation in training accuracy compared to a `fp32`

- *Maolin Wang is with the ACCESS – AI Chip Center for Emerging Smart Systems, InnoHK Centers, Hong Kong Science Park, Hong Kong, China. E-mail: mlwang@eee.hku.hk.*
- *Seyedramin Rasoulinezhad and Philip H. W. Leong are with the School of Electrical and Information Engineering, The University of Sydney, Camperdown, NSW 2006, Australia. E-mail: raminrasoolinezhad@gmail.com, philip.leong@sydney.edu.au.*
- *Hayden K.-H. So is with the Department of Electrical and Electronic Engineering, University of Hong Kong, Hong Kong. E-mail: hso@eee.hku.hk.*

Manuscript received 2 Sept. 2021; revised 31 Dec. 2021; accepted 3 Feb. 2022. Date of publication 9 Feb. 2022; date of current version 23 May 2022.
This work was supported in part by Croucher Foundation Croucher Innovation Award 2013 and in part by the ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by Innovation and Technology Fund (ITF), Hong Kong SAR.
(Corresponding author: Hayden K.-H. So.)
Recommended for acceptance by A. J. Peña, M. Si and J. Zhai.
Digital Object Identifier no. 10.1109/TPDS.2022.3149787

baseline [8]. As a way to reduce energy consumption in edge applications, Lee *et al.* have also recently demonstrated a mixed `fp8-fp16` scheme in their neural network learning processor [9].

While the above works have made promising progress toward low bitwidth DNN training, they all relied on non-standard floating point arithmetic that requires dedicated and sometimes non-existent hardware support. As illustrated in Table 2, the use of integer or fixed-point arithmetic promises to further reduce hardware complexity and improve energy efficiency when compared to their floating-point counterparts. Furthermore, unlike non-standard floating-point operations, standard integer arithmetic units that operate on low bitwidth data are readily available in a wide range of existing computing systems and accelerators, making an integer-only DNN training framework an attractive alternative for rapid deployment especially in edge scenarios. Unfortunately, training DNNs with integer-only arithmetic is particularly challenging because of their lack of dynamic range and precision that are needed throughout the training process. Previous attempts in integer neural network training have commonly relied on floating-point arithmetic for at least a portion of the process to ensure successful training. Examples include accumulators, gradients, weights, error and activation computation during backpropagation [2], [10], [11].

In this work, we present NITI, a deep neural network training framework that operates *exclusively* with integers, i.e., all parameters and intermediate accumulation values are updated and stored as 8-bit integers. To provide the necessary dynamic range for training, a per-layer block exponentiation scheme is proposed to dynamically adjust the scale of the stored values. Furthermore, to address the unique challenges of training DNN with integers, a novel *discrete* parameter update scheme that allows low precision integer storage of all intermediate variables is proposed. To facilitate rounding of intermediate values to the low bitwidth storage, a hardware-efficient *pseudo stochastic rounding scheme* that utilizes the extra precision in intermediate accumulation as an in-situ random number source has been developed. Finally, an efficient approximation of cross-entropy loss backpropagation is proposed to allow integer-only arithmetic for even the challenging output layer of deep neural networks.

Since NITI operates exclusively with standard integer operations, it can readily be accelerated with existing accelerators that provide high performance integer operations. To illustrate this, an open-source implementation of NITI accelerated with native `int8` matrix-multiplication using Tensor Cores of NVIDIA GPUs is presented. To the best of our knowledge, the proposed implementation is the first to utilize the `int8` inference capability of modern GPUs to accelerate the entire training process without the help of any floating-point arithmetic. Furthermore, to evaluate the hardware benefits of training neural networks using integer arithmetic only, an implementation of NITI's main algorithm using `int8` was compared to an equivalent implementation that employed `fp32` operations on an FPGA.

In terms of training accuracy, we show that NITI can achieve negligible accuracy degradation on the MNIST dataset. On the CIFAR-10 dataset using a network with 9 weight layers similar to VGG [12], also with `int8`, NITI was able to achieve $91.8\,\%$ top-1 validation accuracy compared with $91.5\,\%$ achieved by an equivalent floating-point implementation. On the ImageNet dataset using the original AlexNet, NITI was able to achieve $48.3\,\%$ top-1 compared to $49.0\,\%$ with an equivalent floating point implementation.

In the next section, we summarize related work in integer training. The design of NITI will be presented in Section 3, followed by experimental results in Section 4. The GPU and FPGA acceleration of NITI will be discussed in Section 5. We will conclude and discuss future enhancements to NITI in Section 6.

2 Related Works

The study of hardware-efficient neural network training can be traced to research in low-precision neural network inference, which is a complementary problem. Early studies of binary and ternary neural networks have already demonstrated the feasibility of using hardware-efficient bit-operations to replace complex floating-point multiply-accumulate (MAC) operations during training [13], [14], [15]. Moreover, researchers have also argued that such weight quantization during training could serve as a regularizer that improved the training performance on small datasets [15]. Subsequently, Jacob *et al.* extended quantization to 8 bit weights and activations and demonstrated promising results for deploying DNN models on mobile devices [16]. With a large-scale hardware accelerator, Gupta *et al.* [19] further showed that DNNs could be trained with fixed-point weights using 16-bit precision, and that rounding was crucial to success. They proposed *stochastic rounding* where the probability of rounding x to $\lfloor x \rfloor$ is proportional to their proximity, which has formed the basis of many modern integer training frameworks.

Beyond weight and activations, recent works have begun to address the challenges of quantizing gradient and error computation during the backward pass of training. In the work of DoReFa-Net [17], weight, activation as well as gradients of activations were quantized to allow discretized computation during backpropagation. Similarly, Banners *et al.* developed a bifurcation scheme that quantized gradients of activations during training for 8-bit integer operations in mobile processors [11]. Although promising results have been demonstrated in all the above cases, they have all relied on the use of full-precision floating-point computation in at least some parts of the training process. In [11], [13], [14], [15], [16], [17], `fp32` were used as intermediate storage for weight accumulation, while in [1], [6] they were used as a proxy for the low-precision datatype for computation.

For efficient hardware implementations, a training scheme that employs integer arithmetic exclusively such as this proposed work is highly desirable. There are also platforms where a neural network training method without relying on floating-point is necessary. For example, [20], [21] explored training neural networks with memristor crossbar, whose underlying operations were integer arithmetic instead of floating-point arithmetic. Frequent switching between floating-point format and integer format would be an efficiency burden for memristor-based

TABLE 1
Datatype Comparison for Various Low-Precision Integer DNN Training Frameworks

	$w(infer)$	$w(acc)$	a	g	e	softmax
TTQ[13]	2	32	32	32	32	fp32
Xnor [14]	1	32	1(32)	32	32	fp32
Binaryconnect[15]	1	32	32	32	32	fp32
Jacob *et al*. [16]	8	32	8(32)	32	32	fp32
Dorefa[17]	1	32	2(32)	32	6(32)	fp32
Banner *et al*. [11]	8	32	8(32)	32	8(32)	fp32
WAGE[1]	2	8(32)	8(32)	8(32)	8(32)	fp32
FxpNet[6]	1	12(32)	1(32)	12(32)	12(32)	fp32
[18]	16	32	16	16	16	fp32
NITI (*this work*)	8	8	8	5	8	integer

(32) means the low precision data format is emulated by quantizing `fp32` *number.*

computation. Another example is federated machine learning at wireless edge [22], [23]. In these works, gradients were calculated locally at edge devices that have strict power budgets. An integer-only training framework will save the circuit area reserved for the floating-point unit and leave more area for dedicated integer arithmetic engines, which can be used for both inference and training processes.

The closest works to NITI that we can identify are FxpNet [6], WAGE [1] and [18]. In [18], 16-bit integer arithmetic was used to accelerate forward pass, backward pass and weight update, but weight update still used `fp32` precision. In FxpNet [6], 12-bit fixed point arithmetic was employed throughout the training process to produce a binary network for inference. However, their proposed framework was only tested on CIFAR10 and lacked the ImageNet results. In WAGE, weight, activations, gradients, and error values were all quantized to 8-bit integers to train a ternary network for inference. They achieved moderate accuracy drop using 8 bits to train AlexNet on ImageNet, but the first convolution layer, last fully connected layer, and softmax still uses `fp32` for computation. In contrast, NITI employs integer arithmetic exclusively, even when used in training large networks for classifying ImageNet dataset with 1000 classes using softmax and have achieved only moderate accuracy degradation. A summary of the datatype employed in related works is shown in Table 1.

