上海电力大学

微电子专业英语大作业

题　　目:基于 FPGA 的尖峰神经网络硬件实现

设计（论文）题目

院　　系：电信学院

专业年级：

学生姓名：　　　　学号：

指导教师：宋小军

2023 年 12 月 24 日

基于 FPGA 的尖峰神经网络硬件实现设计（论文）题目

摘　要

受真实生物神经模型的启发，尖峰神经网络 (SNN) 使用离散尖峰处理信息，并显示出构建低功耗神经网络系统的巨大潜力。本文提出了一种基于现场可编程门阵列（FPGA）的 SNN 硬件实现方案。它采用混合更新算法，结合了现有算法的优点，简化了硬件设计并提高了性能。所提出的设计支持多达 16 384 个神经元和 1680 万个突触，但需要最少的硬件资源，并且实现 0.477 W 的极低功耗。使用 Xilinx FPGA 评估板基于所提出的设计构建了一个测试平台，我们在该平台上进行了部署 MNIST 数据集上的分类任务。评估结果显示准确率为 97.06%，帧率为 161 帧/秒。

关键词：微电子专业英语大作业

第一章　介绍

近年来，神经网络（NN）已成功部署在广泛的应用中。与使用模拟值表示网络内部激活的传统人工神经网络 (ANN) 相比，尖峰神经网络 (SNN) 模仿真实的生物神经元，并使用神经元尖峰的时序信息对激活进行编码。与 ANN 不同，ANN 的主要运算是权重的矩阵乘法和网络层的激活，SNN 的尖峰特性避免了复杂的矩阵乘法，因此它们需要的计算资源更少，并且能源效率更高。

SNN 由神经元构建，神经元与加权突触连接以形成层和网络。作为 SNN 的基本构建块，每个尖峰神经元都维护一个代表其当前状态的值。这些神经元的输入是通过突触传输的尖峰。神经元收集这些输入尖峰并整合相应的权重以更新其内部状态。每当状态达到某个阈值时，神经元就会被激活并向其后继神经元输出一个尖峰。经过预先配置的传播延迟后，后继者会收到尖峰并相应地更新其状态。除了这些共同的操作原理之外，不同的神经元模型还具有不同的神经元行为。最常用的神经元模型之一是泄漏积分与激发 (LIF) 模型。LIF 神经元受到生物神经元中泄漏的膜电压的启发，如果不存在输入尖峰，它们的状态会随着时间的推移而降低。

然而，当前的计算机架构并不非常适合执行 SNN。SNN 固有的大规模并行性（其中大量神经元以相似但简单的方式工作）需要比当前中央处理单元 (CPU) 提供的并行架构更加并行的架构。尽管图形处理单元 (GPU) 可以利用 SNN 的并行性，但当前 GPU 的内核启动编程范式使得它们不适合事件驱动计算，而事件驱动计算是 SNN 更新算法的一类重要内容（如下所述）第 4.1 节）。然而，现场可编程门阵列（FPGA）可以解决这个问题，因为它们提供并行处理能力，并且可以灵活地重新配置以满足不同的计算模型。此外，FPGA 比当前的 CPU 和 GPU 更加节能。

因此，本文提出了一种基于 FPGA 的 SNN 模块实现。通过利用增强的混合更新算法，该模块以最少的片上资源和 0.477 W 功耗支持多达 16 384 个神经元和 1680 万个突触。我们构建了一个测试平台来评估所提出的加速器，在该平台上我们映射了一个 SNN 模型，用于对 MNIST 数据集中的手写数字进行分类。评估结果显示，分类准确率为 97.06%，处理神经元放电事件的性能为每秒 6.72×10^5 事件，导致 MNIST 数据集每秒处理 161 帧。

第二章　背景

在 SNNs 的硬件实现中，使用了各种神经元模型，包括 Izhikevich 模型 2 和 Integrate-andFire (IF) 模型的变体。我们的方法采用了 LIF 神经元模型，它用指数过程模拟了生物神经元的膜电压泄漏：

$$V(t_2) = V(t_1) \cdot e^{-(t_2 - t_1)/\tau}$$

其中 $V(t)$ 是神经元在时间 t 的状态（生物神经元的膜电压），τ 是泄漏常数。对于时间步进的更新算法，t 的值是离散的，而 $t_2 - t_1$ 的项是一个常数 Δt。对于事件驱动的算法，t_1 是上一个更新这个神经元的事件的时间戳，t_2 是当前事件的时间戳。

神经元的状态也会受到其他神经元的输入电流的影响。一个权重 $W_{i,j}$ 被分配给从神经元 i 到神经元 j 的突触。当神经元 i 产生一个输出脉冲时，$W_{i,j}$ 在一个固定的延迟后被加到神经元 j 的状态上。因此，对于时间步进的更新算法，神经元 j 从时间步 t_n 到 t_{n+1} 的更新方程为

$$V(t_n + 1) = V(t_n) \cdot e^{-\Delta t/\tau} + \sum_{i=0}^{m-1} s_i(t_n) \cdot W_{i,j}$$

其中 m 是神经元 j 的输入数，$s_i(t_n)$ 是神经元 i 在时间步 t_n 的发放状态。如果神经元 i 输出一个脉冲，发放状态 s_i 为 1，否则为 0。对于事件驱动的更新算法，一个更新是由神经元 i 的激活事件触发的。对于每个神经元 j，其中神经元 j 是神经元 i 的后继，更新是通过

$$V(t') = V(t) \cdot e^{-(t'-t)/\tau} + W_{i,j}$$

来进行的，其中的参数和方程 (1) 中的含义相同。

在更新神经元状态之后，一个阈值 V_{th} 被用来决定一个神经元是否被激活并输出一个脉冲。在一个神经元激活之后，它的状态被重置为 V_{reset}。

第三章　相关工作

3.1　硬件实现

人们在 SNN 的硬件实现上投入了大量精力，并且一些设计以数字、模拟和混合模拟/数字电路的形式呈现。对于数字实现，基于 FPGA 的系统和专用集成电路 (ASIC) 系统都已得到广泛研究。

TrueNorth 芯片是最著名的 ASIC 设计之一。TrueNorth 系统的核心包含一个 256×256 的交叉开关，它实现突触的功能，并被配置为将传入的尖峰映射到神经元。通过集成 4096 个此类处理核心，TrueNorth 芯片可承载 100 万个神经元和 2.56 亿个突触。通过将多个芯片连接在一起，可以进一步扩展规模。SpiNNaker 是另一个完全定制的数字系统，由许多小型 ARM 处理器组成。它具有定制的互连通信方案，该方案被设计为适合大量小的尖峰状消息，从而针对基于尖峰的网络架构的通信行为进行了优化。与 TrueNorth 一样，SpiNNaker 支持多个芯片级联形成大规模系统。

