doi:10.11887/j.cn.202401013 http://journal.nudt.edu.cn

长向量处理器高效RNN推理方法

苏华友[1,2],陈抗抗[1,2],杨乾明[1*]

(1. 国防科技大学 计算机学院,湖南 长沙 410073;
2. 国防科技大学 并行与分布计算全国重点实验室,湖南 长沙 410073)

摘 要:模型深度的不断增加和处理序列长度的不一致对循环神经网络在不同处理器上的性能优化提出巨大挑战。针对自主研制的长向量处理器FT-M7032,实现了一个高效的循环神经网络加速引擎。该引擎采用行优先矩阵向量乘算法和数据感知的多核并行方式,提高矩阵向量乘的计算效率;采用两级内核融合优化方法降低临时数据传输的开销;采用手写汇编优化多种算子,进一步挖掘长向量处理器的性能潜力。实验表明,长向量处理器循环神经网络推理引擎可获得较高性能,相较于多核ARM CPU以及Intel Golden CPU,类循环神经网络模型长短记忆网络可获得最高62.68倍和3.12倍的性能加速。

关键词:多核DSP;长向量处理器;循环神经网络;并行优化
中图分类号:TP391 **文献标志码**:A **开放科学(资源服务)标识码(OSID)**:
文章编号:1001-2486(2024)01-121-10

Efficient RNN inference engine on very long vector processor

SU Huayou[1,2], *CHEN Kangkang*[1,2], *YANG Qianming*[1*]

(1. College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China;
2. National Key Laboratory of Parallel and Distributed Computing, National University of Defense Technology, Changsha 410073, China)

Abstract: With the increasing depth and the inconsistent length of processing sequences, the performance optimization of RNN (recurrent neural network) on different processors makes it difficult to researchers. An efficient RNN acceleration engine was implemented for the self-developed long vector processor FT-M7032. This engine proposed a row-first matrix vector multiplication algorithm and a data-aware multi-core parallel method to improve the computational efficiency of matrix vector multiplication. It proposed a two-level kernel fusion optimization method to reduce the overhead of temporary data transmission. Optimized handwritten assembly codes for multiple operators were integrated to further tap the performance potential of long vector processors. Experiments show that the RNN engine for long-vector processors is efficient, when compared with the multi-core ARM CPU and Intel Golden CPU, the RNN-like model long short term memory networks can achieve a performance acceleration of up to 62.68 times and 3.12 times, respectively.

Keywords: multicore DSP; very long vector processor; recurrent neural networks; parallel optimization

传统的卷积神经网络(convolutional neural networks,CNN)在视觉领域取得了巨大的成功,例如图像分类、目标检测等。但是对于语言类时序数据任务,某些时刻前面的输入和后面的输入是有关系的,CNN这种只能处理独立输入的模型就无法获得较好的效果。循环神经网络(recurrent neural network, RNN)是面向时序数据处理的一类重要的深度神经网络,它以序列数据为输入,并对每一个时刻的输入结合当前模型的状态给出一个输出。目前,以RNN、长短记忆网络[1](long short term memory networks, LSTM)、门控循环单元[2](gate recurrent unit, GRU)为代表的时序数据处理类循环神经网络已广泛应用于文件分类[3]、语音识别[4-5]和机器翻译[6]等自然语言相关的任务处理中。随着自然语言处理任务的复杂度增加,计算量更大的Transformer结构和BERT模型得到广泛的应用,对计算资源的需求和计算效率的提升都提出很大的挑战。

随着模型规模的扩大和模型层次的增加,类循环神经网络的训练和推理过程对目标平台的执行效率提出了极大的挑战。如果沿用通用深度学习框架,诸如PyTorch[7]、TensorFlow[8]等直接按

收稿日期:2022-11-07
基金项目:国家自然科学基金资助项目(61872377);湘江实验室基金资助项目(22XJ01012)
第一作者:苏华友(1985—),男,广西桂林人,副研究员,博士,硕士生导师,E-mail:shyou@nudt.edu.cn
***通信作者**:杨乾明(1984—),男,湖南邵阳人,助理研究员,博士,E-mail:yqm21249@nudt.edu.cn

照类循环神经网络的计算过程进行碎片式、拼接化执行,硬件的利用率得不到保障。例如,在图形处理器(graphics processing unit, GPU)上直接调用各个算子函数进行端到端的拼接执行,LSTM 的执行效率还不能达到峰值性能的 10%。当前有很多的工作针对 RNN 在 GPU 上的执行过程进行了优化,例如 cuDNN[9] 深度学习库。该工作通过算子级的优化、层级融合等多种技术手段,大大提高了 RNN 计算在 GPU 上的效率。随着物联网技术的发展,越来越多的设备接入网络中,例如中央处理器(central processing unit, CPU)和数字信号处理器(digital signal processor,DSP)等,这类处理器的数量远远多于 GPU,并且离数据源更近。然而,当前针对 CPU 或者以长向量为重点计算单元的 DSP 的类循环神经网络计算优化工作相对较少,只能借助于基础的矩阵乘等算子进行功能实现,以 LSTM 等为代表的类循环神经网络在 DSP 等长向量处理器上的性能优化空间还很大,难度也很高。

首先,RNN 处理对象的序列长度可变性导致计算不均衡。对于 RNN 处理的序列数据而言,不同语句的长度往往不一致,会导致描述语句的输入向量长度不一致。在大 batch_size 主导的推理应用模式下,所有的时序都会以当前 batch 中序列长度最大的样本为参照,进行样本数据的填充以保证输入的底层计算规模相同,从而不会影响循环的深度,同时使得分批处理执行变得简单,但是这种填充的方法会造成大量的额外的计算需求。其次,序列长度的可变性导致内存管理的不确定性。序列长度的不同会使得内存的分配和片上存储分配动态变化,否则,所有序列都需要按照最长序列的需求进行存储分配。另外,由于基于长向量处理器的智能算子库的缺失,需要重新构建完整的算子,需要对矩阵向量乘和矩阵乘进行优化,同时 RNN 及其变种还存在大量的边缘算子,这些算子的性能也会影响整体的效率,并且随着参数的增大,RNN 等模型中核心矩阵规模增大,矩阵计算效率的提高会放大边缘算子计算的耗时占比。