The goal of NITI is to facilitate efficient hardware accelerator designs for neural network training, particularly in demanding scenarios where area and power-efficiency (throughput/watt) are the dominating optimization goals such as in the edge and in remote deployment locations. From Table 2 below, it is evident that all floating point datatypes (`fp8`, `fp16`, `bfloat16`) consume significantly more resources than `int8`. implementations: $3.1\times$ to $7.2\times$ more LUTs and $4.3\times$ to $7.4\times$ more registers in FPGAs; $2.2\times$ to $4.3\times$ more silicon area in ASICs, which is linearly proportional to power-efficiency. ($\text{power-efficiency} = \text{power}/\ \text{freq} = k \times \text{area}$, where k is a constant.) Even when the additional storage needed for scaling factors and extended intermediate parameters are taken into account, the significant advantages in area and power-efficiency still present a strong case for `int8`-only training accelerators. Moreover, as illustrated by our current implementation using the Tensor Cores in nvidia GPUs, `int8` operations are readily available in today's computer systems, both in the high performance and low power areas, often accelerated by SIMD/vector extensions and GPUs. NITI enables accelerated training on these standard platforms by using `int8` exclusively.

3 An Integer-Only Training Framework

Algorithm 1 shows the overall training algorithm of NITI, which is based on the commonly used stochastic gradient descent (SGD) with backpropagation (BP) scheme. To achieve the goal of computing and storing all values as integers, NITI relies on 3 tightly coupled innovations: (i) a per-layer block exponentiation scaling scheme for integer values; (ii) a discrete weight update scheme; and (iii) a shift-and-round scheme for intermediate values that integrates with the above two.

The judicious use of datatype is essential to the success of NITI. Fig. 1 shows the mathematical symbols and their associated datatypes for various parts of the NITI algorithm. To facilitate discussions, throughout this work, vectors are named with lower case alphabets and are typeset in boldface (e.g., $\boldsymbol{a}$) while matrices are named with upper case alphabets (e.g., $\boldsymbol{X}$). Scalars are typeset with normal fonts (e.g., a) with an optional parentheses in superscript (e.g., $a^{(l)}$) denoting the layer l that it belongs. Furthermore, unless otherwise noted, all scalar variables are represented as 8-bit

TABLE 2
Preliminary Results of Multiply-Accumulate (MAC) Operator Hardware Resource Consumption Using Low-Precision Datatype Proposed in Related Works

Datatype	Input Data Config	Accumulate Data Config	FPGA Implementations			ASIC Implementations	
			LUT	Reg	freq (MHz)	area (μm^2)	freq (MHz)
`fp16`	(1,5,10)	(1,5,10)	506	152	528	1412	1077
`bfloat16`	(1,8,7)	(1,8,7)	419	152	449	1165	1766
`fp8` [5]	(1,5,2)	(1,5,10)	362	136	556	980	1522
`fp8`[25]	(1,4,3)	(1,6,9)	395	140	554	1065	1579
`fp8` [26]	(1,5,2)	(1,8,23)	886	238	425	1972	1096
`int16`	`int16`	int32	345	32	775	903	1001
`int8` (NITI)	`int8`	int32	120	32	775	455	1001

Designs generated with deepfloat framework[24], synthesized for Xilinx FPGAs using Vivado, and synthesized for ASIC with Yosys targeting FreePDK45nm process. Floating point designs pipelined to maintain reasonable clock speed. Generic `fp16` *and* `bfloat16` *configuration were used for accumulation. Floating point data configuration* $(\mathrm{s}, \mathrm{e}, \mathrm{m})$ *stands for the number of bits devoted to sign, exponent and mantissa.*

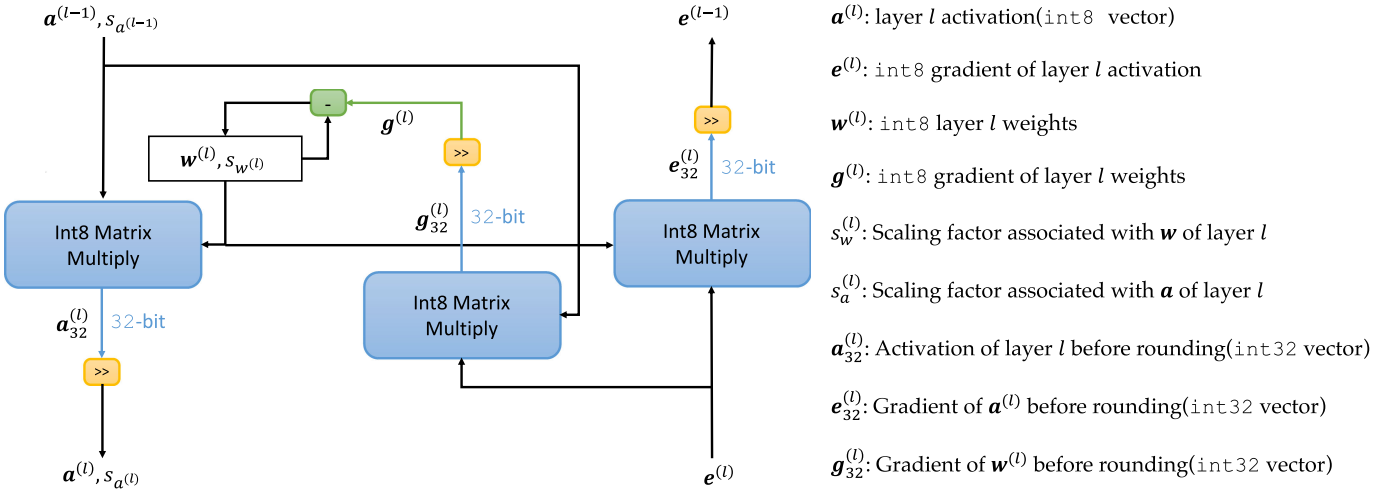


Fig. 1. Integer layer forward and backward pass.

signed integers (int8). If a scalar variable is represented with more than 8 bits, then the variable is subscript with the number of bits used. For example, a_{32} is a scalar represented as 32-bit signed integer (int32). As shown in Fig. 1, except for quantizing floating point (fp32) data input during initialization of the training scheme, no floating point arithmetic and no further quantization is performed in NITI.

Algorithm 1. Forward and Backward Passes in NITI for Each Batch of Data X and Label Y

/* Forward Pass, with quantized input $\boldsymbol{a}^{(0)}, s_{a^{(0)}}$ from X */
for each layer l **do**
  $\boldsymbol{a}_{32}^{(l)}, s_{a^{(l)}} \leftarrow \text{INT8MATRIXMULTIPLY}\boldsymbol{a}^{(l-1)}, \boldsymbol{w}, s_{a^{(l-1)}} + s_w$;
  $b \leftarrow \text{EFFECTIVEBITWIDTH}\boldsymbol{a}_{32}^{(l)}$;
  $\boldsymbol{a}^{(l)}, s_{a^{(l)}} \leftarrow \text{SHIFTANDROUND}\boldsymbol{a}_{32}^{(l)}, s_{a^{(l)}}, \max(0, b-7)$;
end
/* Backward Pass, with $\boldsymbol{a}, s_a$ being final layer's output, scale */
$\boldsymbol{e} \leftarrow \text{INT8LOSSGRADIENT}\boldsymbol{a}, s_a, Y$;
for each layer l in reverse order **do**
  $\boldsymbol{g}_{32}^{(l)} \leftarrow \text{INT8MATRIXMULTIPLY}\boldsymbol{a}^{(l-1)}, \boldsymbol{e}^{(l)}$;
  $\boldsymbol{e}_{32}^{(l-1)} \leftarrow \text{INT8MATRIXMULTIPLY}\boldsymbol{w}^{(l)}, \boldsymbol{e}^{(l)}$;
  $b \leftarrow \text{EFFECTIVEBITWIDTH}\boldsymbol{e}^{(l-1)}$;
  $\boldsymbol{e}^{(l-1)}, \_ \leftarrow \text{SHIFTANDROUND}\boldsymbol{e}_{32}^{(l-1)}, \_, \max(0, b-7)$;
  $b \leftarrow \text{EFFECTIVEBITWIDTH}\boldsymbol{g}_{32}^{(l)}$;
  $\boldsymbol{g}^{(l)}, \_ \leftarrow \text{SHIFTANDROUND}\boldsymbol{g}_{32}^{(l)}, \_, \max(0, b-m_u)$;
  $\boldsymbol{w}^{(l)} \leftarrow \boldsymbol{w}^{(l)} - \boldsymbol{g}^{(l)}$;
end

3.1 A Block Exponentiation Scaling Scheme

To allow the range of representable values be dynamically adjustable during training, the neural network model's activations ($\boldsymbol{a}$), gradients of activations ($\boldsymbol{e}$) and gradients of weights ($\boldsymbol{g}$) are each coupled with their own associated scaling exponent s_a, s_e and s_g respectively. All elements of the vector share the same scaling exponent, making the scaling exponent adjustable for each layer of the model. A scaling exponent s carries a weight of 2^s. For example, given an activation vector $\boldsymbol{a}$ and its coupled scaling exponent s_a, the actual values of the model activation are $\boldsymbol{a} \cdot 2^{s_a}$. Since the scaling exponent is shared among the entire vector and is representable as an int8 value, the storage overhead is minimal. The computation overhead of the block scaling exponentiation is also minimal. For instance, the scaling factor of the convolution output between $\boldsymbol{w} \cdot 2^{s_w}$ and $\boldsymbol{a} \cdot 2^{s_a}$ can be obtained by a single int8 addition.