之前的工作也提出了几种基于 FPGA 的 SNN 加速器设计。BlueHive 采用 multiFPGA 架构，支持多达 65536 个神经元和 6710 万个突触。然而，它使用复杂的 Izhikevich 模型和低放电率假设来实现神经元，其目标是生物神经网络模拟，不能有效支持现实世界的应用。另一个代表性设计是 Minituar，它将神经元的激活视为事件，并利用事件驱动的算法来更新它们。它实现了多达 65536 个 LIF 神经元和 1680 万个突触。Minituar 忠实地模拟了神经元的指数泄漏过程，并采用片上数字信号处理器 (DSP) 来执行定点计算。它还维护一个硬件事件队列，需要对每个传入事件进行排序操作，以支持带有延迟的尖峰；这增加了设计复杂性和运行时延迟。

3.2　网络模型

近年来，深度置信网络（DBN）已被证明在机器视觉和机器试听等多个领域中有效。DBN 是一种多层概率生成模型，使用多个受限玻尔兹曼机 (RBM) 的堆叠结构。之前的工作提出了将 DBN 转换为基于 LIF 的尖峰 DBN 的方法，并探索了事件驱动算法对尖峰 DBN 的处理。

另一项研究试图解决从全连接网络（FCN）和卷积神经网络（CNN）到 SNN 转换时出现的精度损失。所提出的优化技术包括在训练期间使用零偏差的整流线性单元 (ReLU) 以适应尖峰编码、帮助调节发射率的权重归一化方法以及实现低延迟处理的阈值平衡方案。

在本文中，我们使用参考文献中提出的技术训练我们的 SNN 模型并探索此类 SNN 模型的高效硬件实现。

第四章　系统设计

4.1　混合更新算法

我们使用的更新算法是传统的时间步进更新算法和事件驱动更新算法的混合。在本小节中，我们首先简要描述这两种现有算法，然后介绍我们的混合算法。

时间步长算法根据离散时间步长处理所有神经元。在每个时间步内，每个神经元的状态都会被更新和检查，以确定它是否输出尖峰。有关这些尖峰的信息被存储起来，以便根据它们的传输延迟在未来的时间步中使用。该算法可能会浪费计算资源，因为它为不接收任何输入尖峰的神经元安排不必要的操作。另一方面，事件驱动算法仅处理神经元的激活事件。事件队列用于存储事件，并按事件时间戳排序。每个事件从事件队列中出队后，仅更新连续神经元的状态，从而生成新事件。这样就避免了不必要的操作。尽管事件驱动算法可能很高效，但事件队列的硬件实现很复杂，因为每当有新事件入队时就需要对事件进行排序。

因此，我们结合了时间步进和事件驱动算法，如算法 1 中所述。我们使用多个事件队列，每个事件队列都标有时间戳 Q_n（其中 n 范围从 0 到 $D-1$，其中 D 是允许的最大延迟），存储从当前时间开始 n 个时间步后要处理的事件。这样，具有相同时间戳的事件就可以存储在同一个队列中，而无需进行排序操作。为了管理这些事件队列，需要全局维护时间步长。在每个时间步，带有标记 Q_0 的事件队列被设置为活动队列，并处理其事件。一旦事件队列为空，当前时间步就结束。在下一个时间步之前，所有事件队列的标签减一，使得 $Q_1 - Q_{D-1}$ 变为 $Q_0 - Q_{D-2}$ 并且 Q_0 以循环方式重用：Q_{D-1} 。这种混合更新算法避免了排序操作，从而减少了系统的运行时延迟。

4.2　系统架构

所提出模块的架构如图 1 所示。（翻译版已省略）

有四个主要内存组件，每个组件都有自己的控制器来管理其读写操作。事件队列子模块是上面 4.1 节中描述的多个事件队列的硬件实现。事件控制器子模块负责通过将生成的事件入队和将事件出队进行处理来管理这些事件队列。权重存储器和状态存储器子模块分别用于存储权重和状态数据。权重存储子模块是只读的，而状态控制器还控制来自状态更新器的更新状态的写回。另一个内存子模块是延迟内存，它是只读的，存储不同事件的延迟值。有关存储器组件实现的详细信息将在下面第 4.3 节中讨论。

状态更新器执行计算的主体。它首先衰减神经元状态，然后对输入权重求和以更新神经元状态。然后执行神经元激活检查以确定是否生成新事件。如果任何神经元被激活，其神经元状态将重置为预定常数。状态更新器可以通过同时更新多个神经元状态来利用 SNN 的并行性。SNN 的分层结构保证了神经元的后继者彼此独立，

这使得同步更新成为可能。

执行流程如下。事件控制器从系统控制器（为了清楚起见，在图 1 中省略了）接收控制信号和当前时间步长的值。然后，它将当前事件队列设置为活动状态，并依次从中读取事件。事件数据被发送到权重控制器和状态控制器。他们使用事件数据访问权重和状态存储器，然后将它们发送到状态更新器。

如果状态更新后有激活事件，这些事件将通过延迟控制器查找它们的延迟。事件控制器根据延迟计算出对应的目标事件队列，并将事件写入其中。

系统以异步方式工作以提高吞吐量。例如，事件控制器向权重控制器和状态控制器发送事件数据后，立即开始读取事件队列。当它从重量和状态控制器收集数据请求信号时，事件数据被再次发送。其他子模块之间的通信类似，通过请求和响应。另一个例子是状态更新器的源控制器（为了清楚起见，图 1 中也省略了）。它包含两个先进先出（FIFO），用于保存来自权重控制器和状态控制器的状态更新器的操作数。这样，虽然后两个控制器具有不同的存储器访问时间，但只要 FIFO 仍有空间容纳传入数据，就可以避免它们之间的等待。

4.3　执行

我们使用带符号的 16 位定点数来表示权重和神经元状态。神经元的最大数量设置为 16 384，这导致每个神经元的索引为 14 位。一层最大神经元数量为 1024 个，支持全连接突触。这意味着突触的最大数量为 1680 万个。最大延迟设置为 16，根据之前的研究这已经足够了。

建议的模块需要存储最多 32 MB 的重量数据。由于 FPGA 的片上 Block RAM (BRAM) 的数量通常是有限的，因此将所有权重存储在片上是不切实际的。因此，我们使用外部双倍数据速率 (DDR) 存储器来存储所有权重，同时使用 BRAM 实现所有其他存储器模块。考虑到每个事件对应的权重始终属于同一组，我们将这些权重存储在外部存储器的连续空间中。通过这种映射，可以利用 DDR 存储器的突发读取功能来优化存储器访问的延迟。

对于事件队列，我们使用 BRAM 实现 16 个 FIFO，可以使用 4 位地址单独访问。为了实现混合更新算法，事件控制器总是读取 FIFO 中当前时间寄存器的低四位。状态更新器生成新事件并从延迟控制器获取延迟后，将延迟值与当前时间相加，结果的低四位用于选择要写入数据的正确 FIFO 。