针对上述问题,本文面向国产自主长向量处理器 FT-M7032,研究如何高效实现 RNN 等类循环神经网络推理任务的并行优化实现。首先,为了克服序列长度可变的问题,提出了一种行优先的矩阵向量乘方法,通过改变原有的计算顺序,有效地利用长向量处理器的计算单元和开发数据的局域性,最大化提高计算效率。然后,建立了一个手工汇编实现的高效的 RNN 算子库,对矩阵向量乘算子的底层实现进行了深度优化,包括单核内的指令排布、多核任务划分以及片上存储空间的替换策略等。最后,高效实现了 RNN 等类循环神经网络模型的边缘算子,并且基于有效利用片上存储的原则对边缘算子和矩阵向量乘算子进行了融合。通过上述工作,可以有效地利用自主长向量处理器对循环神经网络算法进行加速。

1 背景介绍

1.1 RNN 介绍

1.1.1 模型结构介绍

RNN 常用于解决输入样本为连续序列且序列的长短不一的问题,单层 RNN 神经网络的结构如图 1 所示。

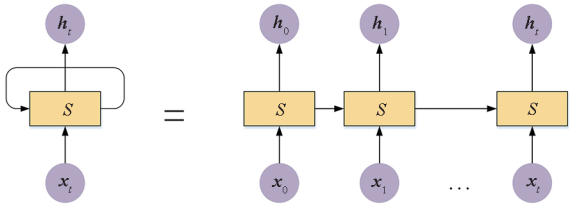


图 1 单层 RNN 神经网络的结构

Fig. 1 Structure of a single-layer RNN neural network

RNN 每读取一个新的输入 $\boldsymbol{x}_t$,就会生成状态向量 $\boldsymbol{h}_t$ 作为当前时刻的输出和下一时刻的输入,将 T 个输入 $\boldsymbol{x}_0 \sim \boldsymbol{x}_t$ 依次输入 RNN,相应地会产生 T 个输出,其中 S 表示神经元。

1.1.2 计算原理

典型的 RNN 神经网络的计算原理如图 2 所示,主要由两条单向流动的信息流处理组成。一条信息流从输入单元到达隐藏单元,另一条信息流从隐藏单元到达输出单元。假设 $\boldsymbol{x}$ 为输入层,一般为向量;$\boldsymbol{O}$ 为输出层,$\boldsymbol{h}$ 为隐含层,而 t 指计算次数;$\boldsymbol{W}_i$、$\boldsymbol{W}_h$ 为权重,一般为矩阵。其中计算第 t 次的输出结果如式(1)所示。

$$\boldsymbol{O}_t = f(\boldsymbol{x}_t \times \boldsymbol{W}_i + \boldsymbol{h}_{t-1} \times \boldsymbol{W}_h) \tag{1}$$

从式(1)可以看出,循环神经网络的主要计

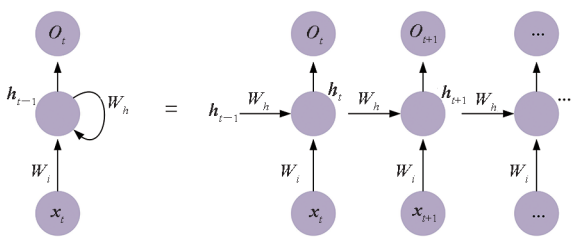


图 2 RNN 神经网络的计算原理

Fig. 2 Calculation principle of RNN neural network

算模式是矩阵向量乘,当以 batch 模式执行的时候则是矩阵乘。除此之外,还涉及 tanh、sigmoid 等边缘计算过程。诸如 LSTM 和 GRU 等 RNN 变种模型的计算模式也主要以矩阵计算为主,区别在于 LSTM 相对于标准的 RNN 网络涉及的信息流更多,即矩阵计算模块更多。

1.2　FT-M7032 长向量处理器

FT-M7032 是一款国防科技大学自主研制的长向量处理器,采用异构结构,集成了 1 个 16 核 ARMv8 CPU 和 4 个通用数字信号处理器(general purpose digital signal processor, GPDSP)。多核 CPU 主要负责线程管理和通信,其单精度浮点峰值性能为 281.6 GFLOPS。每个 GPDSP 集群包括 8 个 DSP 核心,它们共享 6 MB 片上全局共享内存(global shared memory, GSM)。每个计算簇中的 8 个DSP 核和 GSM 通过片上网络进行通信,通信带宽可达 300 Gbit/s。GSM 是软件可管理的片上存储器,需要由开发人员维护 GSM 和 DSP 之间的数据一致性。CPU 和每个 GPDSP 计算簇以共享内存的方式进行数据交互。CPU 可以访问整个内存空间,但每个 GPDSP 计算簇只能访问其对于内存通道的存储空间,理论访存带宽为 42.6 Gbit/s。

GPDSP 中的每个 DSP 核心基于超长指令字(very long instruction word, VLIW)架构,包括指令调度单元(instruction fetch unit, IFU)、标量处理单元(scale processing unit, SPU)、向量处理单元(vector processing unit, VPU)和直接内存访问(direct memory access, DMA)引擎,如图 3 所示。IFU 被设计为每个周期最多启动 11 条指令,其中包含 5 条标量指令和 6 条向量指令。SPU 用于指令流控制和标量计算,主要由标量处理元素(scale processing element, SPE)和 64 KB 标量内存(scale memory, SM)组成,它们匹配 5 条标量指令。VPU 是长向量处理单位,为每个 DSP 核心提供主要计算性能,包括 768 KB 阵列存储器(array memory, AM)和以单指令多数据(single instruction multiple datastream, SIMD)方式工作的 16 个矢量处理元件(vector processing element, VPE)。每个 VPE 有 64 个 64 位寄存器和 3 个融合乘加(float multiply accumulate, FMAC)单元,1 个 FMAC 每周期可以处理 2 个 FP32 或者 4 个 FP16 的乘加计算。当工作在 1.8 GHz 时,每个 DSP 核可以提供的单精度浮点峰值性能为345.6 GFLOPS,半精度浮点峰值性能为691.2 GFLOPS。AM 可以通过 2 个向量 load/store 单元,在每个周期向寄存器传输 512 个字节。SPU 和 VPU 之间通过广播指令和共享寄存器传输数据。DMA 引擎用于在不同级别的存储器(即主存储器、GSM 和 SM/AM)之间进行块数据传输。