Most importantly, when compared with other works that utilize similar block scaling schemes, our training framework handles the scaling factors of $\boldsymbol{a}$, $\boldsymbol{w}$, $\boldsymbol{e}$ and $\boldsymbol{g}$ differently in order to optimize the training performance and minimize computation overhead. In the case of activations ($\boldsymbol{a}$), s_a is passed along with layer activation and is adjusted dynamically to help the results fully utilize the int8 range after shift-and-round (Section 3.4). Furthermore, the cross entropy loss is also approximated with integer arithmetic based on the s_a of the final layer. In the case of $\boldsymbol{e}$ and $\boldsymbol{g}$, the learning rules in NITI have been co-designed with the integer cross entropy loss computation to embed the computation of the scaling exponents s_g and s_e into the main algorithm. As a result, although s_g and s_e are part of the underlying training logic, their computation can be optimized out in the final implementation, as indicated by the underscores in Algorithm 1. See Sections 3.3 and 3.5 for details.

Finally, s_w remains static during the entire training. As shown in Fig. 1, $\boldsymbol{w}$ is not the result of a matrix multiplication. Its bitwidth won't be expanded dramatically like $\boldsymbol{a}, \boldsymbol{e}$ and $\boldsymbol{g}$. Hence, using shift-and-round to adjust s_w dynamically is not necessary.

3.2 Forward and Backward Pass

At the core of NITI are the forward pass and backward pass of convolution and fully connected layers performed with integer-only arithmetic. For convolution layers, our framework employed implicit GEneral Matrix Multiply(GEMM) to reduce convolution operations into matrix-matrix multiplications. As a result, similar to the case of training fully connected layers, integer matrix multiplies form the dominating operation in our training framework. With our use of int8 as input, we were able to accelerate this matrix

multiply in our current implementation by leveraging the newly introduced integer matrix multiply function unit in the latest GPUs that were originally designed for inference acceleration [27].

During the forward pass (left side of Fig. 1), activation of previous layer ($\boldsymbol{a}^{(l-1)}$) enters this layer as `int8` with a scaling factor $s_{a^{(l-1)}}$. The results of the matrix multiply are accumulated in `int32` precision and must be rounded back to `int8` before propagating to the next layer. This rounding operation is very important to the success of low precision training and repeatedly appears in forward pass, backward pass and model weights update. They are shown as shift operators in Fig. 1 as they are combined with the scaling operation in our scheme.

In theory, the output scaling factor $s_{a^{(l)}}$ is simply a sum of the 2 input scaling factors $s_{a^{(l-1)}}$ and s_w. However, to make full use of the signed 8-bit precision to represent the integer part of the results, an additional shift operation is performed based on the *effective bitwidth* of the 32-bit matrix multiply output. Here, effective bitwidth of an integer matrix $\boldsymbol{V}$, denoted as $B(\boldsymbol{V})$, is defined as the minimum number of bits required to fully represent the maximum value $v \in \boldsymbol{V}$. If $b = B(\boldsymbol{V}) > 7$, then the 32-bit results are shifted right by $b - 7$ bit, just enough to maximize the use of the `int8` datatype. The fractional part is rounded off using the pseudo stochastic rounding scheme described in Section 3.4 and the scaling factor is adjusted accordingly.

The backward pass involves propagating the error back through the integer network and computing the gradient of weights in each layer for weight update as shown in the right hand side of Fig. 1. The 32-bit errors are rounded similarly to the forward pass back into 8-bit values before being propagated to the next layer. The 32-bit gradients are rounded with special procedures that fuse with the weight update process as explained in Section 3.3.

3.3 Weight Update

Normally with SGD, the gradient $\boldsymbol{g}^{(l)}$ of layer weights can be computed by a matrix multiplication between its activation input $\boldsymbol{a}^{(l-1)}$ and gradients of its activation output $\boldsymbol{e}^{(l)}$. The model weights could subsequently be updated by some combinations of this gradient $\boldsymbol{g}$, the global learning rate and other heuristics to determine the amount of weight update.

However, due to the limited range of our `int8` weights $\boldsymbol{w}$, the task of maintaining any necessary precision is challenging, notably due to the range discrepancy between $\boldsymbol{w}$ and the desired update value. For example, we observed empirically that even in a medium-size network, direct application of typical learning rules often resulted in overflow or underflow in the weight update. Consequently, most of the update values are saturated as ± 127 or 0.

Instead, we propose a learning heuristic that combines update with the rounding of the gradient for weight computation in `int8`. The proposed learning heuristic draws inspiration from the Resilient backpropagation (RPROP) algorithm [28]. In the original RPROP algorithm, $\boldsymbol{w}$ was updated with only the sign of $\boldsymbol{g}_{32}^{(l)}$. In our case, in addition to the sign, we also utilize the effective bitwidth of $\boldsymbol{g}_{32}^{(l)}$ to decide the magnitude of the update.

In particular, let $b = B(\boldsymbol{g}_{32}^{(l)})$ be the effective bitwidth of $\boldsymbol{g}_{32}^{(l)}$, then $\boldsymbol{g}_{32}^{(l)}$ is shifted and rounded by $b - m_u$ bits to obtain an effectively m_u bits weight update value $\boldsymbol{g}^{(l)}$. Recall that the value of s_w for each layer is set during initialization and remains unchanged during training. As a result, the proposed update heuristic essentially operates by updating $\boldsymbol{w}$ with small quantum $\boldsymbol{g}^{(l)} \cdot 2^{s_w}$. We have evaluated different values of m_u and have determined that values of m_u in the range of 1 to 5 bits performed well in general. We show empirically that this learning rule has comparable convergence speed on MNIST and CIFAR10 to SGD with momentum, which is commonly used in floating point training. See Section 4.3 for experimental results. On ImageNet, this learning rule has comparable convergence speed with SGD without momentum.

3.4 Shifting and Rounding

Mapping `int32` results from matrix-multiply back to `int8` data for downstream computation is a crucial step that has a significant influence in the final training accuracy. To propagate activations and errors during training, a special *shift and round* scheme defined in Algorithm 2 is employed. Using this scheme, the values in concern (i.e., $\boldsymbol{a}$ and $\boldsymbol{e}$) are first logically shifted right by an amount that is determined by their effective bitwidth $B(\boldsymbol{a})$ and $B(\boldsymbol{e})$ respectively (See lines 4 and 11 in Algorithm 1). This shift avoids overflow and maximize the number of useful bits in the final `int8` representation. In addition, the corresponding activation scaling factor s_a is adjusted according to maintain correct magnitude. Next the `int32` values are rounded to `int8`. In the context of NITI, rounding refers to the task of mapping a 32-bit *fixed point* number $x = \langle q.f \rangle$ to a nearby 8-bit integer $\tilde{x}$. In this notation, q and f are the integer and fraction parts of x respectively, and the binary point is located at position bp, which is equal to the bitwidth of f.

Algorithm 2. SHIFTANDROUND

Input: q_{32} : 32-bit fixed point number,
$s_{q_{32}}$: the scaling factor of q_{32},
bp : Binary point
Output: q : Rounded 8-bit integer,
s_q : the scaling factor of q

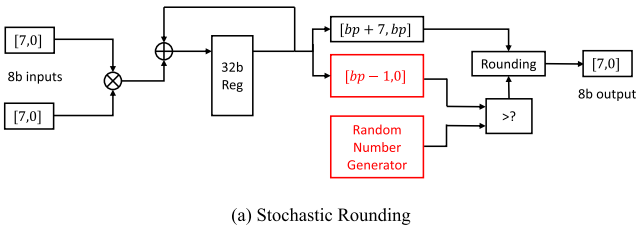
1: $s_q \leftarrow s_{q_{32}} + bp$;
2: $q \leftarrow \text{ROUND}q_{32}, bp$;

As discussed in [19], the use of stochastic rounding with zero rounding bias is crucial to the success of training neural networks with low precision. Stochastic rounding $\text{Rs}\cdot$ can be defined as:

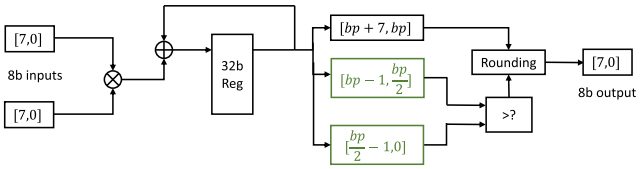
$$\text{Rs}(q.f) = \begin{cases} q & \text{with probability } 1 - f \cdot 2^{-bp} \\ q+1 & \text{with probability } f \cdot 2^{-bp} \end{cases}$$

A typical implementation of the stochastic rounding function would therefore compute the rounding probability by comparing f with a random number r, as shown in Fig. 2.