权重更新器的操作主要是权重和神经元状态的相加以及更新后的状态与阈值参数的比较。神经元的衰减也是由权重更新器执行的。为了简单起见，指数计算是通过减去一个常数来实现的。为了充分利用存储器访问带宽，权重更新器中实例化了 32 个加法器和 32 个比较器。下面第 5.4 节中的进一步分析表明，权重更新器的吞吐量及其对硬件资源的消耗都不是所提出模块的限制因素。

第五章　评估

5.1　实验装置

基准. 我们将具有两个隐藏层的前馈 SNN 应用于 MNIST 手写数字数据集的分类任务，以评估所提出的模块。网络拓扑如图 2 所示。输入层包括 784 个神经元，用于处理从 28×28 像素数字转换而来的输入尖峰。两个隐藏层都有 1024 个神经元，而输出层有 10 个神经元，对应数字 0-9 的分类结果。这些层是完全连接的，这意味着两个相邻层之间的连接是全对全的。

本文重点讨论 SNN 的硬件实现，并使用基准测试来衡量系统性能。尽管模型的深度（即层数）有限，但它尽可能地占用了每层的扇入/扇出。所提出的模块根据事件更新神经元状态，每个事件的更新操作和内存访问取决于扇入/扇出。因此，正如下面 5.4 节分析的那样，用这个基准测试测得的系统性能接近理论上限。下面第 5.3 节中报告的分类精度也证明了所提出模块的功能正确性。

测试平台. 我们使用带有 XC7Z045 SoC 的 Xilinx ZC706 评估板来构建测试平台。该评估板提供 1 GB DDR3 内存和两个 ARM Cortex-A9 MPCore。测试平台的结构如图 3 所示。我们在评估板上的可编程逻辑中实现了所提出的模块。该模块的运行频率为 200 MHz。其中一个 ARM 处理器用于配置所提出模块的内部寄存器并管理数据移动。如上所述，重量数据存储在 DDR3 存储器中。这三个部分通过片上高级可扩展接口（AXI）总线连接，如图 3 所示。

5.2　硬件利用率

硬件利用率结果是根据所提出的加速器的综合结果获得的，如表 1 所示。根据表 1 中报告的结果，最密集的片上资源是 BRAM，其中神经元状态和激活事件队列已实施。这意味着管理 BRAM 是基于 FPGA 的 SNN 加速系统的一个重要设计考虑因素。潜在的优化技术，例如压缩事件的表示或减少状态和/或事件的位宽，可能是有益的。

总功耗为 0.477 W，详细细分如图 4 所示。静态功耗为 0.246 W，接近总功耗的 52%。BRAM 占 19%，因此也是系统功耗的主要组成部分。

表 5-1 ZC706 开发板的硬件利用率。

Component	Cells used	Utilization(%)
LUT	5381	2.46
FF	7309	1.67
BRAM	40.5	7.43
BUFG	1	3.13

5.3 准确性

分类精度在 MNIST 测试数据集上进行测试，该数据集由 10 000 帧图形组成。该模型通过 MATLAB 的权重和阈值平衡方案进行训练，使用 32 位浮点数。训练结果准确率为 98.48%。训练后的权重被转换为 16 位定点数并部署在建议的模块上。车载测试准确率为 97.06%。因此，精度损失了 1.42%，这主要是由浮点数到 16 位定点数的转换造成的。

5.4 系统性能

我们根据基准数据集的总体吞吐量（以每秒帧数为单位）和每秒处理的激活事件数来评估所提出的模块的性能。为了测量处理时间，我们在测试平台中插入了一个硬件计数器。当 ARM 处理器完成 DDR3 内存和建议模块的初始配置时，它开始计数，并在最终帧获得其分类结果时停止计数。10 000 帧的处理时间为 62.1 秒，提供每秒 161 帧的帧速率。

对于激活事件的数量，我们使用 MATLAB 来模拟所提出模块的定点计算。使用此模拟器，激活事件的数量为 4.2×10^7 。根据上述获取的处理时间，处理激活事件的性能为每秒 6.72×10^5 个事件。

为了使所提出的模块处理神经元的每个激活事件，需要更新其所有连续神经元的状态；这需要从 DDR3 内存中获取 1024 个权重值。

鉴于所提出的模块运行在 200 MHz 并且数据总线宽度为 64 位，带宽可以高达 1600 MB/s（权重在分类期间是只读的并且仅考虑一个方向）。为了处理每个神经元激活事件，需要其所有连续神经元的权重。由于我们为每个权重使用 16 位，并且每个神经元的最大扇出为 1024，因此数量应为 2 KB。结合这两个约束，系统的最大性能为每秒 8×10^5 个事件。

将实际性能与理论值进行比较，带宽在所提出的设计中得到了很好的利用。这些系统性能结果进一步表明外部存储器带宽是基于 FPGA 的 SNN 实现的瓶颈。减少权重位宽和网络修剪等技术可以缓解这个问题；特别是，网络剪枝已被证明在人工神经网络中是有效的。

5.5 与 GPU 的比较

为了进行比较，相同的 SNN 模型是使用 PyTorch 框架实现的，并在 NVIDIA Tesla P100 GPU 上运行。请注意，GPU 不适合事件驱动的更新，因此我们改为实现时间步进更新算法。我们再次测量 MNIST 测试数据集 10000 帧的执行时间。GPU 执行期间的运行时功率通过 NVIDIA 系统管理接口进行测量。

GPU 上的处理时间为 7.96 秒，平均功耗为 29.6 W。因此，虽然所提出的模块需要更多的执行时间，但它的功耗比 GPU 少得多。这相当于更高的能效：每瓦每秒 337.6 帧，而 GPU 上每瓦每秒 42.2 帧。这些结果意味着所提出的设计适用于对功耗

和执行延迟有严格限制的应用场景。

第六章　讨论

正如上文第 5 节所述，所提议的 SNN 实现还有进一步优化的机会。由于外部存储器带宽对于系统性能至关重要，因此我们在本节中简要讨论降低权重位宽和修剪网络连接的影响。我们将这两种技术应用于第 5 节中评估的 MNIST 模型，并使用我们的软件模拟器重新评估分类准确性。

图 5 显示了较低位宽的分类精度。图 5 所示的结果表明，即使权重位宽降低到 6，位宽对精度的影响也可以忽略不计。这使得外部数据请求的总大小减少了 62.5%。尽管以如此低的位宽进行操作的能力部分源于 MNIST 数据集的简单性，但减少位宽是提高系统性能的一种有前途的方法。因此，对于其他系统的实现和基准测试，应仔细考虑位宽的选择。