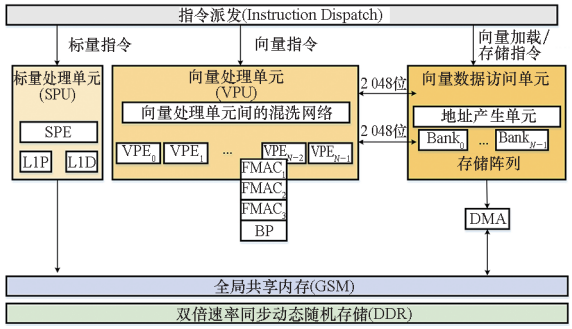


图 3　FT-M7032 长向量处理器体系结构

Fig. 3　FT-M7032 long-vector processor architecture

1.3　相关研究

当前,已有一些面向循环神经网络推理的优化研究,比如:TensorFlow 使用 cuDNN 对底层进行优化,但是其核心采用矩阵乘算法并不能匹配序列长度的可变性,导致不能最大化地利用计算资源。Holmes 等提出了 GRNN[10]引擎,该引擎最小化全局内存访问和同步开销,并通过新的数据重组、线程映射和性能建模技术来平衡芯片上资源的使用;Zhang 等[11]提出了 DeepCPU,创建了专用缓存感知分区,通过形式分析优化共享 L3 缓存到私有 L2 缓存之间的数据移动。DeepCPU 和 GRNN 存在类似的问题,主要优化的是底层算子,没有考虑序列可变带来的问题。Gao 等[12]和 Silfa 等[13]都试图通过批处理相关技术进行性能优化,但是没有针对底层算子进行进一步的提升。除此之外,还有很多针对 GPU 平台对 LSTM[14]、GRU[15]和 Transformer[16]的性能优化工作,其目的都是提升在复杂模型情况下序列任务的执行效率。

除了批处理优化和内核优化,还有一些其他方式的相关工作:Kumar 等[17]提出了一个名为 Shiftry 自动编译器,将浮点深度学习模型编译成 8 位和 16 位定点模型,显著降低内存需求,并且使用了 RAM 管理机制,可在物流网设备上获得更低的延迟和更高的准确性;Thakker 等[18-21]提出了多种 RNN 压缩方法,探索了一种新的压缩 RNN 单元实现,最终实现了比剪枝更快的推理运行时间和比矩阵分解更好的精度的效果。然而,上述工作都是在牺牲精度的情况下进行 RNN 推理优化,存在一定的局限性。此外,还有一些基于现场可编程门阵列[22-23](field programmable gate array, FPGA)的 RNN 高效实现研究。

2 面向 FT-M7032 处理器的 RNN 推理并行优化

包括 RNN、LSTM 在内的类循环神经网络计算过程主要分为矩阵向量乘和激活函数等边缘算子。本节对两类算子均进行了优化。其中,重点对计算耗时较大的矩阵计算进行了优化,提出并实现了面向长向量体系结构的行优先通用矩阵向量乘(general matrix vector multiplication, GEMV)算法,有效提升计算效率。为了减少对片外存储空间的访问,还将边缘算子和部分矩阵算子进行了融合。

2.1 面向长向量处理器的行优先矩阵向量方法

常见的矩阵向量乘有两种格式,第一种为 $\boldsymbol{y}=\boldsymbol{xA}$,第二种为 $\boldsymbol{y}=\boldsymbol{Ax}$,RNN 模型更加接近于第一种,其中 $\boldsymbol{x}$ 代表输入序列向量,$\boldsymbol{A}$ 代表权重矩阵。假定矩阵 $\boldsymbol{A}$ 的大小为 $k\times n$,向量 $\boldsymbol{x}$ 的大小为 $1\times k$,向量 $\boldsymbol{y}$ 的大小为 $1\times n$,且

$$y_j = x_i \times \sum_{i=0}^{k-1} A_{ij} \tag{2}$$

式中,$j=0,1,2,\cdots,n-1$。传统的实现方式是取向量 $\boldsymbol{x}$ 中的第 i 个元素与矩阵 $\boldsymbol{A}$ 的每一行的第 i 个元素进行对应的计算。然而,对于超长向量体系结构来说,这种实现方式难以有效利用长向量处理单元:一是矩阵 $\boldsymbol{A}$ 需要转置,增加大量的数据访问,对于存储带宽本就较低的处理器而言,会导致性能急剧下降;二是需要对最后结果进行规约,FT-M7032 处理器结构对于规约计算并不友好。

针对这一问题,提出按行计算的 GEMV 算法,该向量化方法的基本思想是每次计算矩阵 $\boldsymbol{A}$ 的一行元素,计算向量 $\boldsymbol{y}$ 的第 j 个值由 k 次向量累加完成。首先将向量 $\boldsymbol{x}$ 分成 p 组,假定 k 能被 p 整除,每组包含 q 个元素,然后取矩阵 $\boldsymbol{A}$ 的 q 行,与向量 $\boldsymbol{x}$ 对应的 q 个元素进行乘加运算,之后分别从矩阵 $\boldsymbol{A}$ 以及向量 $\boldsymbol{x}$ 取下一组元素进行对应的计算,并且与上一组的结果进行相加,最后在 p 组内部进行对应相加运算。图 4 表示其中组内的矩阵向量乘实现。

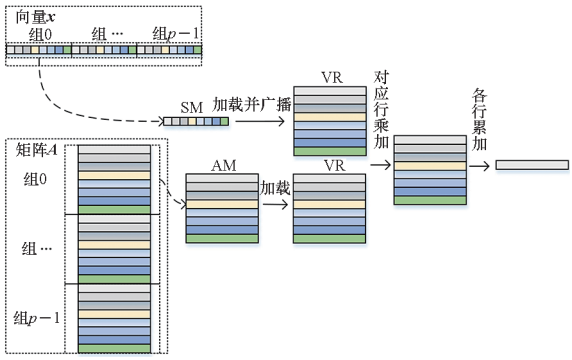


图 4 组内的矩阵向量乘实现

Fig. 4 Matrix vector multiplication implementation within a group

基于上述原理,为减少循环开销,本文通过手动编排汇编代码,让 VPE 的 v 个 FMAC 指令同时并行执行,并根据 FMAC 指令延迟槽进行循环展开,以获得较好的流水排布和最佳的性能。以 $q=8$ 为例,核心汇编循环如图 5 所示。SLDH 指令表示将输入数据从 SM 中按照半字大小格式加载到标量寄存器中,这时标量寄存器中低位数据为传输的半字大小的数据,高位数据为 0,但是因为标量寄存器可以存储单字大小的数据,所以使用 SFEXTS32L 指令将标量寄存器中的低位数据扩展到高位上,使得高低位的数据都为传输进来的单字大小的数据,继而使用 SVBCAST 指令将数据从标量寄存器广播到向量寄存器 VR,使用 VLDW/VLDDW 指令将数据以单字/双字大小从 AM 加载到另外的向量寄存器 VR,并使用 VFMULAS32 指令对不同的 VR 寄存器中的数据进行乘加计算,其中的 SBR 指令为条件跳转语句,用于算法的循环,SUB 指令用于控制循环变量的自减。