Instead of relying on an external random number source to produce r, we propose a hardware-efficient *pseudo stochastic rounding* algorithm that generates in-situ pseudo



(a) Stochastic Rounding



(b) Pseudo Stochastic Rounding

Fig. 2. Rounding Schemes Hardware Implementation.

random numbers using the additional bits from the 32-bit input. As shown in Algorithm 3, the proposed scheme operates in ways similar to the original stochastic rounding function, except the stochastic rounding decision is now based on comparing values between different portions of f, the fractional parts of the input. Our current implementation divides the fractional bits into 2 halves for comparison. The more significant (top) half of f is essentially a truncated version of f, and the less significant (bottom) half of f now serves as a pseudo random number. As shown in Section 4.2, the proposed pseudo stochastic rounding scheme is able to support training with comparable accuracy as the original stochastic rounding scheme.

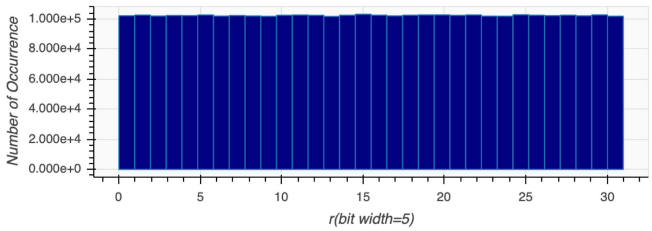
Algorithm 3. Pseudo Stochastic Rounding

Input: q_{32}: 32-bit fixed point number, bp: Binary point
Output: Rounded 8-bit integer q
/* Extract bits for integer (q) and fractional (f) parts from $|q_{32}|$ */
$q,\ f \leftarrow |q_{32}|[bp+7, bp],\ |q_{32}|[bp-1, 0]$;
if bp is odd **then**
  $f \leftarrow f \gg 1$ /* right shift by 1 bit */
  $bp \leftarrow bp - 1$;
end
if $f[bp-1, \frac{bp}{2}] > f[\frac{bp}{2} - 1, 0]$ **then**
  $q \leftarrow q + 1$
end
$q \leftarrow q \cdot \mathrm{sgn}(q_{32})$

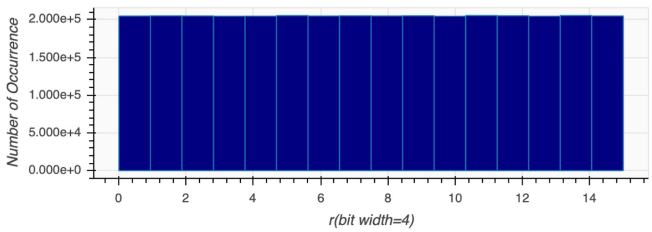
Fig. 3 shows histograms of the pseudo random number used for rounding $\boldsymbol{a}$, $\boldsymbol{e}$, $\boldsymbol{g}$ in one convolution layer during the course of training. The figures show that our scheme is able to produce overall random numbers with near uniform distribution. Similar distribution in other convolution and fully connected layers can also be made.

3.5 Integer Cross Entropy Loss

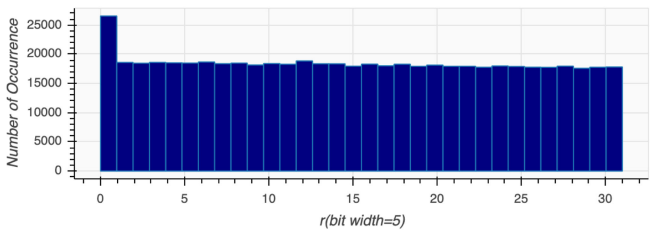
After computing the final layer activations, $\boldsymbol{a} \cdot 2^{s_a}$, NITI computes the predicted probability of each class for the input image using a softmax layer. Assume we have N classes with $i \in \{1 \dots N\}$, then the predicted probability of class i is



(a) Activation $\boldsymbol{a}$



(b) Error $\boldsymbol{e}$



(c) Gradient $\boldsymbol{g}$

Fig. 3. Histogram of pseudo random number (r) used for rounding different variables during training of one convolution layer.

$$\hat{y}_i = \frac{e^{a_i \cdot 2^{s_a}}}{C}$$

where $C = \sum_{i=1}^{N} e^{a_i \cdot 2^{s_a}}$.

With the predicted probability $\hat{\boldsymbol{y}}$ and the target one hot probability $\boldsymbol{y}$ obtained from the labels, the backpropagation process can be initiated by computing the gradient $\boldsymbol{e}$ of cross entropy loss, E. Cross-entropy loss is defined as

$$E = -\sum_{i=1}^{N} y_i \ln(\hat{y}_i)$$

and its partial derivative can be computed as:

$$\begin{aligned} e_i &= \frac{\partial E}{\partial (a_i \cdot 2^{s_a})} \\ &= \hat{y}_i - y_i \\ &= \frac{1}{C}\left(e^{a_i \cdot 2^{s_a}} - y_i C\right) \qquad i \in \{1 \dots N\} \end{aligned} \tag{1}$$

Consider (1), although the factor $\frac{1}{C}$ is difficult to compute using integer arithmetic, note that mathematically it is a constant (independent of i) that affects only the scaling of e_i. In other words, it only affects the scaling factor (s_e) of $\boldsymbol{e}$ but not the relative values among the elements of $\boldsymbol{e}$. Now, as shown in Section 3.3, we have designed our learning rule in such a way that it doesn't depend on s_g. Consequently, the actual value of s_e also has no effect on the final results and thus the computation of $\frac{1}{C}$ can be avoided entirely. The main challenge of implementing (1) with integer arithmetic is

therefore to approximate the term $t_i = e^{a_i \cdot 2^{s_a}}$ accurately with limited precision. We address this challenge by considering 2 cases.

When $s_a \leq -7$, we know $|a_i \cdot 2^{s_a}| < 1$ because, as an 8-bit integer, $a_i \leq 127$. We can therefore approximate t_i via its Taylor expansion:

$$\begin{aligned} t_i &= e^{a_i \cdot 2^{s_a}} \\ &\approx 1 + a_i \cdot 2^{s_a} + \frac{1}{2} a_i^2 \cdot 2^{2s^a} \end{aligned} \tag{2}$$

When $s_a > -7$, we rearrange the term with a base-2 approximation as follows:

$$\begin{aligned} t_i &= e^{a_i \cdot 2^{s_a}} \\ &= 2^{(\log_2 e) \cdot a_i \cdot 2^{s_a}} \end{aligned} \tag{3}$$

Note that in both cases, 2^x corresponds to a shift operation in integer fixed point representations if the exponent x is an integer. In the case of (2), since s_a and $2s_a$ are both integers, t_i can be computed by using simple integer add and shift. In the case of (3), the value of t_i is a power-of-2, but the exponent value is not necessarily an integer.

To efficiently approximate the exponent in (3) as an integer, denote the exponent in (3) as $x_i = (\log_2 e) \cdot a_i \cdot 2^{s_a}$. Now,

$$(\log_2 e) = 1.44269 \approx 47274 \cdot 2^{-15}$$

is a constant, which allows us to approximate x_i as

$$\begin{aligned} x_i &\approx (47274 \cdot 2^{-15}) \cdot a_i \cdot 2^{s_a} \\ &= (47274 a_i) \cdot 2^{s_a - 15} \end{aligned} \tag{4}$$

In (4), the constant $47274 = \texttt{0xB8A8}$ in hex requires 16 bits to represent, while a_i is an 8-bit integer, making their product a 24 bit integer. We observe empirically that $s_a < 0$ in most cases. Therefore, x_i can be approximated by shifting the value $47274 a_i$ *right* by $|s_a| + 15$ followed by truncation of any resulting fractional bits. Denote this integer approximation of x_i as $\hat{x}_i$.