类似地，网络剪枝的影响如图 6 所示。较小值的权重设置为 0，并使用稀疏参数作为阈值。稀疏度为 50% 时，可以修剪一半的权重，同时分类精度的损失最小。通过权重修剪，可以减少处理事件所需的权重，从而有利于性能。我们使用简单的剪枝来掩盖较小的权重；通过进一步优化，例如微调，可以进一步减少精度损失。

此外，这两种优化技术彼此正交并且可以同时应用。这需要进一步探索设计空间，留待未来研究。

第七章　结论

本文提出了一种基于 FPGA 的 SNN 硬件实现方案。所提出的模块是基于时间步进和事件驱动更新算法的混合设计的使用 MNIST 数据集对所提出的模块进行评估，结果显示分类准确度为 97.06%。在 0.477 W 功耗下，处理神经元激活事件的性能为每秒 $6.72 * 10^5$ 个事件，这导致 MNIST 数据集上的帧率为每秒 161 帧。所提出的模块进一步确定内存带宽是系统的瓶颈。为了解决这个问题，讨论了两种用于基于 FPGA 的 SNN 实现的潜在优化技术。

TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 04/10 pp479–486
DOI: 10.26599/TST.2019.9010019
Volume 25, Number 4, August 2020

Hardware Implementation of Spiking Neural Networks on FPGA

Jianhui Han, Zhaolin Li*, Weimin Zheng, and Youhui Zhang*

Abstract: Inspired by real biological neural models, Spiking Neural Networks (SNNs) process information with discrete spikes and show great potential for building low-power neural network systems. This paper proposes a hardware implementation of SNN based on Field-Programmable Gate Arrays (FPGA). It features a hybrid updating algorithm, which combines the advantages of existing algorithms to simplify hardware design and improve performance. The proposed design supports up to 16 384 neurons and 16.8 million synapses but requires minimal hardware resources and archieves a very low power consumption of 0.477 W. A test platform is built based on the proposed design using a Xilinx FPGA evaluation board, upon which we deploy a classification task on the MNIST dataset. The evaluation results show an accuracy of 97.06% and a frame rate of 161 frames per second.

Key words: Spiking Neural Network (SNN); Field-Programmable Gate Arrays (FPGA); digital circuit; low-power; MNIST

1 Introduction

Over recent years, Neural Networks (NNs) have been successfully deployed in a wide range of applications. Compared to conventional Artificial Neural Networks (ANNs) which use analog values to represent activations inside the networks, Spiking Neural Networks (SNNs) imitate real biological neurons and encode the activation with the timing information of neuron spikes. As opposed to ANNs, where the major operations are the matrix multiplication of weights and activation of network layers, the spiking nature of SNNs avoids complex matrix multiplications and thus they require lower computation resources and are more energy efficient.

SNNs are built with neurons, which are connected with weighted synapses to form layers and networks. As the basic building block of SNNs, each spiking neuron maintains a value to represent its current state. Inputs to these neurons are spikes transmitted through synapses. The neurons collect these input spikes and integrate the corresponding weights to update their internal states. Whenever the state reaches a certain threshold value, the neuron is activated and outputs a spike to its successor neurons. After a pre-configured propagation delay, the successors receive the spike and update their states accordingly. Besides these common operating principles, different neuron models feature different neuron behaviors. One of the most commonly used neuron models is the Leaky Integrate-and-Fire (LIF) model. LIF neurons are inspired by the membrane voltage leaking in biological neurons, with their states decreasing over time if no input spikes present.

However, current computer architectures are not ideally suited to executing SNNs. The massive parallelism inherent to an SNN, in which a large number of neurons work in a similar but simple manner, requires a more parallel architecture than that provided by current Central Processing Units (CPUs). Although Graphics Processing Units (GPUs) can capitalize on

• Jianhui Han is with the Institute of Microelectronics, Tsinghua University, Beijing 100084, China. E-mail: hanjh16@mails.tsinghua.edu.cn.
• Zhaolin Li is with the Research Institute of Information Technology, Tsinghua University, Beijing 100084, China. E-mail: lzl73@mail.tsinghua.edu.cn.
• Weimin Zheng and Youhui Zhang are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: zwm-dcs@mail.tsinghua.edu.cn; zyh02@tsinghua.edu.cn.
∗ To whom correspondence should be addressed.
Manuscript received: 2019-03-26; revised: 2019-04-25; accepted: 2019-05-05

the parallelism of SNN, the kernel-launch programming paradigm of current GPUs makes them ill-suited to event-driven computation, which is an important class of updating algorithms for SNNs (as described below in Section 4.1). However, Field-Programmable Gate Arrays (FPGA) can address this issue since they provide parallel processing capability and are flexibly reconfigurable to cater to different computing models. Furthermore, FPGA are more energy efficient than current CPUs and GPUs.

This paper therefore proposes an FPGA-based SNN module implementation. By utilizing an enhanced hybrid updating algorithm, the proposed module supports up to 16 384 neurons and 16.8 million synapses with minimal on-chip resources and 0.477 W power consumption. We build a test platform to evaluate the proposed accelerator, on which we map an SNN model for the classification of handwritten digits in the MNIST[1] dataset. The evaluation results show that the classification accuracy is 97.06% and the performance for processing neuron firing events is 6.72×10^5 events per second, resulting in 161 frames per second for the MNIST dataset.

2 Background

A wide variety of neuron models have been used in hardware implementations of SNNs, including the Izhikevich model[2] and variations of the Integrate-and-Fire (IF) model. Our approach adopts the LIF neuron model, which simulates the membrane voltage leaking of biological neurons with an exponential process:

$$V(t_2) = V(t_1)\cdot \mathrm{e}^{-(t_2-t_1)/\tau} \tag{1}$$

where $V(t)$ are the neuron states at time t (the membrane voltage in biological neurons), and τ is the leaky constant. For the time-stepped updating algorithm, the values of t are discrete and the term $t_2 - t_1$ will be a constant Δt. For the event-driven algorithm, t_1 is the timestamp of the previous event that updated this neuron, and t_2 is the timestamp of the current event.

The neuron state is also changed by input current from other neurons. A weight $W_{i,j}$ is assigned to the synapse from neuron i to neuron j. When neuron i generates an output spike, $W_{i,j}$ is added to the state of neuron j after a fixed delay. Therefore, for the time-stepped updating algorithm, the updating equation of neuron j from time step t_n to t_{n+1} is

$$V(t_{n+1}) = V(t_n)\cdot \mathrm{e}^{-\Delta t/\tau} + \sum_{i=0}^{m-1} s_i(t_n)\cdot W_{i,j} \tag{2}$$

where m is the fan-in of neuron j, and $s_i(t_n)$ is the spiking status of neuron i at time step t_n. The spiking status s_i is 1 if neuron i outputs a spike, otherwise it is 0. For the event-driven updating algorithm, an update is triggered by the activation event of neuron i. For each neuron j, where neuron j is a successor of neuron i, the update is carried out by

$$V(t') = V(t)\cdot \mathrm{e}^{-(t'-t)/\tau} + W_{i,j} \tag{3}$$

where the parameters have the same meaning as in Eq. (1).