SLDH	SLDH	SLDH	SLDH	SLDH	SLDH	SLDH	SLDH
SFEXTS32L	SFEXTS32L	SFEXTS32L	SFEXTS32L	SFEXTS32L	SFEXTS32L	SFEXTS32L	SFEXTS32L
SVBCAST	SVBCAST	SVBCAST	SVBCAST	SVBCAST	SVBCAST	SVBCAST	SVBCAST
VLDW	VLDW	VLDW	VLDW	VLDW	VLDW	VLDW	VLDW
VLDDW	VLDDW	VLDDW	VLDDW	VLDDW	VLDDW	VLDDW	VLDDW
VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32
VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32
VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32	VFMULAS32
	SBR	SUB					

图 5 矩阵向量乘的核心循环

Fig. 5 Core loop of GEMV

本文提出的矩阵向量乘的汇编代码的计算单元利用率很高,当 $k = 1\ 024$ 时,最高利用率为 99.3%;当 $k = 512$ 时,最高利用率为 98.6%;当 $k = 256$ 时,最高利用率为 97.3%。

2.2　向量拼接

如式(3)所示,循环神经网络计算包括大量的矩阵向量乘。

$$\boldsymbol{x} \times \boldsymbol{W}_i + \boldsymbol{B}_i + \boldsymbol{h} \times \boldsymbol{W}_h + \boldsymbol{B}_h \tag{3}$$

式中,$\boldsymbol{x}$ 和 $\boldsymbol{h}$ 为向量,$\boldsymbol{W}_i$ 和 $\boldsymbol{W}_h$ 为权重,$\boldsymbol{B}_i$ 和 $\boldsymbol{B}_h$ 为偏置。如果按照式(3)进行计算和长向量内核调用,需要调用 2 次矩阵向量乘,且每次调用的计算量较小,对于计算能力较强的处理器,会导致硬件利用率较低。此外,对于计算单元足够多的处理器,这种直接碎片式的调用也导致大量碎片式存储空间的搬运过程。如式(3)所示,完成 RNN 的一次操作,需要 11 次数据传输(每个原始数据需要 1 次,一共 6 次;每次运算出的中间结果需要一次输出数据的传输,一共 5 次)。基于上述问题,可以通过数据拼接,将多个小规模计算拼接成较大规模数据的计算,以利用长向量处理单元的强大计算能力和 DMA 的大块数据传输效率。

具体而言,将 $\boldsymbol{x}$ 和 $\boldsymbol{h}$ 以及 $\boldsymbol{W}_i$ 和 $\boldsymbol{W}_h$ 分别进行拼接,并将 $\boldsymbol{B}_i$ 和 $\boldsymbol{B}_h$ 进行累加,将式(3)转换成:

$$\boldsymbol{x} \times \boldsymbol{W}_i + \boldsymbol{B}_i + \boldsymbol{h} \times \boldsymbol{W}_h + \boldsymbol{B}_h = \boldsymbol{x\_h} \times \boldsymbol{W}_{ih} + \boldsymbol{B}_{ih} \tag{4}$$

按照上述公式,1 次 RNN 样本的运算只需要进行 1 次加法以及 1 次乘法,以及 5 次数据传输(每个原始数据需要 1 次,共 3 次;每次运算出的中间结果需要一次输出数据的传输,共 2 次)。这样做的好处在于,虽然整体的计算量不变,但是通过减少碎片化数据的传输,提高了带宽利用率。合并方式如图 6 所示。对于有多个时间步的计算,输入 $\boldsymbol{x\_h}_i t_j$ 按照如图 6 方式在双倍速率同步动态随机存储(double data rate synchronous dynamic random access memory, DDR)中排列。其中 $\boldsymbol{x\_h}_i t_j$ 表示第 i 个 batch 的第 j 个时间步输入数据。

batch1	$x\_h_0t_0$	$x\_h_0t_1$	…	$x\_h_0t_{n-2}$	$x\_h_0t_{n-1}$
batch2	$x\_h_1t_0$	$x\_h_1t_1$	…	$x\_h_1t_{n-2}$	…
batch3	…	…	…	…	…

图 6　向量拼接后输入数据结构

Fig. 6　Input data structure after vector concatenation

2.3　边缘算子融合

RNN 模型的边缘算子一般为 tanh 和 sigmoid,公式分别如式(5)和式(6)所示。

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{5}$$

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{6}$$

这两个算子都和 e^x 相关,所以问题转换为高效求解 e^x。本文通过泰勒展开的方式构造 e^x 的近似值,如式(7)所示。

$$e^x = 1 + \frac{1}{1!}x + \frac{1}{2!}x^2 + \cdots + \frac{1}{5!}x^5 + \frac{1}{6!}x^6 \tag{7}$$

为了更快收敛,采用以下步骤进行优化(需要注意的是,该方法需要上下界约束):

步骤 1:预先计算 ln2。

步骤 2:令 $l = \lfloor \frac{x}{\ln 2} \rfloor$,$r = x - l \times \ln 2$。

步骤 3:$e^x = e^{r + l \times \ln 2} = e^r e^{l \times \ln 2} = e^r \cdot 2^l$。

因为 $0 \leqslant r < \ln 2 < 1$,所以 e^r 会收敛得比较快,而 2^l 可以通过位平移得到。当求得 e^x 的近似值(在深度学习推理中,一般展开 6 次就能得到不影响最终结果的近似值),tanh 和 sigmoid 算子便也能较好地构造,其中构造 e^x 的伪代码如算法 1 所示。

算法 1　基于 DSP 的 e^x 汇编核心算法

Alg. 1　Assembly kernel code of e^x on DSP

输入:大小为 *dlen* 的数据 *input*,数据的上界 *upbor*,下界 *downbor*

输出:大小为 *dlen* 的数据 *output*

```
1.  z←向量长度, dindex←0
2.  while dindex < dlen do
3.    VR0 ~ 7←ln2, 1/ln2, 1!, 2!, 3!, 4!, 5!, 6!
4.    VR8 ~ 10←VLD(input)
5.    if VR8 ~ 10 > upbor then
6.      VR8 ~ 10←upbor
7.    end if
8.    if VR8 ~ 10 < downbor then
9.      VR8 ~ 10←downbor
10.   end if
11.   VR11 ~ 13←⌊VR8 ~ 10 / VR0⌋
12.   VR14 ~ 16←VR8 ~ 10 − VR11 ~ 13 × VR0
13.   VR14 ~ 16←通过式(7)使用乘加运算及 VR0 ~ 7 计算 e^r
14.   VR18 ~ 20←VR11 ~ 13 左移 23 位
15.   VR18 ~ 20←VR18 ~ 20 + VR11 ~ 13 定点加法得到 e^r · 2^l
16.   output←VST(VR18 ~ 20)
17.   input + = z, output + = z, dindex + = z
18. end while
```