Although t_i is now ready to be approximated by $2^{\hat{x}_i}$ with a shift operation, $2^{\hat{x}_i}$ can still be a very long integer if $\hat{x}_i$ is large. We apply an offset p to $\hat{x}_i$, so $t_i = 2^p \cdot 2^{\hat{x}_i - p}$. The term 2^p can be absorbed into $\frac{1}{C}$ and its computation can also be avoided. Note that for any value $\hat{x}_i < p$, after the shift operation, it's at least $2^{max(\hat{\boldsymbol{x}}) - p}$ times smaller than the maximum value in t_i which is $2^{max(\hat{\boldsymbol{x}})}$. We choose p be the smallest value of $\hat{x}_i$ larger than $\max(\hat{\boldsymbol{x}}) - 10$, so for any value $\hat{x}_i < p$, after the shift operation, it is at least 2^{10} smaller than $2^{max(\hat{\boldsymbol{x}})}$. These small values don't influence e_i after rounding back to `int8`, so they can be safely clipped away. Denote $\tilde{x}_i = \max(0, \hat{x}_i - p)$ to be the values after applying the offset and clipping to $\hat{x}_i$. Finally, $t_i = 2^{x_i}$ is approximated as $2^p \cdot 2^{\tilde{x}_i}$. The computation of 2^p can be avoided as mentioned before. $2^{\tilde{x}_i}$ can be computed by $1 \ll \tilde{x}_i$ and its maximum value is 2^{10}. After substituting t_i back to Equation (1), e_i can be represented by a 10-bit integer times an scaling factor for the case $y_i = 0$. For the case $y_i = 1$, we need to add all $2^{\tilde{x}_i}$. In this addition step, we use the same bitwidth as the intermediate accumulation of `int8` matrix multiplication.

The error tensor $\boldsymbol{e}$ in (1) eventually rounded stochastically back to 8 bits before being used in back propagation.

TABLE 3
Average Top-1 Validation Accuracy(5 Runs) of Training VGG-Small-7 on CIFAR10 With Different Weight Initialization Schemes

Distribution	Validation Accuracy(%)
Normal	87.3 ± 0.3
Uniform	90.8 ± 0.2

The rounding scheme for $\boldsymbol{g}$ was stochastic rounding. The rounding scheme for $\boldsymbol{a}, \boldsymbol{e}$ was round-to-nearest. m_u was initiated by 5 bits. It was dropped to 4 bits at the 100th and 3 bits at the 150th epoch. The training was stopped at the end of the 200th epoch.

For a dataset with 1000 classes like ImageNet, stochastic rounding is necessary to avoid round bias in e_i. Despite the crude approximation in some cases, we find the accuracy of the resulting network competitive to related works that depend on floating-point implementations.

4 Experimental Results

In this section, we first evaluate the key factors which influence the performance of trained networks by NITI framework including weights initialization schemes and different rounding schemes for $\boldsymbol{g}$. To do so, we used CIFAR10 [29] datasets with a network with 7 weight layers similar to VGG [12](referred as VGG-small-7 in this section). Detailed network configuration is shown in the first column of Table 5. Learning from the above evaluation, we optimized our NITI framework using the proposed rounding schemes and weight initialization scheme. Then, we compare the performance of NITI framework with its floating point counterpart trained by SGD optimizer over MNIST [30], CIFAR10, and ImageNet [31] datasets.

4.1 Weight Initialization

As mentioned in Section 3.3, the training updates are performed directly on `int8` weights, which have significantly less dynamic range than `fp32` weights. Related mixed precision training works[2], [11] usually initialize the weights with a zero-mean normal distribution, just like the floating-point training. Such initialization is rather sparse in the sense that most of the values are small around zero. We try to compensate the limited weight update precision by a denser weight initialization using uniform distribution. As for the scaling factor $s_w^{(l)}$ associated with the integer weights $\boldsymbol{w}^{(l)}$, it is determined by similar rules used in [32], [33]. The idea is that the scaling factor should be initialized in such a way to avoid inputs being amplified.

The task of training VGG-small-7 on CIFAR10 was chosen to demonstrate the efficiency of two weight initialization schemes: weights with uniform(dense) distribution and normal(sparse) distribution. The average top-1 validation accuracy of 5 training experiments with each initialization scheme is shown in the Table 3. As shown in the table, we observed that using uniformly distributed(dense) network weight initialization generally results in better validation accuracy.

Fig. 4 compared the weight distribution of a typical convolution layer inside VGG-small-7 before and after the training under the two initialization schemes. The learning heuristic introduced in Section 3.2 adjusts weights to have

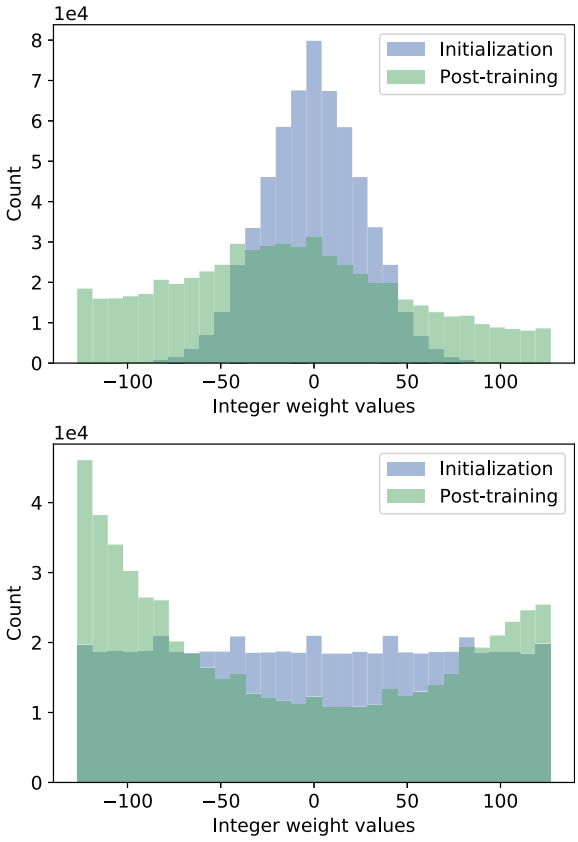


Fig. 4. Weight distribution before and after training using normal(up) and uniform initialization(down).

fewer zeros with proper initialization. We also observed similar weight distribution while using NITI to train networks shown in Table 5 on CIFAR10 and AlexNet on the ImageNet dataset.

We hypothesize that floating-point training has more redundancies in the weight precision, so they tend to have more zero weights. However, if we accumulate the weights in int8 precision directly, the final trained model compensates for the limited precision with more non-zeros weights. Some works [34], [35] demonstrate the trade-off between sparsity and weights bit width in the sense of inference acceleration. However, such a compensation mechanism has never been demonstrated in previous low precision training works. We also observe that weights in the first layer are almost zero-free, both in CIFAR10 and ImageNet experiments. We think that is why related low precisoin training works [1], [2] needs to keep the first layer as floating-point precision to prevent accuracy loss.

4.2 Rounding Scheme

As stochastic rounding is adopted by most recent low precision training implementations [1], [6], [11], we use it as the baseline for evaluating our pseudo stochastic rounding scheme. We also include results from straight-forward round-to-nearest for comparison. Table 4 shows the averge top-1 validation accuracy of 5 experiments for training `int8` VGG-small-7 on CIFAR10 dataset with different rounding schemes. Our pseudo stochastic rounding technique works as well as baseline stochastic rounding while both significantly outperform round-to-nearest method in terms of final best validation accuracy.

TABLE 4
Average Top-1 Validation Accuracy(5 Runs) of Training VGG-Small-7 on CIFAR10 With Different Rounding Schemes for g

Rounding schemes	Validation Accuracy(%)
Round-to-nearest	89.5 ± 0.1
Stochastic	90.8 ± 0.2
Pseudo Stochastic	90.7 ± 0.1

The rounding scheme for $\boldsymbol{a}, \boldsymbol{e}$ was round-to-nearest. Weights were initiated using uniform distribution. m_u schedule was the same as Table 3.

4.3 Accuracy Results on MNIST and CIFAR10 Datasets

We trained LeNet [36] and VGG-small networks respectively over MNIST and CIFAR10 datasets, using both optimized NITI framework and the baseline `fp32` method. The `fp32` baseline technique uses SGD with momentum 0.9. For the case of LeNet model, we fixed the learning rate in `fp32` training to 0.01 and m_u to 3 bits in `int8` training. Both `fp32` and `int8` training processes were stopped at the end of 20th epoch. In the case of VGG-small, we initiated the learning rate by 0.01 in `fp32` training. It was dropped to 0.001 at 100th epoch and 0.0001 at 150th epoch. In `int8` training, we initiated m_u by 5 bits. It was dropped to 4 bits at 100th epoch and 3 bits at 150th epoch. The rounding scheme for $\boldsymbol{a}, \boldsymbol{e}$ was round-to-nearest. The rounding scheme for $\boldsymbol{g}$ was pseudo-stochastic rounding. Weights were initiated using uniform distribution. Both `fp32` and `int8` training processes were stopped at the end of 200th epoch when the training accuracy stopped improving.