After updating the neuron states, a threshold value V_{th} is used to decide whether a neuron is activated and outputs a spike. After a neuron activates, its state is reset to V_{reset}.

3 Related Work

3.1 Hardware implementation

Significant effort has been invested into the hardware implementation of SNNs and some designs have been presented in the form of digital[3–7], analog[8, 9], and mixed analog/digital circuits[10, 11]. For digital implementations, both FPGA-based systems and Application-Specific Integrated Circuit (ASIC) systems have been widely studied.

The TrueNorth chip[3] is one of the most well-known ASIC designs. A core in the TrueNorth system contains a 256×256 crossbar that implements the function of synapses and is configured to map incoming spikes to neurons. By integrating 4096 such processing cores, the TrueNorth chip carries 1 million neurons and 256 million synapses. The scale can be further extended by connecting multiple chips together. SpiNNaker[4] is another fully custom digital system and is composed of many small ARM processors. It features a custom interconnect communication scheme that is designed to be suitable for a large number of small spike-like messages and thus optimized for the communication behavior of a spike-based network architecture. Like TrueNorth, SpiNNaker supports the cascading of multiple chips to form large-scale systems.

Previous works[5–7] have also proposed several FPGA-based SNN accelerator designs. BlueHive[5] supports up to 65 536 neurons and 67.1 million synapses with a multi-FPGA architecture. However, it implements the neurons with the complex Izhikevich model[2] and a low firing rate assumption, which targets biological neural network simulations and does not support real-world applications effectively. Another representative design is Minituar[6], which treats the activation of neurons as events and utilizes an event-driven algorithm to update them. It implements up to 65 536 LIF neurons and 16.8 million synapses. Minituar faithfully models the exponential leaky process of neurons and employs on-chip Digital Signal Processors (DSPs) to carry out the fixed-point computation. It also maintains a hardware event queue that requires a sorting operation for each incoming event to support spikes with delays; this increases design complexity and run-time latency.

3.2 Network model

In recent years, Deep Belief Networks (DBNs)[12] have been proven to be effective in a variety of domains, such as machine vision[13] and machine audition[14]. DBN is a multilayered probabilistic generative model that uses a stacked structure of multiple Restricted Boltzmann Machines (RBMs). Previous work[15] has proposed methods to convert DBNs to LIF-based spiking DBNs and explored the processing of spiking DBNs with the event-driven algorithm.

Another study[16] tried to solve the loss of accuracy arising in the conversion from Fully-Connected Networks (FCNs) and Convolutional Neural Networks (CNNs) to SNNs. The proposed optimization techniques include using Rectified Linear Units (ReLUs) with zero bias during training to suit spiking encoding, a weight normalization method to help regulate firing rates, and a threshold balancing scheme to enable low-latency processing.

In this paper, we use the techniques proposed in Ref. [16] to train our SNN model and explore the efficient hardware implementation of such SNN models.

4 System Design

4.1 Hybrid updating algorithm

We use an updating algorithm that is a hybrid of the conventional time-stepped updating algorithm and the event-driven updating algorithm. In this subsection, we first briefly describe these two existing algorithms and then present our hybrid.

The time-stepped algorithm processes all of the neurons based on discrete time steps. Within each time step, the state of each neuron is updated and checked to decide whether it outputs a spike. Information about these spikes is stored for use in future time steps according to their transmission delay. This algorithm can waste computing resources since it schedules unnecessary operations for neurons that do not receive any input spikes. The event-driven algorithm, on the other hand, processes only the activation events of neurons. An event queue is used as storage for the events, and is sorted by the event timestamps. After each event dequeues from the event queue, only the states of successive neurons are updated, thereby generating new events. In this way, unnecessary operations are avoided. Although the event-driven algorithm can be efficient, the hardware implementation of the event queue is complicated since it requires sorting the events whenever a new event enqueues.

Therefore, we combine the time-stepped and event-driven algorithm, as described in Algorithm 1. We use multiple event queues, each of which is tagged with timestamp Q_n (where n ranges from 0 to $D-1$ where D is the maximum delay allowed), to store the events to be processed after n time steps from the current time. In this way, events with the same timestamp can be stored in the same queue with no sorting operation required. To manage these event queues, time steps are maintained globally. At each time step, the event queue with tag Q_0 is set to be the active queue and its events are processed. Once an event queue is empty, the current time step finishes. Before the next time step, the tags of all event queues decrease by one, such that Q_1–Q_{D-1} becomes Q_0–Q_{D-2} and Q_0 is reused in a circular manner as Q_{D-1}. Sorting operations are avoided in this hybrid updating algorithm, which reduces the system's run-time latency.

4.2 System architecture

The architecture of the proposed module is shown in

Algorithm 1 Hybrid updating algorithm

```
Input: Event queues $Q_0$, $Q_1$, ..., $Q_{D-1}$
1:  for $t \Leftarrow 0 : \Delta t : T$ do
2:      while not $Q_0$.is_empty() do
3:          $event \Leftarrow Q_0$.dequeue()
4:          for $neuron$ in $event$.successors() do
5:              $neuron$.update_state()
6:              $neuron$.check_activation()
7:              if $neuron$.is_activated() then
8:                  $new\_event \Leftarrow neuron$.form_new_event()
9:                  $delay \Leftarrow new\_event$.get_delay()
10:                 $Q_{delay}$.enqueue($new\_event$)
11:             end if
12:         end for
13:     end while
14:     for $i \Leftarrow 1 : D-1$ do
15:         $Q_{i-1} \Leftarrow Q_i$
16:     end for
17:     $Q_{D-1} \Leftarrow Q_0$
18: end for
```

Fig. 1. There are four main memory components, each of which has its own controller to manage its reading and writing operations. The event queues submodule is the hardware implementation of the multiple event queues described above in Section 4.1. The event controller submodule is in charge of managing these event queues by enqueuing generated events and dequeuing events for processing. The weight memory and state memory submodules are used to store weight and state data, respectively. The weight memory submodule is read-only, while the state controller also controls the writing back of updated states from the state updater. Another memory submodule is the delay memory, which is read-only and stores the delay values of different events. Details about the implementation of the memory components are discussed below in Section 4.3.

The state updater carries out the main body of computation. It first decays the neuron states and then sums the incoming weights to update the neuron states. Checks for neuron activation are then carried out to decide whether a new event is generated. If any neuron is activated, its neuron state is reset to a predetermined constant. The state updater can exploit the parallelism of SNN by updating multiple neuron states at the same time. The layered structure of SNN ensures that the successors of a neuron are independent of each other, which makes simultaneous updating possible.

The execution flow is as follows. The event controller receives controlling signals and values of the current time step from the system controller (which is omitted from Fig. 1 for clarity). It then sets the current event queue to be active and sequentially reads events from it. The event data is sent to the weight controller and the state controller. They access the weight and state memory with the event data and then send them to the state updater. If there are activation events after the state update, these events go through the delay controller to look up their delays. The event controller calculates the corresponding destination event queues according to the delay, and writes the events to them.

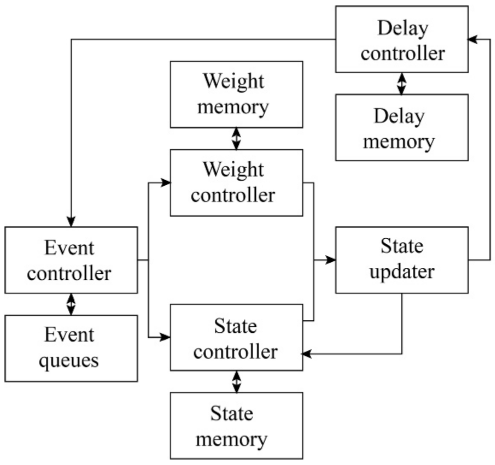


Fig. 1 Architecture of the proposed system.

The system works in an asynchronous manner to improve throughput. For example, after the event controller sends event data to the weight and state controllers, it begins to read the event queue immediately. When it collects the data request signal from the weight and state controllers, event data are sent again. Communication between other submodules is similar, through requests and responses. Another example is the *source controller* (also omitted from Fig. 1 for clarity) for the state updater. It contains two First-In-First-Outs (FIFOs) to hold the operands of the state updater from the weight controller and the state controller. In this way, although the latter two controllers have different memory access times, waiting between them is avoided, provided that the FIFOs still have space for incoming data.

4.3 Implementation

We use signed 16-bit fixed-point numbers to represent the weights and neuron states. The maximum number of neurons is set to 16 384, which results in a 14-bit index for each neuron. The maximum number of neurons in one layer is 1024, with fully-connected synapses supported. This means that the maximum number of synapses is 16.8 million. The maximum delay is set to 16, which is adequate according to previous research[6].

The proposed module needs to store up to 32 MB of weight data. Since the amount of on-chip Block RAM (BRAM) of FPGA is often limited, it is impractical to store all of the weights on-chip. Therefore, we use external Double Data Rate (DDR) memory to store all of the weights, while implementing all of the other memory modules with BRAM. Considering that for each event the corresponding weights are always of the same group, we store these weights in consecutive spaces in the external memory. With this mapping, the burst read feature of the DDR memory can be exploited to optimize the latency of memory access.

For the event queues, we implement 16 FIFOs with BRAM that can be accessed separately with a 4-bit address. To implement the hybrid updating algorithm, the event controller always reads the FIFO with the lower four bits of the current time register. After a new event is generated by the state updater and the delay is obtained from the delay controller, the delay value is added to the current time and the lower four bits of the result are used to select the correct FIFO to which to write the data.

The operations of the weight updater are mainly the addition of weights and neuron states and the comparison of updated states with the threshold parameter. The decay of neurons is also carried out by the weight updater. For simplicity, the exponential computation is implemented with the subtraction of a constant. To fully utilize the memory access bandwidth, 32 adders and 32 comparators are instantiated in the weight updater. Further analysis in Section 5.4 below shows that neither the throughput of the weight updater nor its consumption of hardware resources is limiting factors of the proposed module.

5 Evaluation

5.1 Experimental setup

Benchmark. We apply a feed-forward SNN with two hidden layers to a classification task on the MNIST handwritten digit dataset[1] to evaluate the proposed module. The topology of the network is shown in Fig. 2. The input layer includes 784 neurons to process the input spikes converted from the 28×28 pixel digit figure. The two hidden layers both have 1024 neurons, while the output layer has 10 neurons corresponding to the classification results of digits 0–9. The layers are fully connected, which means the connections are all-to-all between two adjacent layers.

This paper focuses on the hardware implementation of SNN, and the benchmark is used for system performance measurement. Although the model has a limited depth (i.e., number of layers), it takes up the fan-in/fan-out of each layer as much as possible. The proposed module updates neuron states based on events and the updating operations and memory accesses of each event are

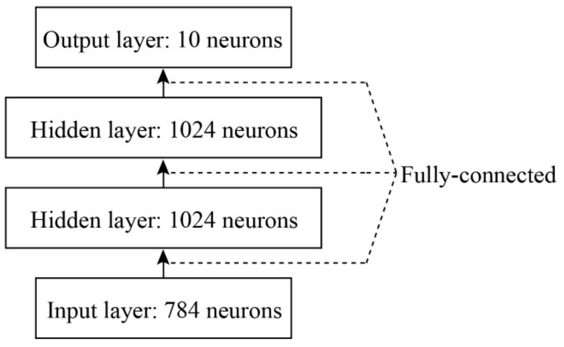


Fig. 2 Topology of the benchmark SNN model.

dependent on the fan-in/fan-out. Therefore, as analyzed in Section 5.4 below, the system performance measured with this benchmark is close to the theoretical upper limit. The functional correctness of the proposed module is also testified by the classification accuracy reporeted in Section 5.3 below.

Test platform. We use a Xilinx ZC706 evaluation board[17] with a XC7Z045 SoC to build the test platform. The evaluation board provides 1 GB DDR3 memory and two ARM Cortex-A9 MPCores. The structure of the test platform is illustrated in Fig. 3. We implement the proposed module in the programmable logic on the evaluation board. The module runs at 200 MHz. One of the ARM processors is employed to configure the internal registers of the proposed module and manage the data movement. The weight data are stored in the DDR3 memory as described above. These three parts are connected through an on-chip Advanced eXtensible Interface (AXI) bus, as shown in Fig. 3.

5.2 Hardware utilization

The hardware utilization results are obtained based on the synthesis results of the proposed accelerator, as listed in Table 1. According to the results reported in Table 1, the most intensive on-chip resource is BRAM, with which the neuron states and activation event queues are implemented. This implies that managing BRAM is an important design consideration for FPGA-based SNN

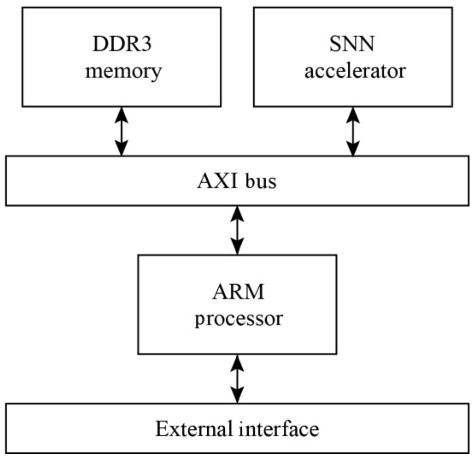


Fig. 3 Structure of the test platform.

Table 1 Hardware utilization for ZC706 board.

Component	Cells used	Utilization (%)
LUT	5381	2.46
FF	7309	1.67
BRAM	40.5	7.43
BUFG	1	3.13

acceleration systems. Potential optimization techniques, such as compressing the presentation of events or reducing the bit-width of states and/or events, could be beneficial.

The total power consumption is 0.477 W, with the detailed breakdown shown in Fig. 4. Static power is 0.246 W, which is nearly 52% of the total power consumption. BRAM accounts for 19%, and is thus also a major component of system power use.

5.3 Accuracy

The classification accuracy is tested on the MNIST test dataset, which consists of 10 000 frames of figures. The model is trained with the weight and threshold balancing scheme[16] with MATLAB, using 32-bit floating-point numbers. The training results in an accuracy of 98.48%. The trained weights are turned into 16-bit fixed-point numbers and deployed on the proposed module. The accuracy of the on-board test is 97.06%. There is therefore an accuracy loss of 1.42%, which is mainly caused by the conversion from floating-point to 16-bit fixed-point numbers.

5.4 System performance

We evaluate the performance of the proposed module with the overall throughput on the benchmark dataset (in frames per second) and the number of activation events processed per second. To measure the processing time, we insert a hardware counter into the test platform. It begins counting when the ARM processor finishes the initial configuration of the DDR3 memory and the proposed module, and stops counting when the final frame obtains its classification result. The processing time for the 10 000 frames is 62.1 s, which supplies a frame rate of 161 frames per second.

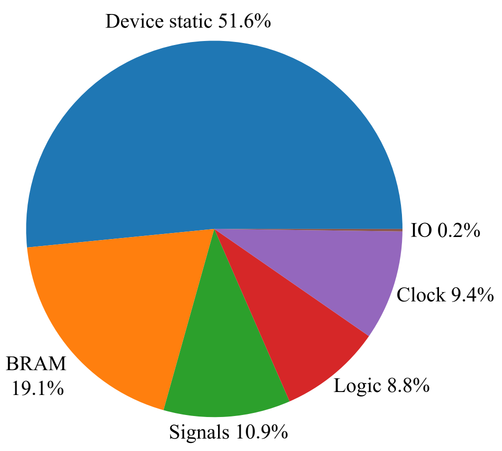


Fig. 4 Power consumption breakdown.

For the number of activation events, we use MATLAB to simulate the fixed-point computation of the proposed module. With this simulator, the number of activation events is 4.2×10^7. With the processing time acquired as mentioned above, the performance in processing activation events is 6.72×10^5 events per second.

For the proposed module to process each activation event of a neuron, the states of all its successive neurons need to be updated; this requires fetching 1024 weight values from DDR3 memory.

Given that the proposed module runs at 200 MHz and the width of the data bus is 64 bits, the bandwidth can be up to 1600 MB/s (the weights are read-only during the classification and only one direction is considered). To process each neuron activation event, the weights for all its successive neurons are required. Since we use 16 bits for each weight and the maximum fan-out of each neuron is 1024, the amount should be 2 KB. Combining these two constraints, the maximum performance of the system is 8×10^5 events per second.

Comparing the practical performance with this theoretical value, the bandwidth has been well exploited in the proposed design. These system performance results further identify that external memory bandwidth is the bottleneck of FPGA-based SNN implementations. Techniques like reducing the bit-width of weights and network pruning can alleviate this issue; in particular, network pruning has been demonstrated to be effective in ANNs[18].

5.5 Comparison with GPU

For comparison, the same SNN model is implemented with the PyTorch[18] framework to run on an NVIDIA Tesla P100 GPU[19]. Note that GPUs are not suitable for event-driven updating, so we implement the time-stepped updating algorithm instead. Again, we measure the execution time of the 10 000 frames of the MNIST test dataset. The runtime power during GPU execution is measured with the NVIDIA System Management Interface[20].

The processing time on a GPU is 7.96 s and the average power consumption is 29.6 W. Therefore, although the proposed module demands more execution time, it consumes much less power than a GPU. This equates to much greater power efficiency: 337.6 frames per second per watt compared to 42.2 frames per second per watt on a GPU. These results imply that the proposed design is suitable for application scenarios that have strict constraints on power consumption and a tolerance for execution latency.

6 Discussion

As mentioned above in Section 5, there are further optimization opportunities for the proposed SNN implementation. Since external memory bandwidth is crucial for the system's performance, we briefly discuss the impact of lowering the weight bit-widths and pruning network connections in this section. We apply these two techniques to the MNIST model evaluated in Section 5 and re-evaluate the classification accuracy with our software simulator.

Figure 5 shows the classification accuracy with lower bit-widths. The results shown in Fig. 5 demonstrate that the influence of bit-width on accuracy is negligible even with the weight bit-width lowered to 6. This gives a 62.5% reduction in the total size of external data requests. Although the ability to operate with such a low bit-width partially stems from the simplicity of the MNIST dataset, reducing bit-width is a promising method of improving system performance. Therefore, for the implementation and benchmarking of other systems, the choice of bit-width should be carefully considered.

Similarly, the impact of network pruning is illustrated in Fig. 6. Weights with small values are set to 0, with a sparsity parameter used for the threshold. With sparsity at 50%, half of the weights can be pruned with minimal loss of classification accuracy. By weight pruning, the weights that are needed to process an event are reduced, which benefits performance. We use a naive pruning that simply masks out small weights; with further optimization such as fine-tuning[21], the loss of accuracy can be reduced further.

Furthermore, these two optimization techniques are orthogonal to each other and can be applied simultaneously. This requires further exploration of the design space and is left to a future study.

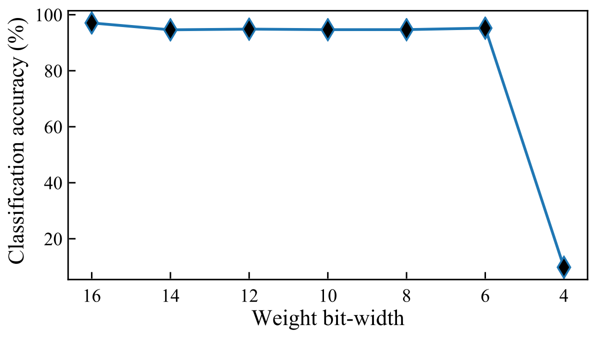


Fig. 5 Classification accuracy vs. bit-widths on MNIST.

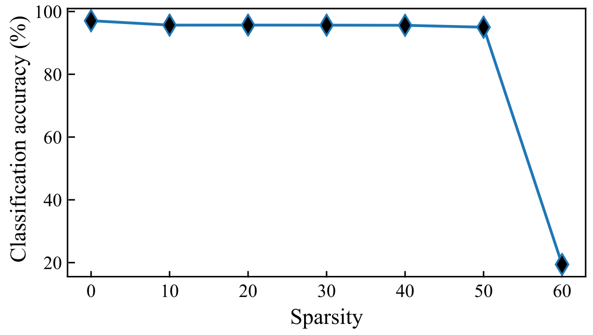


Fig. 6 Classification accuracy vs. sparsity on MNIST.

7 Conclusion

In this paper, an FPGA-based SNN hardware implementation is proposed. The proposed module is designed based on a hybrid of the time-stepped and event-driven updating algorithms. An evaluation of the proposed module is carried out using the MNIST dataset with the results showing a classification accuracy of 97.06%. With 0.477 W power consumption, the performance for processing neuron activation events is 6.72×10^5 events per second, which results in a frame rate of 161 frames per second on MNIST dataset. The proposed module further identifies that memory bandwidth is the bottleneck of the system. To address this issue, two potential optimization techniques are discussed for FPGA-based SNN implementations.

Acknowledgment

This work was supported in part by the Beijing Innovation Center for Future Chip, Tsinghua University, in part by the Science and Technology Innovation Special Zone project, China, and in part by the Tsinghua University Initiative Scientific Research Program (No. 2018Z05JDX005).

References

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, Gradient-based learning applied to document recognition, *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] E. M. Izhikevich, Simple model of spiking neurons, *IEEE Trans. Neural Netw.*, vol. 14, no. 6, pp. 1569–1572, 2003.

[3] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, et al., A million spiking-neuron integrated circuit with a scalable communication network and interface, *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[4] M. M. Khan, D. R. Lester, L. A. Plana, A. Rast, X. Jin, E. Painkras, and S. B. Furber, SpiNNaker: Mapping neural networks onto a massively-parallel chip multiprocessor, in *2008 IEEE Int. Joint Conf. on Neural Networks (IEEE World Congress on Computational Intelligence)*, Hong Kong, China, 2008, pp. 2849–2856.

[5] S. W. Moore, P. J. Fox, S. J. T. Marsh, A. T. Markettos, and

A. Mujumdar, Bluehive—A field-programable custom computing machine for extreme-scale real-time neural network simulation, in *2012 IEEE 20th Int. Symp. on Field-Programmable Custom Computing Machines*, Toronto, Canada, 2012, pp. 133–140.

[6] D. Neil and S. C. Liu, Minitaur, an event-driven FPGA-based spiking network accelerator, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 12, pp. 2621–2628, 2014.

[7] K. Cheung, S. R. Schultz, and W. Luk, A large-scale spiking neural network accelerator for FPGA systems, in *Artificial Neural Networks and Machine Learning – ICANN 2012*, A. E. P. Villa, W. Duch, P. Érdi, F. Masulli, and G. Palm, eds. Springer, 2012, pp. 113–120.

[8] E. Farquhar, C. Gordon, and P. Hasler, A field programmable neural array, in *2006 IEEE Int. Symp. on Circuits and Systems*, Island of Kos, Greece, 2006, pp. 4114–4117.

[9] M. Liu, H. Yu, and W. Wang, FPAA based on integration of CMOS and nanojunction devices for neuromorphic applications, in *Int. Conf. on Nano-Networks*, M. Cheng, ed. Springer, 2009, pp. 44–48.

[10] B. V. Benjamin, P. R. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J. M. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen, Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations, *Proc. IEEE*, vol. 102, no. 5, pp. 699–716, 2014.

[11] T. Pfeil, J. Jordan, T. Tetzlaff, A. Grübl, J. Schemmel, M. Diesmann, and K. Meier, Effect of heterogeneity on decorrelation mechanisms in spiking neural networks: A neuromorphic-hardware study, *Phys. Rev. X*, vol. 6, no. 2, p. 021023, 2016.

[12] G. E. Hinton and R. R. Salakhutdinov, Reducing the dimensionality of data with neural networks, *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[13] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, Deep, big, simple neural nets for handwritten digit recognition, *Neural Comput.*, vol. 22, no. 12, pp. 3207–3220, 2010.

[14] A. R. Mohamed, G. E. Dahl, and G. Hinton, Acoustic modeling using deep belief networks, *IEEE Trans. Audio Speech Lang. Process.*, vol. 20, no. 1, pp. 14–22, 2012.

[15] P. O'Connor, D. Neil, S. C. Liu, T. Delbruck, and M. Pfeiffer, Real-time classification and sensor fusion with a spiking deep belief network, *Front. Neurosci.*, vol. 7, p. 178, 2013.

[16] P. U. Diehl, D. Neil, J. Binas, M. Cook, S. C. Liu, and M. Pfeiffer, Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing, in *2015 Int. Joint Conf. on Neural Networks (IJCNN)*, Killarney, Ireland, 2015, pp. 1–8.

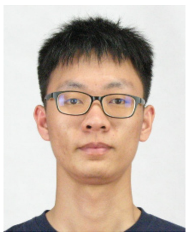
[17] Xilinx Inc., Xilinx Zynq-7000 SoC ZC706 evaluation kit, https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html, 2019.

[18] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. M. Lin, A. Desmaison, L. Antiga, and A. Lerer, Automatic differentiation in PyTorch, in *Proc. 31st Conf. on Neural Information Processing Systems*, Long Beach, CA, USA, 2017, pp. 1–4.

[19] NVIDIA Corporation, NVIDIA Tesla P100: The world's first AI supercomputing data center GPU, https://www.nvidia.com/en-us/data-center/tesla-p100/, 2019.

[20] NVIDIA Corporation, NVIDIA system management interface, https://developer.nvidia.com/nvidia-system-management-interface, 2019.

[21] S. Han, H. Z. Mao, and W. J. Dally, Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, arXiv preprint: 1510.00149, 2015.



Jianhui Han received the BS degree from Tsinghua University, Beijing, China, in 2016. He is currently working toward the PhD degree at the Institute of Microelectronics, Tsinghua University, Beijing, China. His main research interests include digital circuit/system design and emerging technology-based machine learning acceleration.

Zhaolin Li received the BS and PhD degrees from Harbin Institute of Technology, Harbin, China, in 1994 and 2000, respectively. He is currently a professor with the Research Institute of Information Technology, Tsinghua University, Beijing, China. His current research interests include embedded systems, parallel computing, multicore design, and system-on-a-chip.

Weimin Zheng received the MS degree from Tsinghua University, Beijing, China. Currently he is an Academician of Chinese Academy of Engineering and a professor at the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include high performance computing, network storage, and parallel compiler.

Youhui Zhang received the BS and PhD degrees from Tsinghua University, Beijing, China, in 1998 and 2002, respectively. He is currently a professor in the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include computer architecture and neuromorphic computing. He is a member of CCF, ACM, and IEEE.