当使用训练框架时，如 TensorFlow 或 PyTorch 来推理 RNN 模型时，大量的函数调用和中间临时结果传输导致大量的时间花费在一些非计算密集型的边缘算子上。可通过内核融合的方法减少内存访问的数量，利用片上存储提高数据的局域性。同时也可以减少内核启动开销以减少额外开销。本文主要实现了两个级别的内核融合策略：对于 tanh、sigmoid 等逐元素计算的算子，利用寄存器进行计算单元的融合；而对于向量加、点乘等需要两个输入的算子，利用核内空间 AM 的大容量进行算子融合。

2.4 数据敏感的多核并行优化方法

FT-M7032 处理器采用共享 GSM 和 DDR 的存储结构，且拥有最多 8 个计算核心。如何利用更多的计算核心对批量数据进行处理以提高计算效率，以及如何进行任务划分以提高片上存储利用率成为多核优化的关键所在。

FT-M7032 异构处理器通过 DMA 完成核内外空间的数据传输，例如从 DDR 到 GSM 或者 AM 等片上存储空间，常用的传输方式包括点对点和广播两种方式。

通常来说，GSM 的带宽是 DDR 带宽的 10 倍以上，为了更好地提高存储带宽利用率，可将数据尽可能地存放在 GSM 上，但是由于 GSM 的空间有限，所以有些数据也会溢出到 DDR 中。为了尽可能地减少对 DDR 的访问，本文设计了数据敏感的数据划分方式，根据输入数据的大小采用不同的数据存储模式和传输方式。

当向量长度较大时，即 n 较大的情况下，在 n 方向分核。将 $\boldsymbol{W}_{ih}$ 和 $\boldsymbol{B}_{ih}$ 存储在 DDR 上，通过点对点传输到核内空间 AM；$\boldsymbol{x}$ 存储在 GSM 中，通过点对点传输到核内空间 SM；$\boldsymbol{O}$ 存储在 AM 中，通过点对点传输到 DDR 中，如图 7 所示，图中 nc 表示计算核的数量。此外，还根据权重的大小对数据存储生命周期进行了优化，如果满足式(8)，将会让 $\boldsymbol{W}_{ih}$ 常驻在 AM 中，极大地减少数据在 AM 和 DDR 之间的传输。

$$n - nc \times z < 0 \tag{8}$$

式中，z 表示 FMAC 能处理的向量长度。

在这种实现模式下，每个核上的 $\boldsymbol{x}$ 是一样的，而每个核上的 $\boldsymbol{W}_{ih}$ 和 $\boldsymbol{B}_{ih}$ 是不一样的。因为在 GSM 上使用点对点和广播方式的传输效率近似，所以采用点对点传输。

然而，当 n 较小时，根据向量长度在 n 方向分核会导致部分核长时间无法利用。而推理应用通常会采用高吞吐模式，将大批量数据捆绑输入，也就是说有很多条序列同时输入，使得 batch_size (bsz)很大。针对这种情况，本文设计了序列维度的任务划分方式，如图 8 所示。

当输入序列数量较多时，采用 bsz 方向并行的模式。将 $\boldsymbol{W}_{ih}$ 和 $\boldsymbol{B}_{ih}$ 放在 DDR 上，通过广播方式传输；$\boldsymbol{x}$ 放在 GSM 上，通过点对点传输。

在这种实现方式中，每个核上的 $\boldsymbol{x}$ 不一样，而 $\boldsymbol{W}_{ih}$ 和 $\boldsymbol{B}_{ih}$ 是一样的。考虑到在 DDR 上使用点对点和广播方式的传输效率差异大，而广播方式效率更高，因此采用广播传输。

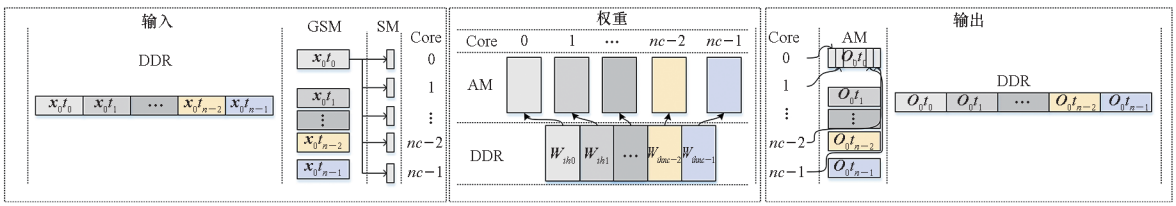
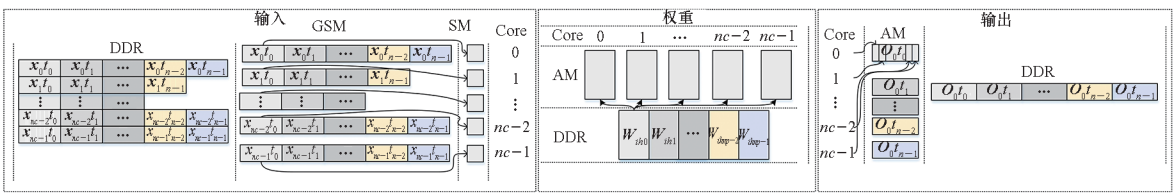


图 7 基于向量长度并行的多核分配方案

Fig. 7 Multi-core distribution scheme based on vector length parallelism



注：$\boldsymbol{W}_{ihnp}$ 表示 $\boldsymbol{W}_{ih}$ 的列数除以 nc 并且向上取整。

图 8 基于多批量并行的多核分配方案

Fig. 8 Multi-core distribution scheme based on multi-batch parallelism

3　实验结果和分析

对提出的面向长向量处理器的 RNN 并行优化实现在自主 FT-M7032 处理器上进行了测试,并与基于 FT-2000 +(16 核)以及 Intel Gold 6242R(20 核)的实现结果进行了对比和分析。首先,基于 SQuAD - 1.1[24] 以及 OpenSLR LibriSpeech Corpus[25] 两个数据集,分别在 FT-M7032 处理器和其他平台上对 RNN 和 LSTM 模型进行了性能测试以及比较;然后,对提出的各种优化方法的效果进行了细化评估;接着对循环神经网络中主要的计算模式矩阵向量乘进行了测试;最后与传统方法进行了对比。