As demonstrated in Fig. 5, NITI trains LeNet model, which is a relatively small network, with almost identical performance comparing to the `fp32` baseline. In the case of training VGG-small-7 over CIFAR10, which is a tougher task, NITI was able to successfully train the network with negligible descend, achieving a top-1 validation accuracy of $90.7\,\%$. Table 6 summarizes the validation accuracy of our framework on CIFAR10 while increasing network depth. As observed, deeper network produced the better accuracy by NITI. NITI even produced slightly better generalization performance than the `fp32` baseline for VGG-small-8 and VGG-small-9 (Fig. 6).

TABLE 5
VGG-Small Network Configurations

7 weight layers	8 weight layers	9 weight layers
conv3-128	conv3-128	conv3-128
conv3-128	conv3-128	conv3-128
	conv3-128	conv3-128
maxpool		
conv3-256	conv3-256	conv3-256
conv3-256	conv3-256	conv3-256
		conv3-256
maxpool		
conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512
maxpool + dropout + FC		

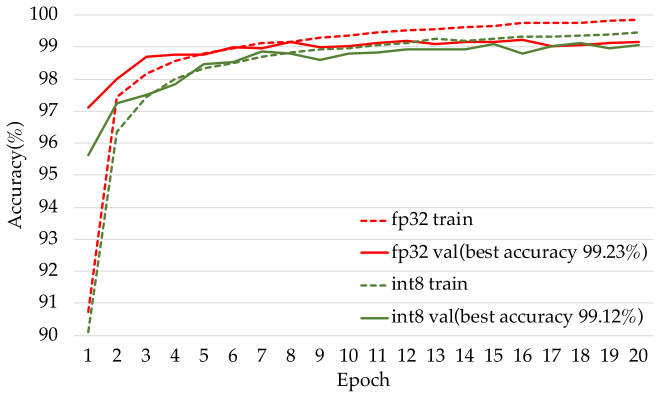


Fig. 5. Training performance comparing NITI (`int8`) and baseline `fp32` implementation on MNIST.

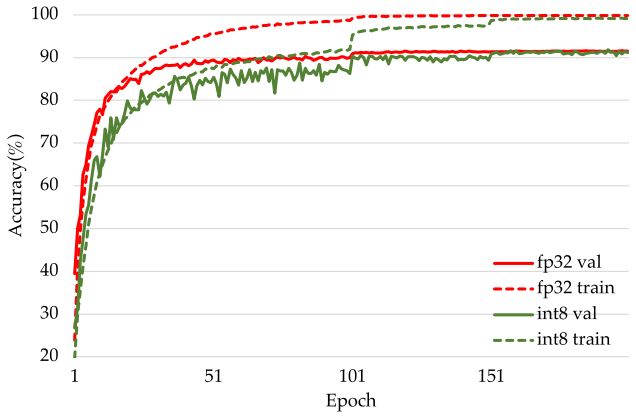


Fig. 6. Training VGG-small (9 weight layers) comparing NITI (`int8`) and baseline `fp32` implementation on CIFAR10.

4.4 Accuracy Results on ImageNet Dataset

Table 6 tabulates the results of several related integer-training works on different datasets. The `fp32` baseline AlexNet were trained using vanilla SGD without momentum, weight decay, dropout, and batch normalization to match with the training setup of WAGE [1]. Initial learning rate was 0.01 and dropped to 10^{-3} and 10^{-4} at epoch 60 and 65 respectively. Total training epochs were 70. Since the original publication of [11] did not include AlexNet results, we obtained the comparison results by using their released code.

As mentioned in Section 2, both [11] and [1] have maintained some degrees of computation using floating point arithmetic during their training process. The result of [1] further excluded cross entropy loss and the last layer from quantization to avoid performance degradation. [18] preserved even more floating point computations in the training. Their accuracy on ImageNet is significantly higher than the comparisons because momentum, weight decay and batch normalization are all computed with floating point precision. Being the only work that performed network training exclusively with integer arithmetic, NITI achieves similar accuracy on the task of training integer AlexNet compared with state-of-the-art integer training frameworks. We plan to test the NITI algorithm on more network architectures in the future.

5 Acceleration Results

To study the benefits of using `int8` operations in accelerating neural network training with NITI, two acceleration strategies have been explored. First, we explored the use of native `int8` matrix multiplication in modern GPUs to accelerate the computational bottleneck of NITI. Then, an FPGA accelerator that performed the SGD backpropagation algorithm for a fully connected layer was implemented to study the resource and speed advantage of `int8` computations over their floating-point counterparts.

5.1 Tensor Cores Acceleration in GPU

Recognizing the importance of `int8` operations in accelerating deep quantized neural network inference, many modern GPUs and AI accelerators have been introducing compute engines that are dedicated to these low precision integer operations. Since NITI relies on `int8` computation natively, here we evaluate how these accelerators originally designed for inference acceleration may be used to accelerate neural network training as well.

In particular, we utilized the tensor cores introduced in modern nvidia GPUs to accelerate the most time-consuming part in the training process, namely, the computation of $\boldsymbol{a}$, $\boldsymbol{e}$,

TABLE 6
Comparison to Related Works

Dataset	Model	Size	Related works	$w(infer)$	$w(acc)$	a	g	e	Loss	Accuracy	Memory
MNIST	LeNet 5	6.87K	fp32 baseline	fp32	fp32	fp32	fp32	fp32	fp32	99.2%	2019 MiB
MNIST	LeNet 5	6.87K	NITI (*this work*)	8	8	8	5	8	integer	99.1%	1538 MiB
CIFAR10	VGG-small 7	4.66M	fp32 baseline	fp32	fp32	fp32	fp32	fp32	fp32	$91.0 \pm 0.1\%$	5825 MiB
CIFAR10	VGG-small 7	4.66M	NITI (*this work*)	8	8	8	5	8	integer	$90.7 \pm 0.1\%$	3169 MiB
CIFAR10	VGG-small 8	4.72M	fp32 baseline	fp32	fp32	fp32	fp32	fp32	fp32	$91.3 \pm 0.2\%$	5957 MiB
CIFAR10	VGG-small 8	4.72M	NITI (*this work*)	8	8	8	5	8	integer	$91.5 \pm 0.2\%$	3301 MiB
CIFAR10	VGG-small 9	5.39M	fp32 baseline	fp32	fp32	fp32	fp32	fp32	fp32	$91.5 \pm 0.1\%$	6033 MiB
CIFAR10	VGG-small 9	5.39M	NITI (*this work*)	8	8	8	5	8	integer	$91.8 \pm 0.2\%$	3303 MiB
CIFAR10	VGG-like	9.29M	FxpNet[6]	1	12	1	12	12	fp32	88.5%	-
ImageNet	AlexNet	62.38M	fp32 baseline	fp32	fp32	fp32	fp32	fp32	fp32	49.0%	7389 MiB
ImageNet	AlexNet	62.38M	Banner *et al.* [11]	8	32	8	8	8	fp32	48.9%	9317 MiB
ImageNet	AlexNet	62.38M	WAGE[1]	2	8(32)	8(32)	8(32)	8(32)	fp32	48.4%	-
ImageNet	AlexNet	62.38M	NITI (*this work*)	8	8	8	5	8	integer	48.3%	6369 MiB
ImageNet	AlexNet	62.38M	DFP-16[18]	16	fp32	16	fp32	16	fp32	56.9%	-

(1) The VGG-small series results on CIFAR10 are the average top-1 validation accuracy of 5 runs.
(2) The "Size" column lists the total number of parameters in the model.
(3) The "Memory" column lists memory footprints of training on RTX 3090 GPU. Some data are missing("-") because codes are not available.

TABLE 7
Speedup of Convolution Layers on GPUs Using `int8` Compared With Using `fp32`

Computing a						
GPU	RTX 2080 Ti			RTX 3090		
input	int8(ms)	fp32(ms)	speedup	int8(ms)	fp32(ms)	speedup
224	11.624	51.092	4.40	3.928	13.033	3.32
112	3.069	13.301	4.33	1.006	3.214	3.19
56	1.166	3.821	3.28	0.289	1.125	3.90
28	0.367	1.425	3.88	0.089	0.311	3.51
14	0.101	0.684	6.76	0.038	0.163	4.30
Computing e						
GPU	RTX 2080 Ti			RTX 3090		
input	int8(ms)	fp32(ms)	speedup	int8(ms)	fp32(ms)	speedup
224	10.985	96.088	8.75	4.731	17.765	3.75
112	3.091	24.509	7.93	1.269	4.337	3.42
56	1.191	6.804	5.71	0.453	1.448	3.20
28	0.343	2.284	6.65	0.185	0.362	1.96
14	0.228	0.992	4.35	0.117	0.150	1.29
Computing g						
GPU	RTX 2080 Ti			RTX 3090		
input	int8(ms)	fp32(ms)	speedup	int8(ms)	fp32(ms)	speedup
224	365.696	535.745	1.46	122.301	462.291	3.78
112	109.723	132.783	1.21	27.327	130.079	4.76
56	27.611	43.139	1.56	6.911	32.083	4.64
28	5.061	10.695	2.11	1.074	0.458	0.43
14	1.466	3.632	2.48	0.272	0.178	0.65

All the convolution layers had the same configuration(batch size 64, 64 input channels, 128 output channels, 3×3 kernel, stride 1, and padding 1) except the input size.

and $\boldsymbol{g}$ of convolution layers as shown in Fig. 1. Tensor cores in nvidia GPUs are designed to perform `int8` matrix multiplications with high efficiency.

The use of `int8` can lead to 4 times improvement in the peak computation throughput of tensor cores in NVIDIA's Ampere architecture [37] when compared to `fp32` operations.

Accelerating the computation of $\boldsymbol{a}$ with `int8` precision on tensor cores was well supported on commonly used DNN training frameworks like PyTorch [38] and TensorFlow [39]. However, the current API (cuDNN v8.2 [40]) for tensor core operations does not provide native support for computing $\boldsymbol{e}$ and $\boldsymbol{g}$ with `int8` precision directly. Although the underlying computations of $\boldsymbol{e}$ and $\boldsymbol{g}$ are matrix multiplications similar to the computation of $\boldsymbol{a}$ as shown in Fig. 1, the reduced precision imposes great challenges for maintaining the training accuracy. With our proposed training framework, taking the speed advantage of `int8` precision became feasible.

Table 7 shows the speedup of NITI's $\boldsymbol{a}, \boldsymbol{e}, \boldsymbol{g}$ computation under different convolution input size conditions on two different NVIDIA GPUs while compared with their equivalent floating-point implementations. For $\boldsymbol{a}$ and $\boldsymbol{e}$ computations, the use of `int8` matrix multiply provided up to $6.76\times$ and $8.57\times$ speedup on the 2080ti GPU respectively. The performance of both `int8` and `fp32` operations improved with the newer 3090 GPU, with the gap between `fp32` and `int8` reduced, likely due to the highly optimized routine employed in the `fp32` implementations.

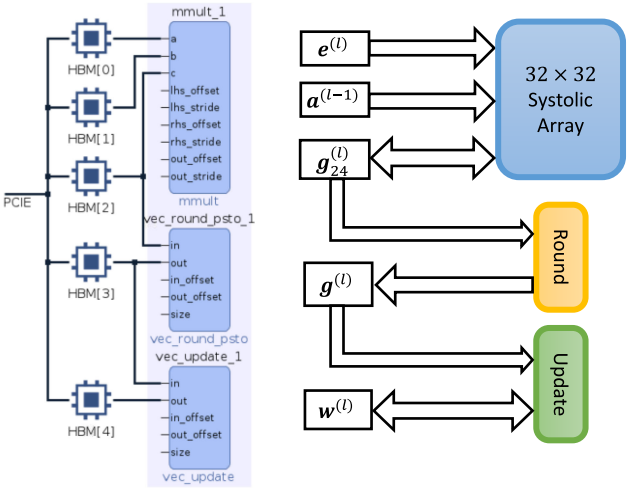
As for $\boldsymbol{g}$ computation, the speedup of `int8` over `fp32` is less significant. $\boldsymbol{g}$ was computed by convolving the rearranged $\boldsymbol{a}^{(l-1)}$ with the rearranged $\boldsymbol{e}^{(l)}$. The kernel (rearranged $\boldsymbol{e}^{(l)}$) used in this convolution was very different from normal inference convolutions. In the cases shown in Table 7, its size varied from 224×224 to 14×14. Currently, we only optimized our implementation to handle small convolution kernels used in mainstream CNNs, like 3×3, 5×5 and 7×7. Different kernel sizes require different strategies of mapping to matrix multiplication tiles of tensor cores. We plan to optimize this part in the future.

The last column of Table 6 summarizes the GPU memory footprint of different training experiments. The evaluations were conducted using PyTorch 1.8.0 and CUDA 11.1. The training batch sizes used on MNIST, CIFAR10, and ImageNet were 256, 256, and 512, respectively. Comparing with highly optimized `fp32` baseline, our preliminary GPU implementation of NITI reduced the end-to-end training memory footprint by up to $1.31\times, 1.84\times, 1.16\times$ on MNIST, CIFAR10 and ImageNet, respectively. [11] consumed more GPU memory during training, as the low precision arithmetic were emulated through `fp32` operations. The memory footprint of [1], [6] were not reported in Table 6 because related training codes are not available. We expect [1], [6] will consumed more GPU memory than `fp32` baseline due to their emulation-based implementation like [11].

An open-source implementation of NITI can be found in [41], which includes an implementation of native `int8` backward pass using tensor cores that can be integrated to the PyTorch training framework as a CUDA extension.

5.2 FPGA Acceleration

To evaluate the hardware benefits of using `int8` operations in NITI, an FPGA accelerator prototype was implemented



(a) (b)

Fig. 7. (a)preliminary FPGA prototype of training a fully connected layer (b)the data flow of computing g and updating w on the prototype.

using an Xilinx Alveo U280 FPGA card with the Xilinx Vitis HLS tools.

Fig. 7a shows the top-level block diagram of the design. The current design implements the 3 main operations of NITI, namely matrix multiplication, rounding, and weight update, needed to perform back propagation of a layer. The layer weights, input, gradients and intermediate matrix multiplication accumulations during the training process are stored in the on-chip High Bandwidth Memory (HBM).

Fig. 7b shows the data flow of computing g and updating w on the hardware. The data flow of computing a and e are similar and are omitted for brevity. To effectively evaluate the hardware efficiency of each individual module when different arithmetic operations are employed, data are passed between different modules through the HBM.

Matrix multiplications form the core computational kernel of NITI. Here, a 32×32 output-stationary systolic array was implemented in the hardware, which performs the operations for 1 tile of the overall tiled matrix multiplications that compute a, e and g. Synthesis results with different data types are shown in Table 8. Considering the core systolic array for matrix multiplication shown in the first 2 columns of the Table 8, when compared to the equivalent `fp32` implementation, the `int8` implementation consumes $5\times$ fewer DSP blocks and is able to operate at $46\,\%$ higher clock speed. It also incurs a kernel computation latency that is $12.1\times$ lower.

TABLE 8
Synthesis Comparison of `fp32` and `int8`

Module	Systolic Array		Update	
Precision	fp32	int8	fp32	int8
Latency	448 cycles	37 cycles	10 cycles	3 cycles
Frequency	411.0 MHz	601.3 MHz	427.0MHz	520.3 MHz
LUT	299,091	18,531	13,280	7,424
FF	609,469	24,656	16,032	2,592
DSP	5120 (56%)	1024 (11%)	0	0
BRAM	6	6	0	0

TABLE 9
Synthesis Comparison of Different Rounding Schemes

Rounding	Nearest	Stochasic	Pseudo
Latency	3 cycles	3 cycles	3 cycles
Clock	520.3MHz	520.3MHz	520.3MHz
LUT	3,328	4,928	3,680
FF	608	1,728	640

No DSP or BRAM Usage.

As mentioned in section 3.4, the use of a proper rounding scheme is crucial to maintain training accuracy. Here we evaluated the hardware cost of implementing different rounding schemes. Table 9 compares the hardware cost of running 32 rounding operators in parallel using different schemes. Although our proposed pseudo stochastic rounding schemes consumes more LUT and FF resources than the simple round to nearest scheme, it consumes $63.0\,\%$ fewer FFs and $25.3\,\%$ fewer LUT than the commonly used stochastic rounding scheme.

Finally, although simple in design, the weight update module similarly illustrates the benefit of `int8` operations. As shown in Table 8, updates of weight is 3-cycle operation in `int8`. On the other hand, due to the large trip time for the floating point operator, the `fp32` update modules requires $1.79\times$ more LUT and $2.86\times$ FF, while achieving a lower clock speed and $3.3\times$ more cycles latency.