测试中向量 $\boldsymbol{x}$ 的大小为 $1 \times k$,矩阵 $\boldsymbol{A}$ 的大小为 $k \times n$,得到的向量 $\boldsymbol{y}$ 的大小为中 $1 \times n$。为了测试的多样性,对 RNN 以及 LSTM 模型常用的 4 个参数进行了 6 组配置,如表 1 所示。其中 batch_size(*bsz*)、input_size(*ipts*)、hidden_size(*hids*)、num_layers(*nl*)分别代表批次大小、输入大小、隐藏层大小、网络层数。

表 1　RNN 测试参数

Tab. 1　RNN test parameters

配置编号	*bsz*	*ipts*	*hids*	*nl*
1	1	240	320	1
2	32	1 024	1 024	1
3	64	1 024	1 024	1
4	64	1 024	1 024	4
5	128	1 024	1 024	4
6	64	1 024	2 048	4

3.1　整体性能评测

采用矩阵向量乘的方法适应序列的可变性,并将边缘算子与矩阵乘算子进行了融合以获得在长向量处理器上的良好性能。以上述两个数据集为输入,分别在 FT-M7032、FT-2000 +(16 核)以及 Intel Gold 6242R(20 核)上对 RNN 和 LSTM 的性能进行了测试,对同一数据集在不同平台下的吞吐量进行了对比。其中后面两个 CPU 平台的实现是基于 PyTorch 框架,后端是标准的深度神经网络库。测试结果分别如图 9 和图 10 所示。

从图 9 中可以看出,在各组参数下,提出的方法在两个数据集上的性能都能大幅超过对应循环神经网络模型在 FT-2000 + 上的性能。从多组参数配置的加速效果可以看出,当参数配置较大时,加速效果更加明显,主要是因为:参数小时的计算量也较小,可并行度较低,计算占比较低;而当参数量增大时,可并行度增加,计算占比增加,加速比有明显提升,最大可高达 62. 68。

从图 10 中可以看出,除了小参数的情况,提出的方法在两个数据集上的性能都能超过对应模型在 Intel CPU 上的性能,RNN 模型加速比最大能达到 3. 51,LSTM 模型最大加速比为 3. 12。而 FT-M7032 的峰值计算能力与 Intel CPU 相当,说明提出的实现方法针对硬件结构进行了有效的优化。

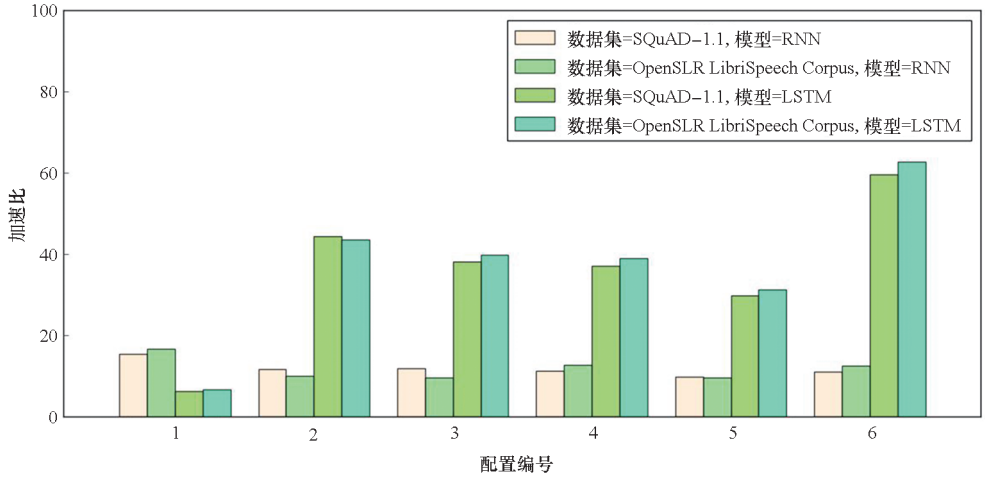


图 9　基于长向量处理器的 RNN 和 LSTM 模型推理相比于 FT-2000 + 的性能加速比

Fig. 9　Performance acceleration ratio of RNN and LSTM model inference based on long vector processor compared to FT-2000 +

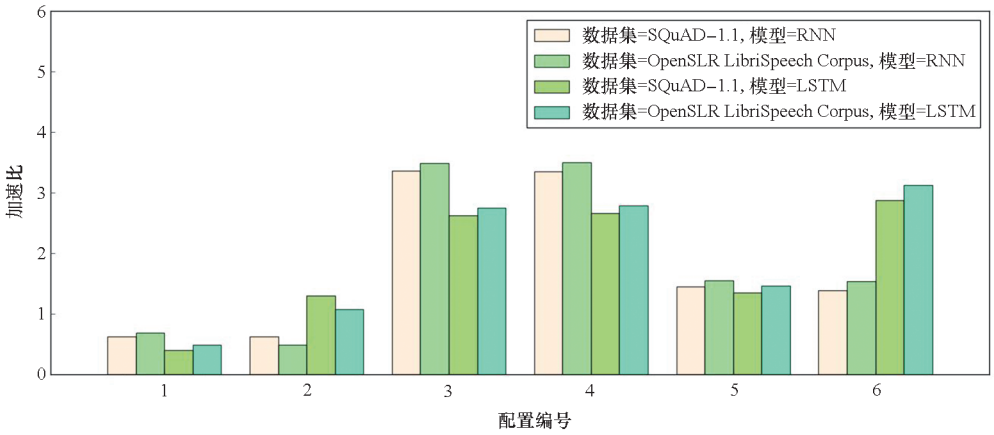


图 10　基于长向量处理器的 RNN 和 LSTM 模型推理相比于 Intel Gold 6242R 的性能加速比

Fig. 10　Performance acceleration ratio of RNN and LSTM model inference based on long vector processor compared to Intel Gold 6242R

3.2　基于可变序列的 RNN 以及 LSTM 性能加速效果

本文采取矩阵向量乘的方法而不是矩阵乘的方法进行 RNN 及其变种 LSTM 的处理，主要原因是 batch 中的序列长度不一致导致计算量不一致，如果使用矩阵乘实现，将引入很多的无效计算。分析了两个数据集的序列长度分布，如图 11、图 12 所示。从图中可以看出，序列长度非常不均匀，如果对序列长度进行补齐的话，将存在大量的无效计算，可能导致性能的下降。

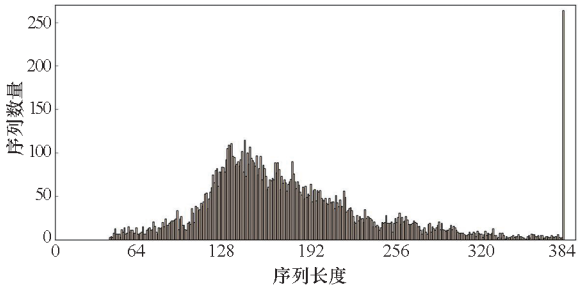


图 11　SQuAD－1.1 数据集序列长度分布

Fig. 11　Sequence length distribution of SQuAD-1.1 dataset

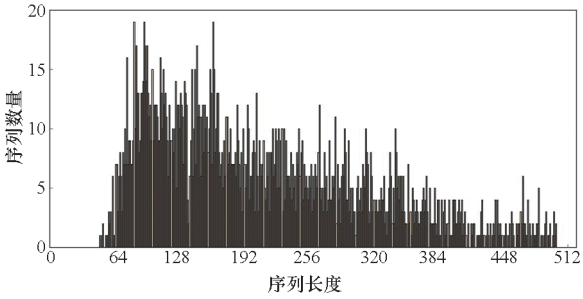


图 12　OpenSLR LibriSpeech Corpus 数据集序列长度分布

Fig. 12　Sequence length distribution of OpenSLR LibriSpeech Corpus dataset

在不同的参数下，为了适应可变序列，采用矩阵向量乘的方法，并且根据数据集的特点，将 SQuAD－1.1 的最大序列长度设置为 384，将 OpenSLR LibriSpeech Corpus 数据集的最大序列长度设置为 500，将适应可变序列的方法与固定最大序列长度的方法进行对比，其中 RNN 模型的对比结果如图 13所示。

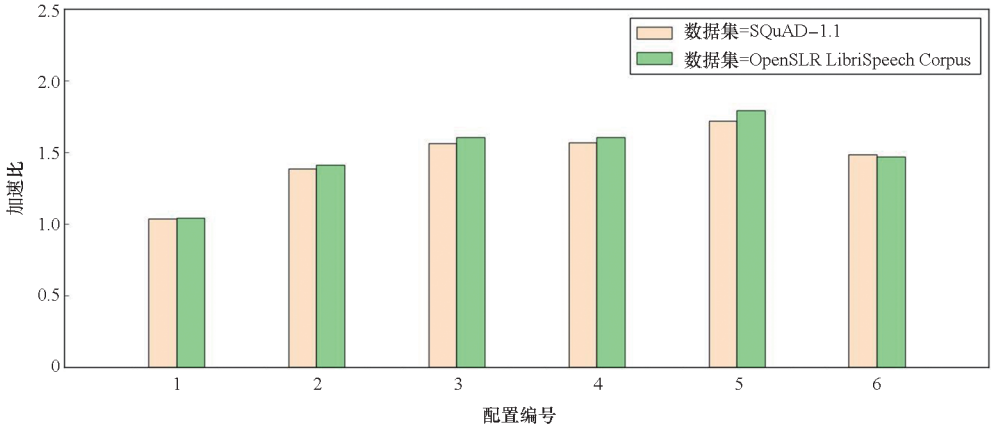


图 13　使用 RNN 模型时，可变序列长度相比于固定序列长度的性能加速比

Fig. 13　Performance acceleration ratio of variable sequence length compared to fixed sequence length when using RNN model

从图 13 中可以看出,当 *bsz*、*ipts* 以及 *hids* 较小时,提出的实现方式收益较小,因为在此情况下,主要的时间消耗在数据搬运上,提高计算效率的影响不大。但是随着各个参数的增大,数据搬运的耗时占比慢慢减小,因此,通过针对算子等进行的极致优化效果变得明显,最高能产生 1.79 的加速比。

3.3　矩阵向量乘性能

本文实现了高效的矩阵向量乘算子库,通过手动编排汇编的方式提高计算效率,对优化实现后的矩阵向量乘和基于 PyTorch 框架下矩阵向量乘在 FT-2000 + CPU 的执行时间进行对比,不同规模下的加速比如图 14 所示。

从图 14 中可以看出,所有参数下,提出的矩阵向量乘的效率都高于 FT-2000 +,而当矩阵向量维度比较大的时候,性能提升更加明显,性能提升可达 6 倍以上。同时,本文注意到矩阵向量乘的加速效果不如 RNN 以及 LSTM 模型整体效果明显,其原因有两点:第一点是矩阵向量乘本质上不受可变序列长度的影响;第二点是使用 RNN 以及 LSTM 模型时,可以采用大批次,能更好地复用权重。

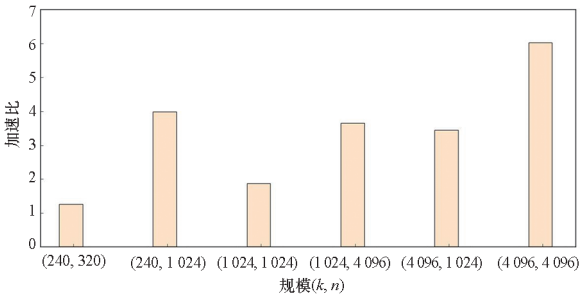


图 14　基于长向量处理器的矩阵向量乘相比于 FT-2000 + 的性能加速比

Fig. 14　Performance acceleration ratio of matrix vector multiplication based on long vector processor compared to FT-2000 +

3.4　不同 RNN 推理方法对比测试

通用的框架,如 PyTorch,使用矩阵乘实现 RNN 模型的推理,其主要考虑的问题是可以批量执行。因此,矩阵乘算法需要首先对序列进行预处理,将所有的带处理的序列进行对齐,然后进行并行处理。这种模式需要更大的内存空间,同时会引入很多额外的计算。本文对比了在长向量处理器上使用矩阵乘以及矩阵向量乘对 RNN 模型进行推理的吞吐量性能,如图 15 所示,提出的基于矩阵向量乘的方法在 RNN 模型推理上的加速比最大可达 1.82。

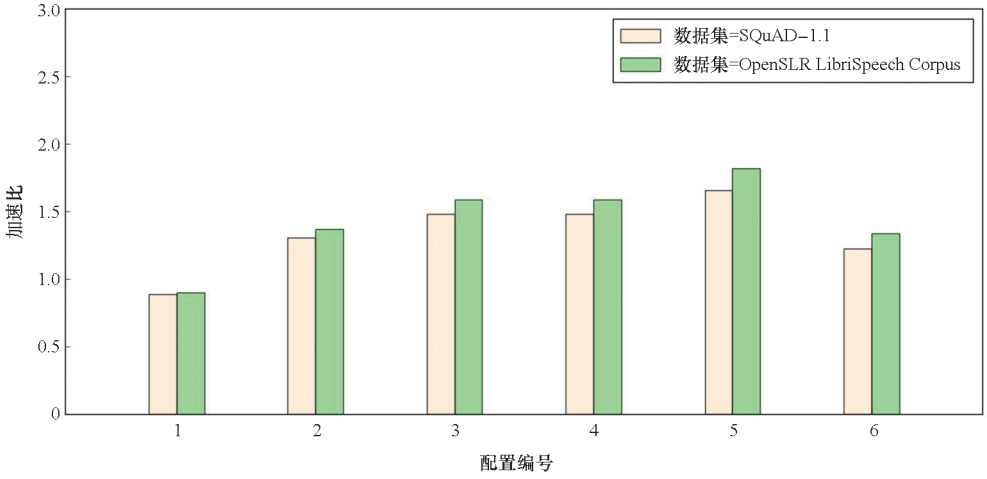


图 15　基于矩阵向量乘优化方法相对于矩阵乘实现方法的 RNN 模型推理加速

Fig. 15　Performance acceleration ratio of matrix-vector multiplication for RNN model inference compared to matrix multiplication

4　结论

本文面向 FT-M7032 长向量处理器对 RNN 模型推理进行了优化。首先,对 RNN 神经网络结构进行剖析,将其分为了矩阵向量乘算子和边缘算子,并根据处理器的体系结构,针对矩阵向量乘算子,提出了基于按行计算的矩阵向量乘算法,在多核层面采用了内核选择等优化方法;而针对边缘算子,提出了两个级别的内核融合优化方法,并且使用手写汇编对单核的两种算子进行了优化,旨在挖掘 FT-M7032 处理器的最佳性能。实验表明,提出的面向长向量处理器的 RNN 实现效果较好,相较于 FT-2000 + 以及 Intel Gold 6242R,LSTM 模型最多分别获得了 62.68 倍以及 3.12 倍

的性能加速。

参考文献(References)

[1] HOCHREITER S, SCHMIDHUBER J. Long short-term memory[J]. Neural Computation, 1997, 9(8): 1735 – 1780.

[2] CHO K, VAN MERRIENBOER B, GULCEHRE C, et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation[C]//Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014.

[3] RAHMAN S, CHAKRABORTY P. Bangla document classification using deep recurrent neural network with BiLSTM[C]//Proceedings of International Conference on Machine Intelligence and Data Science Applications, 2021.

[4] SAON G, TÜSKE Z, BOLANOS D, et al. Advancing RNN transducer technology for speech recognition[C]// Proceedings of 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2021.

[5] ORUH J, VIRIRI S, ADEGUN A. Long short-term memory recurrent neural network for automatic speech recognition[J]. IEEE Access, 2022, 10: 30069 – 30079.

[6] ASHENGO Y A, AGA R T, LEMMA ABEBE S. Context based machine translation with recurrent neural network for English-Amharic translation[J]. Machine Translation, 2021, 35(1): 19 – 36.

[7] PASZKE A, GROSS S, MASSA F, et al. PyTorch: an imperative style, high-performance deep learning library[C]//Proceedings of the 33rd International Conference on Neural Information Processing Systems, 2019.

[8] ABADI M, BARHAM P, CHEN J M, et al. TensorFlow: a system for large-scale machine learning[C]//Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, 2016.

[9] CHETLUR S, WOOLLEY C, VANDERMERSCH P, et al. cuDNN: efficient primitives for deep learning[C]// Proceedings of Introduction to Deep Neural Networks, 2016.

[10] HOLMES C, MAWHIRTER D, HE Y X, et al. GRNN: low-latency and scalable RNN inference on GPUs[C]// Proceedings of the Fourteenth EuroSys Conference, 2019.

[11] ZHANG M J, RAJBHANDARI S, WANG W H, et al. DeepCPU: serving RNN-based deep learning models 10x faster[C]//Proceedings of the 2018 USENIX Annual Technical Conference, 2018.

[12] GAO P, YU L F, WU Y W, et al. Low latency RNN inference with cellular batching[C]//Proceedings of the Thirteenth EuroSys Conference, 2018: 1 – 15.

[13] SILFA F, ARNAU J M, GONZÁLEZ A. E-BATCH: energy-efficient and high-throughput RNN batching[J]. ACM Transactions on Architecture and Code Optimization, 2022, 19(1): 1 – 23.

[14] ZHENG W D, CHEN G. An accurate GRU-based power time-series prediction approach with selective state updating and stochastic optimization[J]. IEEE Transactions on Cybernetics, 2022, 52(12): 13902 – 13914.

[15] XU X H, YONEDA M. Multitask air-quality prediction based on LSTM-autoencoder model[J]. IEEE Transactions on Cybernetics, 2021, 51(5): 2577 – 2586.

[16] FANG J R, YU Y, ZHAO C D, et al. TurboTransformers: an efficient GPU serving system for transformer models[C]// Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2021: 389 – 402.

[17] KUMAR A, SESHADRI V, SHARMA R. Shiftry: RNN inference in 2 KB of RAM[J]. Proceedings of the ACM on Programming Languages, 2020, 4(OOPSLA): 1 – 30.

[18] THAKKER U, BEU J, GOPE D, et al. Compressing RNNs for IoT devices by 15 – 38x using Kronecker products[J]. Machine Learning, 2019.

[19] THAKKER U, DASIKA G, BEU J, et al. Measuring scheduling efficiency of RNNs for NLP applications[J/OL]. Distributed, Parallel, and Cluster Computing, 2019: arXiv: 1904.03302[2022 – 10 – 10]. https://arxiv.org/pdf/1904.03302.pdf.

[20] THAKKER U, FEDOROV I, BEU J, et al. Pushing the limits of RNN compression[C]//Proceedings of Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS), 2021.

[21] THAKKER U, BEU J, GOPE D, et al. Run-time efficient RNN compression for inference on edge devices[C]// Proceedings of 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2), 2019.

[22] CHANG A X M, CULURCIELLO E. Hardware accelerators for recurrent neural networks on FPGA[C]//Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS), 2017.

[23] SUN Y X, AMANO H. FiC-RNN: a multi-FPGA acceleration framework for deep recurrent neural networks[J]. IEICE Transactions on Information and Systems, 2020, E103.D(12): 2457 – 2462.

[24] RAJPURKAR P, ZAHNG J, LOPYREV K, et al. SQuAD: 100 000 + questions for machine comprehension of text[C]// Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, 2016.

[25] PANAYOTOV V, CHEN G G, POVEY D, et al. LibriSpeech: an ASR corpus based on public domain audio books[C]//Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2015.

(编辑: 熊立桃, 杨琴)