6 Conclusion

In this work, we demonstrated the benefits of training deep neural networks using integer arithmetic exclusively in NITI. The innovations of this work include: our update scheme that utilizes low precision integer operations for all intermediate values; a new stochastic rounding scheme that uses discarded bits as a random number source, obviating the need for an additional random number generator; and an efficient integer-only cross-entropy loss backpropagation scheme that is suitable for use in other integer DNN training frameworks. By using 8-bit integer arithmetic operations exclusively, we demonstrated that DNN training can be accelerated with existing low bitwidth integer operators in modern GPU originally designed for inference acceleration. Preliminary study in the design of an FPGA based training accelerator also shows significant hardware advantages of `int8` implementations over `fp32` in terms of resource consumptions, memory storage requirements, as well as computation latency. Looking into the future, our work lays the foundation for an integer-only accelerator which could greatly reduce cost, chip area and energy consumption required for DNN training.

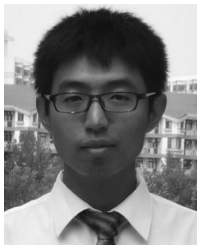
References

[1] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," in *Proc. 6th Int. Conf. Learn. Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=HJGXzmspb

[2] Y. Yang, L. Deng, S. Wu, T. Yan, Y. Xie, and G. Li, "Training high-performance and large-scale deep neural networks with full 8-bit integers," *Neural Netw.*, vol. 125, pp. 70–82, 2020.

[3] P. Micikevicius *et al.*, "Mixed precision training," in *Proc. 6th Int. Conf. Learn. Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=r1gs9JgRZ

[4] N. P. Jouppi *et al.*, "A domain-specific supercomputer for training deep neural networks," *Commun. ACM*, vol. 63, no. 7, pp. 67–78, Jun. 2020.
[5] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Proc. Adv. Neural Informat. Process. Syst.*, 2018, pp. 7675–7684.
[6] X. Chen, X. Hu, H. Zhou, and N. Xu, "FxpNet: Training a deep convolutional neural network in fixed-point representation," in *Proc. Int. Joint Conf. Neural Netw.*, 2017, pp. 2494–2501.
[7] S. Fox, S. Rasoulinezhad, J. Faraone, D. Boland, and P. Leong, "A block minifloat representation for training deep neural networks," in *Proc. Int. Conf. Learn. Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=6zaTwpNSsQ2
[8] J. Lu, C. Fang, M. Xu, J. Lin, and Z. Wang, "Evaluations on deep neural networks training using posit number system," *IEEE Trans. Comput.*, vol. 70, no. 2, pp. 174–187, Feb. 2021.
[9] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H.-J. Yoo, "LNPU: A 25.3TFLOPS/W sparse deep-neural-network learning processor with fine-grained mixed precision of FP8-FP16," in *Proc. IEEE Int. Solid- State Circuits Conf.*, 2019, pp. 142–144.
[10] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv , and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6869–6898, 2017.
[11] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," in *Proc. Adv. Neural Informat. Process. Syst.*, 2018, pp. 5145–5153. [Online]. Available: http://papers.nips.cc/paper/7761-scalable-methods-for-8-bit-training-of-neural-networks.pdf
[12] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015.
[13] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," in *Proc. 5th Int. Conf. Learn. Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=S1_pAu9xl
[14] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.*, 2016, pp. 525–542.
[15] M. Courbariaux and Y. Bengio, "BinaryNet: Training deep neural networks with weights and activations constrained to +1 or -1," 2016, *arXiv:1602.02830*.
[16] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 2704–2713.
[17] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," 2016, *arXiv:1606.06160*.
[18] D. Das *et al.*, "Mixed precision training of convolutional neural networks using integer operations," in *Proc. 6th Int. Conf. Learn. Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=H135uzZ0-
[19] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 1737–1746.
[20] P. Yao *et al.*, "Fully hardware-implemented memristor convolutional neural network," *Nature*, vol. 577, no. 7792, pp. 641–646, 2020.
[21] C. Li *et al.*, "Efficient and self-adaptive in-situ learning in multilayer memristor neural networks," *Nature Commun.*, vol. 9, no. 1, pp. 1–8, 2018.
[22] Y. Du, S. Yang, and K. Huang, "High-dimensional stochastic gradient quantization for communication-efficient edge learning," *IEEE Trans. Signal Process.*, vol. 68, pp. 2128–2142, 2020.
[23] M. M. Amiri and D. Gündüz, "Machine learning at the wireless edge: Distributed stochastic gradient descent over-the-air," *IEEE Trans. Signal Process.*, vol. 68, pp. 2155–2169, 2020.
[24] J. Johnson, "Rethinking floating point for deep learning," 2018, *arXiv:1811.01721*.
[25] X. Sun *et al.*, "Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks," in *Proc. 33rd Int. Conf. Neural Inf. Process. Syst.*, 2019, Art. no. 441.
[26] L. Cambier, A. Bhiwandiwalla, T. Gong, M. Nekuii, O. H. Elibol, and H. Tang, "Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks," in *Proc. 8th Int. Conf. Learn. Representations*, 2020.
[27] NVIDIA, "Turing Architecture," 2019. [Online]. Available: https://www.nvidia.com/en-us/geforce/turing
[28] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," in *Proc. IEEE Int. Conf. Neural Netw.*, 1993, pp. 586–591.
[29] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Univ. Toronto, Tech. Rep., 2009.
[30] Y. LeCun *et al.*, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[31] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei , "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.
[32] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 1026–1034.
[33] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proc. 13th Int. Conf. Artif. Intell. Statist.*, 2010, pp. 249–256.
[34] H. Yang, S. Gui, Y. Zhu, and J. Liu, "Automatic neural network compression by sparsity-quantization joint learning: A constrained optimization-based approach," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 2178–2188.
[35] G. Srivastava, D. Kadetotad, S. Yin, V. Berisha, C. Chakrabarti, and J.-S. Seo, "Joint optimization of quantization and structured sparsity for compressed deep neural networks," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, 2019, pp. 1393–1397.
[36] Y. Lecun *et al.*, *Learning Algorithms for Classification: A Comparison on Handwritten Digit Recognition*. Singapore: World Scientific, 1995, pp. 261–276.
[37] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "NVIDIA A100 tensor core GPU: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, Mar./Apr. 2021.
[38] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Informat. Process. Syst.*, 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
[39] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: http://tensorflow.org
[40] S. Chetlur *et al.*, "cuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.
[41] M. Wang, NITI source code, 2021. [Online]. Available: https://github.com/wangmaolin/niti



Maolin Wang received the BEng degree in electrical engineering from Tsinghua University in 2015, and the PhD degree from the Department of Electrical and Electronic Engineering, The University of Hong Kong in 2020. His research interests include efficient hardware architectures and algorithms for the training and inference of deep neural networks.

Seyedramin Rasoulinezhad received the BS degree in electrical engineering and digital systems from the Sharif University of Technology, Iran. He is currently working toward the PhD degree researching on computer architectures with The University of Sydney, Australia, under the supervision of Prof. Philip Leong and Dr. David Boland. His research interests include new FPGA architectures for the implementation of a variety of applications while considering machine learning algorithms as high demand future applications.

Philip H. W. Leong (Senior Member, IEEE) received the BSc, BE, and PhD degrees from the University of Sydney. In 1993, he was a consultant to ST Microelectronics, Milan, Italy, working on advanced flash memory-based integrated circuit design. From 1997 to 2009, he was with the Chinese University of Hong Kong. He is currently a professor of computer systems with the School of Electrical and Information Engineering, University of Sydney, a visiting professor with Imperial College, and the chief technology advisor to Cluster Technology. He is the author of more than 150 technical papers and four patents. He was the co-founder and program co-chair of the International Conference on Field Programmable Technology (FPT), program co-chair of the International Conference on Field Programmable Logic and Applications (FPL), and is the senior associate editor of the *ACM Transactions on Reconfigurable Technology and Systems*. He was the recipient of 2005 FPT conference Best Paper and also the 2007 and 2008 FPL conference Stamatis Vassiliadis Outstanding Paper awards.

Hayden K.-H. So (Senior Member, IEEE) received the BS, MS, and PhD degrees in electrical engineering and computer sciences from the University of California, Berkeley, CA, USA, in 1998, 2000, and 2007, respectively. He is currently an Associate professor and the co-director of Computer Engineering Program with the Department of Electrical and Electronic Engineering, and the co-director of the Joint Lab on Future Cities with the University of Hong Kong. His research interests include highly-efficient reconfigurable computing systems and their applications. He was the recipient of the Test-of-time Award CODES+ISSS, in 2021, Croucher Innovation Award, in 2013, IEEE-HKN C. Holmes MacDonald Outstanding Teaching Award, in 2021, University Outstanding Teaching Award (Team), in 2012, and Faculty Best Teacher Award, in 2011. He was the technical program chair for various international conferences, including the International Symposium on Applied Reconfigurable Computing (ARC), International Conference on Field-Programmable Technology (FPT), International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART), and IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP).