RTL级可扩展高性能数据压缩方法实现

陈晓杰[1],李 斌[2],周清雷[2]
（1. 数学工程与先进计算国家重点实验室,河南郑州450001；2. 郑州大学计算机与人工智能学院,河南郑州450001）

摘 要： 针对传统的数据压缩实现方法处理性能较低,难以满足高速网络高负载、低能耗要求,本文提出了基于FPGA（Field-Programmable Gate Array）的高性能数据压缩方法. 在数据计算方面,定制化一种专用并行数据匹配方法,并对压缩算法进行子任务划分,设计细粒度的串/并混合结构实现数据压缩和数据编码；在数据存储方面,设计了面向硬件的专用高效字典处理,并采用多级缓存机制优化访存结构；基于FPGA的资源面积,设计了多通道、可扩展数据压缩结构,并采用轮询策略实现多通道的数据分配和回收；在优化过程中,采用RTL（Register Transfer Level）实现数据压缩算法. 实验结果表明优化后的压缩算法与CPU相比达到了1.634的加速比,吞吐量为4.33 Gb/s.

关键词： 数据压缩；FPGA；多通道；并行匹配；多级缓存
中图分类号： TP391 **文献标识码：** A **文章编号：** 0372-2112(2022)07-1548-10
电子学报URL:http://www.ejournal.org.cn **DOI:**10.12263/DZXB.20210448

Implementation of RTL Scalable High-Performance Data Compression Method

CHEN Xiao-jie[1], LI Bin[2], ZHOU Qing-lei[2]
（1. *State Key Laboratory of Mathematical Engineering and Advanced Computing*, *Zhengzhou*, *Henan* 450001, *China*;
2. *School of Computer and Artificial Intelligence*, *Zhengzhou University*, *Zhengzhou*, *Henan* 450001, *China*）

Abstract: As the low processing performance makes traditional data compression implementation methods difficult to meet the high load and low energy consumption requirements of high-speed networks, a high-performance data compression method based on field-programmable gate array is proposed. In terms of data calculation, this paper customize a dedicated parallel data matching method, divide the compression algorithm into sub-tasks, and design a fine-grained serial/parallel hybrid structure to achieve data compression and data encoding. In terms of data storage, a dedicated and efficient dictionary processing for hardware is designed, and a multi-level cache mechanism is used to optimize the memory access structure. Based on the resource area of FPGA, a multi-channel, scalable data compression structure is designed, and a polling strategy is used to realize multi-channel data allocation and recovery. In the optimization process, register transfer level is used to realize the data compression algorithm. The experimental results show that the optimized compression algorithm achieves a speedup ratio of 1.634 compared with the CPU, with a throughput of 4.33 Gb/s.

Key words: data compression; field-programmable gate array; multi-channel; parallel matching; multi-level cache

1 引言

随着信息技术的迅速发展,5G网络逐步普及,接入互联网的设备将迅速增长,根据相关统计和预测,到2025年,接入互联网的设备将达到5 000亿[1]. 通信设备的急剧增加导致数据量呈现爆炸式的增长,为了应对海量数据的处理,逐步形成了在边缘计算、内存计算、智能计算等模式下,以数据为中心的新一代计算网络. 但是,大量的网络数据带来的网络负载和存储问题仍然制约着网络性能的进一步发展.

数据压缩技术在满足用户获取原信息的同时,通过有效的编码,能够减少数据量,在数据的存储管理和网络数据传输方面得到了有效的应用[2,3]. 数据压缩分为有损压缩和无损压缩,前者主要应用于视频和图像,即使某些数据的丢失对于用户的感官也没有太大的影响,代表的算法有基于离散余弦变换的数据压缩、基于小波的数据压缩和基于线性预测编码的数据压缩等.

收稿日期:2021-04-07;修回日期:2022-04-09;责任编辑:李勇锋
基金项目:国家自然科学基金（No.61702518）;国家重点研发计划重点专项（No.2018XXXXXXXX01）

后者主要应用于数据正确性要求较高的场景,代表的算法有LZ77、RLE(Run-Length Encoding)、Snappy等.其中LZ77算法较复杂,且需要消耗较大的内存和计算资源[4],RLE主要应用于位图压缩[5],通用性较低,而Snappy是Google开源的压缩/解压缩库,在满足一定压缩率的条件下,具有较高的压缩速度,已经被应用到内存数据库和电子探测器的高容量数据压缩方面[6,7].

为了降低数据存储和网络数据传输的限制等问题,同时满足接收端获取完整的数据,采用Snappy无损压缩算法对数据进行压缩处理.传统的数据压缩算法主要以软件的形式在CPU上进行实现,压缩速度较低,并且应用场景有限,而GPU主要应用于密集型计算的任务,能耗较高.FPGA具有低功耗、可重构、能效低等特点,近几年被广泛应用于边缘神经网络的加速[8]、高速的数据加密[9]、并行数据压缩[10]等,在FPGA上实现数据压缩算法能够在有限带宽的限制下,极大的提高数据传输速度[11].因此,本文首先分析Snappy数据压缩算法的结构特征和算法流程;然后针对Snappy算法进行RTL级实现,并采用多种有效方法进行深度优化,例如专用并行数据匹配、访存优化、流水线技术等,同时,采用轮询策略、流水线数据转换等实现多通道的数据压缩算法,使算法能够动态扩展,进一步提高算法性能;最后通过算法实现、性能测试、性能对比等实验,验证本文实现方法的加速效果和性价比.

2　数据压缩原理及算法分析

Snappy压缩算法是基于hash值运算的字典匹配压缩,如果两组数据计算hash值的结果相同,则两组数据相等的概率较大.最初版本Snappy算法流程首先将待压缩数据分为32 KB大小的块分别进行压缩,然后在32 KB数据中对每字节数据与其相邻的3字节数据计算hash值,并在字典中获取有相同hash值的数据,进行匹配、压缩,最后对处理后的数据进行编码输出.但是,原始的算法压缩及硬件实现效果较差.

Xilinx在高端数据板卡Alveo U200中对Snappy算法进行了改进实现[12],降低了压缩率,压缩速度达到了10 GB/s,但是Xilinx是由高级编程语言实现压缩算法,只能应用在特定FPGA芯片,可移植性较低,因此,本文对Xilinx改进后的Snappy算法进行RTL级实现,在不损失算法压缩率的情况下,提高压缩速度和算法可移植性.改进后的Snappy算法处理框架如图1所示.

首先对数据划分为64 KB大小的block块,然后对每个块进行四个阶段的处理,完成数据的压缩和编码,最后将每个块的压缩结果写入Snappy文件,四个处理阶段是整个算法的核心,也是本文分析和实现的重点,详细分析如下:

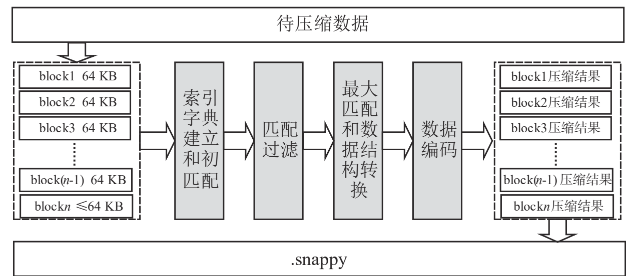

图1　Snappy数据压缩框架

(1)索引字典建立和初匹配.

初匹配阶段是数据扩充阶段,用于获得匹配的可能性,这一阶段对输入的字节数据进行处理,计算对应的匹配偏移offset和匹配长度len.

首先初始化索引字典和滑动窗口,索引字典的内容包含3字节的索引值和6字节的数据,存储结构如图2所示,索引值是6字节数据中第一个字节数据在block块的位置index.

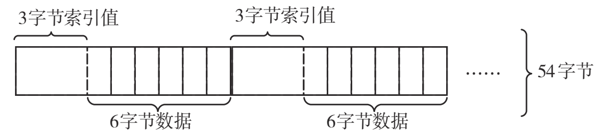

图2　索引字典存储结构

其次,依次读取待压缩内容,并写入滑动窗口,根据滑动窗口内容计算哈希值,以哈希值为字典索引,读取索引位置内容,并将窗口内容和索引值写入哈希值对应的字典中.

然后将读取内容与窗口内容进行匹配,此时匹配的最大长度为6字节.

最后,以当前的数据D_n、匹配偏移offset和匹配长度len共同构成V_n并输出,其中offset由索引值之差计算,处理结构示意图如图3所示.

图3中索引字典的每行存储6组数据,每组数据的长度是54字节,存储在字典中同一行的6组数据具有

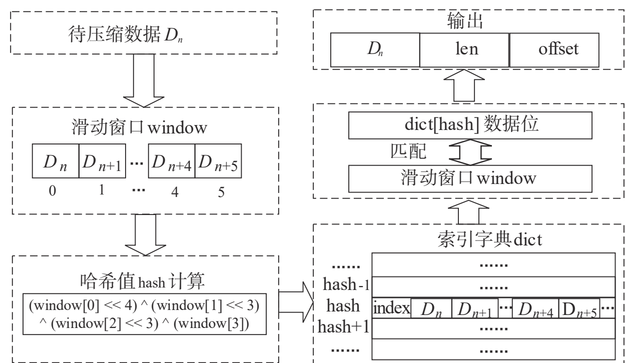

图3　压缩第一阶段处理结构图

相同的hash值，例如在字典中的hashNum地址对应的值包含6组数据，表示为$B_h \sim B_{h+5}$，每组数据的内容如式(1)所示.

$$\begin{aligned} B_h &= \{\text{index0}, D_{\text{index0}} \sim D_{\text{index0}}+5\} \\ &\cdots\cdots \\ B_h &= \{\text{index5}, D_{\text{index5}} \sim D_{\text{index5}}+5\} \end{aligned} \tag{1}$$

6组数据的索引值存在关系式(2)：

$$\text{index0} < \text{index1} < \text{index2} < \text{index3} < \text{index4} < \text{index5} \tag{2}$$

6组数据的内容存在关系式(3)：

$$\begin{aligned} \text{hash}(B_h) &= \text{hash}(B_h+1) = \text{hash}(B_h+2) = \text{hash}(B_h+3) \\ &= \text{hash}(B_h+4) = \text{hash}(B_h+5) \\ &= \text{hashNum} \end{aligned} \tag{3}$$

当计算滑动窗口的值满足hash(window)=hashNum时，则将匹配窗口的值和6组数据进行匹配，同时更新hashNum地址对应的字典行，行内数据变为B_{h+1}、B_{h+2}、B_{h+3}、B_{h+4}、B_{h+5}、index、window. 以上即完成字典的更新，在这个阶段每组数据的最大匹配长度为6，每个数据都有输出，代表了其自身和之后的5个数据与具有相同哈希值组的最大匹配.

(2)匹配过滤.

首先填充比较窗口，每个窗口存放的值是第一阶段输出的三部分组成的值. 然后，读取窗口0的值V_n，并将窗口左移，同时，读入一组数据V_{n+6}，重新填满比较窗口. 最后，将V_n与窗口中的每个值进行条件过滤. 如果满足，原样输出，否则将匹配长度与匹配偏移赋值为0. 其中，过滤条件为V_n的匹配长度与比较窗口位置相加大于比较窗口值的匹配长度. 处理结构如图4所示.

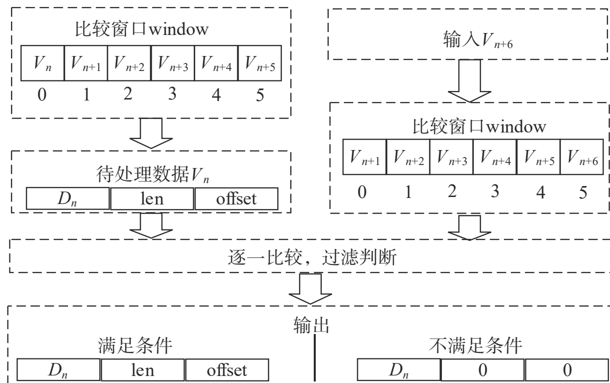

图4 压缩第二阶段处理结构图

这一阶段是获取最优匹配的可能性，并将匹配效果较差的数据设置为0匹配.

(3)最大匹配和数据结构转换.

这一阶段是数据压缩阶段，通过前两阶段的筛选，此时可以将匹配数据进行压缩，处理过程是逐个读取V_n，获取匹配偏移，然后通过偏移从匹配字典中读取数据进行最大匹配，并将匹配数据丢弃，从而完成压缩，匹配分为三种情况：

1)没有匹配，则原样输出；

2)有匹配，且偏移offset≤16 K，则依次读取数据，并丢弃，只记录匹配长度，最大匹配长度为64；

3)有匹配，且偏移offset＞16 K，则根据匹配长度，读取相应数据进行丢弃；

此时，可得到数据D_n，改变后的匹配长度len，以及匹配偏移offset.

进一步将数据进行转换，如果len为0，则将数据D_n单独输出，此时offset为0，同时，增加未压缩字符计数litCount，共同作为匹配信息输出. 如果不为0，则将数据D_n丢弃，只输出匹配数据len和offset，且litCount为0. 转换后的数据分为数据D_n和匹配信息match_info(len，offset，litCount).

4)数据编码阶段.

根据litCount、len、offset的值按照Snappy规则进行编码，编码格式内容如图5所示.

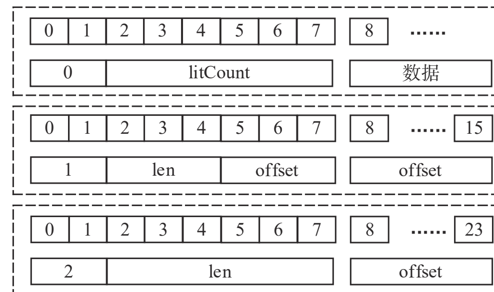

图5 编码格式

图5中每个虚线框内的上方表示编码bit位的索引，下方为编码内容，从上到下为三种编码格式：

(1)数据未压缩；

(2)len＜12，offset＜2048；

(3)2^{11}≤offset≤2^{16}并且len≤64，或者，11＜len≤64并且offset＜2^{11}.

从算法流程可以看出Snappy算法的计算量较小，但是包含了大量的数据比较和匹配，并且需要大容量字典存储数据，因此，要实现高效的Snappy算法需要对算法的关键路径、存储需求以及实现结构进行优化.

3 并行Snappy压缩算法实现

为了满足算法的计算、存储、通信需求，实现高性能的Snappy算法，本文结合FPGA的芯片资源分布，布局布线的特点，以及频率对时序的影响等，对算法的关键路径、存储需求以及实现结构等多方面进行细粒度优化.

3.1 定制化并行数据匹配

Snappy算法的前两个阶段都包含6次数据匹配，是

整个算法的关键路径,消耗较长时间,因此,需要对窗口的数据匹配进行优化.

对于第一个阶段的数据匹配,将滑动窗口数据与来自于索引字典中的6组数据分别进行匹配,匹配如算法1所示.

算法1　数据匹配

```
输入:dict[hash], window
输出:match_len
1.   len←0 ;
2.   FOR i←0 to 5 /*每组数据匹配*/
3.       compare ← dict[hash] [i];
4.       FOR j←0 to 5 /*逐个字节匹配*/
5.           IF (compare[j] = = window[j])THEN
6.               len← len+1 ;
7.           ELSE
8.               break ;
9.           END IF
10.      END FOR
11.      IF(match_len > len) THEN /*获取最长匹配*/
12.          match_len← match_len ;
13.      ELSE
14.          match_len←len , len←0;
15.      END IF
16.  END FOR
```

算法1中主要的消耗时间为两层循环,即6个字节的匹配时间t_{byte_match}和6组数据的匹配时间t_{data_match},总时间为$6t_{byte_match}\times6t_{data_match}$. 而每组数据匹配时相互独立,因此,可进行并行匹配,并行匹配如算法2所示.

算法2　并行数据匹配

```
输入:dict[hash], window
输出:match_len
1.   GENERATE i←0 to 5 /*并行模块*/
2.     compare←dict[hash][i];
3.     FOR k←0 to 5 PAR-DO /*并行字节匹配*/
4.         IF (compare[k] = = window[k]) THEN
5.             flag[k] ←1;
6.         ELSE
               flag[k] ←0;
7.         END IF
8.     END FOR
9.     len[i] ←function_long(flag) ;/*获取匹配长度*/
10.  END GENERATE
11.  match_len←function_max(len) ;/*获取匹配长度*/
```

算法2中步1～10是通过generate生成器并行化6个模块,则$6t_{data_match}$可在1个t_{data_match}时间内完成. 步3～8是并行字节匹配,消耗时间为$1t_{byte_match}$,由于是并行匹配,只能获得每个字节是否匹配,因此,增加flag作为匹配标记,并以flag为function_long函数的输入,求得每组数据的匹配长度len,最后以len为输入,通过function_max函数,求得最终匹配长度. 在RTL实现中,函数function_long和function_max通过组合逻辑进行实现,消耗时间较少,因此,通过并行匹配,关键路径消耗的总时间为$1t_{byte_match}$,极大的减少了时间复杂度.

在并行匹配的过程成,包含两步操作,赋值和匹配,布线路径从compare到flag,如果在一个时钟内进行实现,消耗大量的查找表资源,增加时序影响,因此,添加buff_com和buff_win寄存器用于中间缓存,减少FPGA布局布线的路由复杂度,同时,将字节匹配的过程分为数据赋值和匹配两个模块,RTL实现如算法3所示.

算法3　关键路径并行扩展

```
输入:compare,window
输出:flag
1.  FOR m←0 to 5   PAR-DO /*第1个时钟*/
2.      buff_com[m] ← compare [m];
3.      buff_win[m] ← window[m];
4.  END FOR
5.  FOR n←0 to 5   PAR-DO /*第2个时钟*/
6.      flag[n] ← (buff_com[n]= =buff_win[n]) ? 1′b1:1′b0;
7.  END   FOR
```

通过增加寄存器资源减少查找表资源,使并行匹配消耗的物理时间为2个时钟周期,定制化并行匹配硬件实现结构如图6所示. 时钟clk贯穿6个并行模块,在每个模块内,第一个时钟完成寄存器赋值,第二个时钟进行逻辑运算,实现匹配判断,并通过时序逻辑在较短的时延内计算匹配长度,从而两个时钟可完成并行匹配.

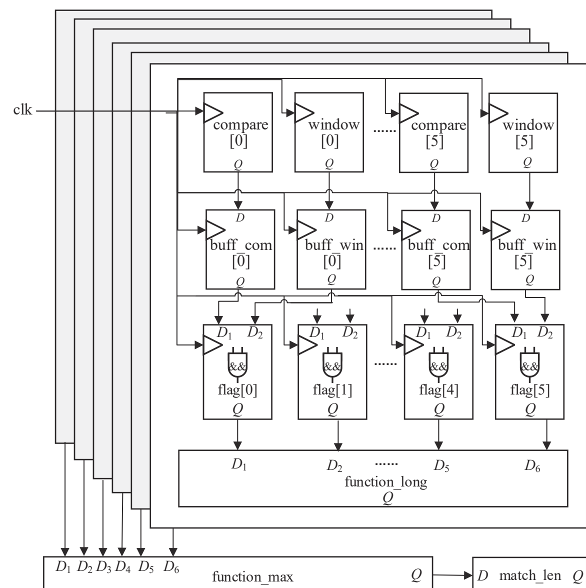

图6　定制化并行匹配硬件实现结构

3.2 细粒度串并混合结构

Snappy算法对每个字节数据进行多次处理，具有重复性，在数据压缩过程中，数据具有单向流动性，FPGA的各种逻辑计算单元具有独立性，通过时钟驱动可实现资源的并行，因此，Snappy压缩算法可在FPGA上采用流水线进行实现.

数据压缩的前三个阶段在实现中共细分为15个子任务模块，第一阶段分别为读数据、写滑动窗口、哈希值计算、读字典、数据匹配和输出缓存，其中数据匹配经过优化后分为寄存器赋值和并行匹配，共7个子任务. 第二阶段分别为读窗口数据、每个窗口的条件过滤匹配和最后的输出缓存，同样，过滤匹配也分为寄存器赋值和并行匹配，共4个子任务. 第三阶段分别为接收数据、读字典、匹配、数据转换，共4个子任务，即处理1字节数据需要经过15个子任务.

在编码阶段，由于数据是经过压缩后的数据，数据量变小，并且在不同的条件下处理两种数据，采用流水线实现会增加计算复杂度，消耗大量的计算资源，因此，在这一部分的实现上采用串行的状态机进行实现，而在处于读取明文的处理状态中，最大需要64次读取数据，通过计数器控制可进行连续读取，即使在没有压缩的极端情况下，仍然可满足要求. Snappy实现结构如图7所示.

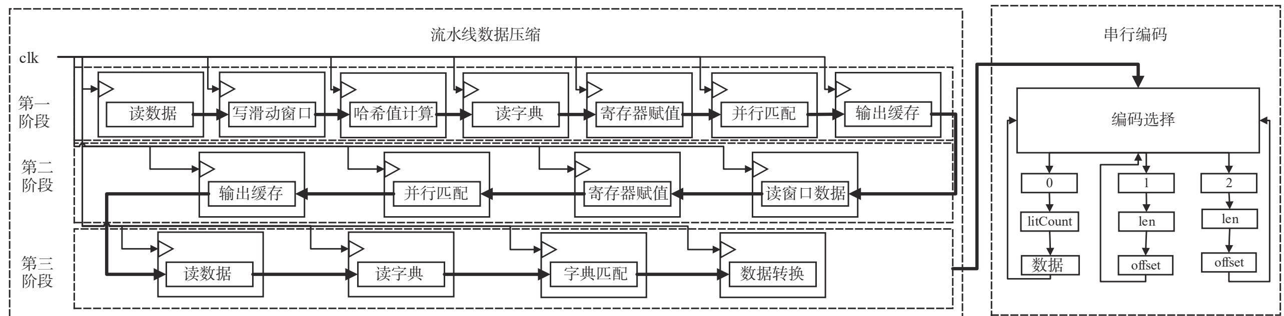

图7 Snappy算法实现结构

图7中流水线数据压缩部分表示压缩过程中的15个子运行模块，数据从输入到输出一共需要消耗15个时钟，并且15个子模块并行计算，形成15级流水线. 通过连续的读取数据，15个时钟之后，每个时钟处理一组数据，相当于每个字节数据只需要1个时钟，极大的提高了算法性能.

通过细粒度的串-并混合的结构设计，实现Snappy算法，能够最大的提高算法性能，同时，编码阶段采用了串行实现，在满足条件的情况下，可减少资源的利用和电路的负载，从而降低功耗.

3.3 面向硬件的多级缓存优化

随机存取存储器(Random Access Memory，RAM)包含地址线和数据线，将指定的数据写入指定的地址内，且读取数据后不会造成数据丢失. 先进先出存储器(First In First Out，FIFO)没有地址的约束，实现较简单，主要用于数据缓存. 因此，利用RAM和FIFO分别对字典实现和访存进行存储优化.

(1) 字典实现

Snappy压缩算法中第一、三阶段中都包含对字典的操作，通过计算索引并从索引位置存取数据，字典的空间较大，在FPGA中实现字典可采用寄存器和Block RAM进行实现，而前者消耗了有限的寄存器资源和查找表资源，对于较大的字典使得资源利用不充分. Block RAM是FPGA上专用存储单元，分布在逻辑计算资源的边界上，因此，采用Block RAM实现字典，操作结构如图8所示.

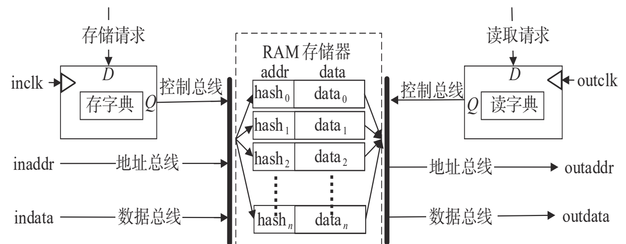

图8 字典实现及操作结构

Block RAM实现的字典操作结构中，当有存储请求时，通过时钟inclk触发执行，在控制总线作用下，将数据indata存储到地址inaddr；当有读取请求时，通过时钟outclk触发执行，从地址outaddr读取相应数据outdata，存储器中索引地址与hash值一一对应. 进行流水线计算过程中，读写操作分别只需要一个时钟即可完成，满足高性能的数据操作要求，具有较大的存储和通信效率.

(2) 访存优化

由于FPGA内没有足够的存储单元，而数据压缩需要数据存储到本地或者实时的传输到FPGA中，并且压

缩的数据是连续的,因此,利用内存DDR与FPGA的连接,将待压缩的数据存储到DDR中,进行压缩时,直接从内存存取数据.FPGA与DDR之间通过IP核MIG(Memory Interface Generator)进行连接,采用AXI4通信协议实现物理层面数据传输,为了满足Snappy算法高速的数据输入,需要对访存进行优化,实现FPGA与DDR的高效数据传输,实现结构如图9所示.

FPGA与DDR之间设计多级缓存机制进行优化,首先,添加读内存缓存模块,在该模块中预先计算待压缩数据的地址并存储在fifo_addr_rd中,通过读控制状态机读取地址,再利用AXI4协议从DDR中读取数据.其次,将读取的数据写入fifo_data_rd中,Snappy算法根据FIFO的空满信号读取数据.然后,添加写内存控制模块,并将压缩后的数据存入fifo_data_wr中,最后,预先计算写内存地址,并存入fifo_addr_wr,写控制状态机通过AXI4协议根据地址将相应数据写入DDR中.

DDR与Snappy算法之间增加了两个缓存控制模块和多级缓存,使算法的实现与地址之间进行解耦和,数据之间只存在FIFO操作,并且操作的端口相互独立,最大化的减少各模块之间的相关性,从而降低布线路由的复杂度.

Snappy压缩算法在FPGA上仅占用较小部分的计算与存储资源,并且随着工艺的提升,芯片资源也在不断增加,因此,采用多通道并行方法,当有大量的数据需要压缩处理时,对数据进行划分,通过逻辑控制将多个Snappy算法并行处理,实现最优性能.同时也可提高算法在不同芯片上的可扩展性.

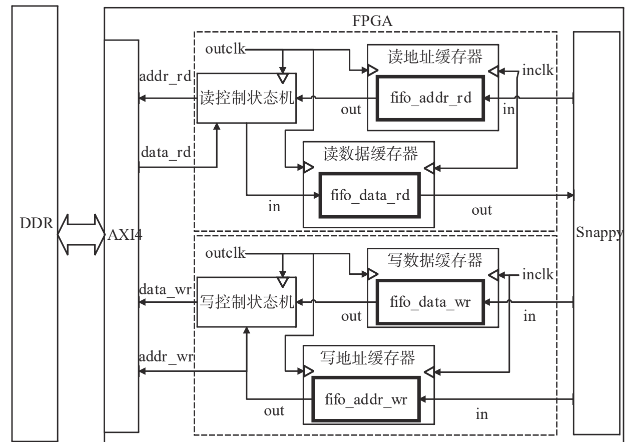

图9　访存优化结构

内存具有地址映射的功能,包含控制总线、数据总线和地址总线,数据存储内存的位置和大小可预先计算,因此,设计的第一步是不同通道的数据划分,Snappy算法将待压缩的数据按照64 KB大小进行block划分,数据是连续的,对于不同的通道,通过地址偏移将处理通道与待压缩数据进行一一对应,多通道实现如图10所示.

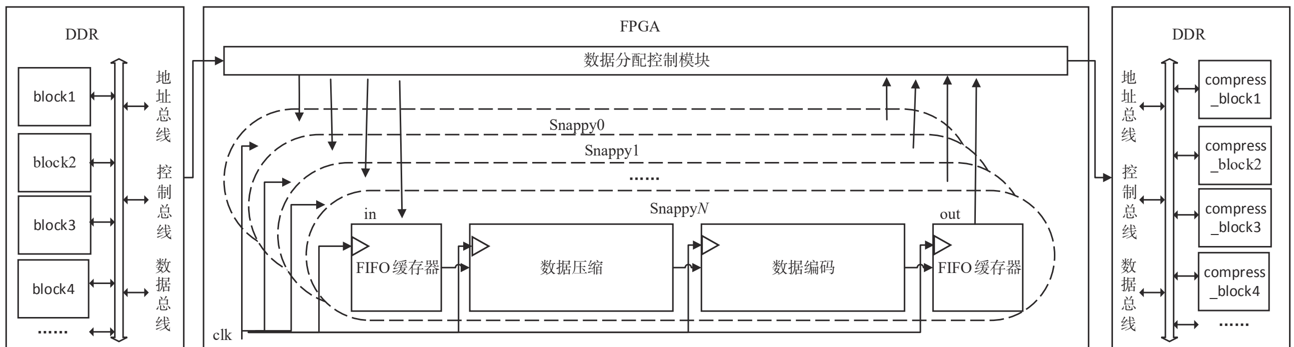

图10　多通道并行结构

在整个结构中,FPGA内包含一个主控制模块,用于将内存的数据进行划分,并计算不同block的大小和偏移地址,然后按照Snappy返回的偏移地址从内存中获取数据,顺序存入到不同通道对应的FIFO中,并启动相应的Sanppy算法进行压缩.压缩结束后,主控制模块将压缩后的数据顺序写入内存指定位置.

主控制模块与DDR的交互只有一个通道,而与算法之间具有N个并行通道,因此,从内存取数据的速度$\mathrm{speed}_{\mathrm{DDR}}$与Snappy的压缩速度speed应该满足式(4).

$$\mathrm{speed}_{\mathrm{DDR}} \geqslant \mathrm{speed}_0 + \mathrm{speed}_1 + \mathrm{speed}_2 + \cdots + \mathrm{speed}_N \quad (4)$$

FPGA与DDR之间通过AXI4通信协议实现物理层面的高速数据传输,支持突发式数据传输,即一次请求响应的交互中,可以传输多组数据,每组数据为256位,突发长度(Burst length)在1和255之间.因此,采用顺序轮询策略进行访存操作的数据存储,结构如图11所示.

图11中$\mathrm{start}_i_\mathrm{addr}_\mathrm{offset}_j$表示第$i$个通道、第$j$组数据的偏移地址,首先,主控制模块顺序从地址FIFO中读取地址,然后AXI4协议顺序从DDR读取数据并存入数据FIFO中,最后主控制模块通过轮询状态机将数据传

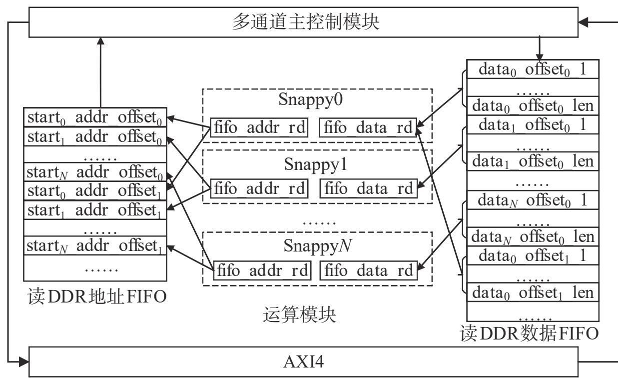

图11 访存数据轮询存储结构

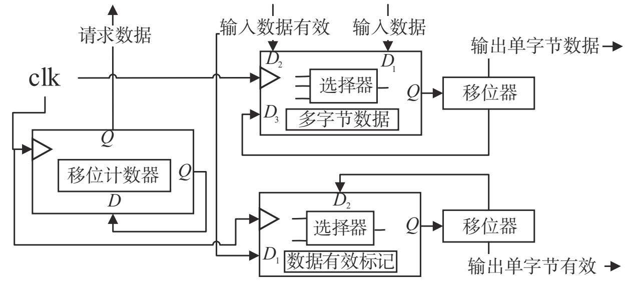

图12 单字节转换硬件实现原理

输到相应Snappy算法中．假设整个FPGA实现的频率为100 MHz，则单位时间内压缩数据量如式(5)：

$$\text{compressData}=N\times 100\text{M(Byte)} \tag{5}$$

内存数据传输量如式(6)：

$$\text{ddrData}=\frac{100\text{M}}{\text{time}_{\text{delay}}}\times \text{len}\times 32\text{(Byte)} \tag{6}$$

其中len表示突发传输长度，$\text{time}_{\text{delay}}$表示从内存传输一次数据的总消耗时间，只需满足式(7)：

$$\text{ddrData}\geqslant \text{compressData} \tag{7}$$

也即式(8)所示：

$$\frac{\text{len}\times 32}{\text{time}_{\text{delay}}}\geqslant N \tag{8}$$

由于采用的是轮询策略，则单个压缩通道在单轮数据输入满足式(9)：

$$N\times(\text{len}+2)\leqslant \text{len}\times 32 \tag{9}$$

前者表示轮询一周需要的时间，len+2表示轮询单个通道需要的时间，状态转换和FIFO启动需要消耗两个时钟，后者表示轮询一次传输的数据量，只有满足公式才能保证单通道内连续的数据输入．最大通道数和突发长度根据实际板卡性能进行可变的扩展，满足上述要求，从而实现FPGA最大数据压缩性能．

3.4 基于移位器的流水线数据转换

FPGA与DDR通信数据位宽是多字节，而Snappy算法是单字节处理，因此，需要对Snappy算法接收到的数据快速转化为单字节进行处理，压缩后的数据同样需要将单字节数据快速拼接为多字节数据输出．

多字节转换单字节的原理是将多字节数据经过移位器处理，从而获得单字节数据，硬件实现如图12所示．

各处理模块以时钟clk为触发条件并行计算，移位计数器是在一定时间内请求数据；移位器将多字节数据每次向右移动8位，输出单字节数据和移位后的数据；多字节数据处理中，通过输入数据是否有效，从而判断接收的数据是移位后的数据或者是输入数据；数据有效标记是与输出的单子节数据一一对应，表示输出数据有效．当有大量的数据需要处理时，每个时钟都有数据输出．

本文中，处理的数据为32字节，RTL实现的并行处理如算法4所示，其中valid表示数据有效，算法分为两个部分，第一部分添加计数器，32个周期获取一组输入，第二部分则是输出赋值，由于FPGA是并行计算，只要有输入时钟，各模块运算不会终止，因此，增加flag标记，表明数据的有效．两部分并行处理，通过算法4的实现，使数据转换能够流水线的连续输出．

算法4 并行数据转换

```
输入:mutilData[255:0] ,validIn
输出:singleData[7:0],validOut
1.   /* 第一部分:计数器*/
2.       count← count + 1
3.       IF(count= =0) THEN
4.           获取输入值
5.       END IF
6.   /* 第二部分:流水线输出 */
7.       IF(validIn= =1) THEN
8.           tempData[255:0] ← mutilData[255:0];
9.           tempFlag[31:0] ←32′hFFFF;
10.      ELSE
11.          tempData[255:0] ← tempData[255:0]>>8;
12.          tempFlag[31:0] ← tempFlag[31:0]>>1;
13.      END IF
14.      singleData[7:0] ← tempData[7:0];
15.      validOut← tempFlag[0];
```

单字节转换为多字节可通过对算法4进行修改，首先输入单字节数据，然后对数据进行移位拼接，最后依据计数器输出．通过对数据转换的高速实现，使Snappy算法从输入到输出形成连续的数据处理链，在流水线的处理中，满足各个部件的满负载运行．

4 实验结果与分析

本文采用的硬件为Zynq-7035，搭载的FPGA芯片包含逻辑单元(Logic Cells)270 K、查找表(LUT)171900、500

(17.6 Mb)存储大小的Block RAM以及343 800个触发器(Flip-flops)等,与FPGA连接的内存DDR存储为1 GB,并且内存数据的数据通信带宽最高为50 Gb/s,硬件板卡有PCIE通道,可将PC机数据传输到板载内存中.

FPGA与DDR之间支持突发式数据传输,而不同突发长度影响请求响应信号,从而导致实际的数据带宽存在差异. 为了满足Snappy算法的性能最优,需要保证数据能够满足Snappy的输入,因此,对FPGA与DDR之间的数据通信,设计不同突发传输大小并进行测试,在不同时钟频率下,结果如图13所示.

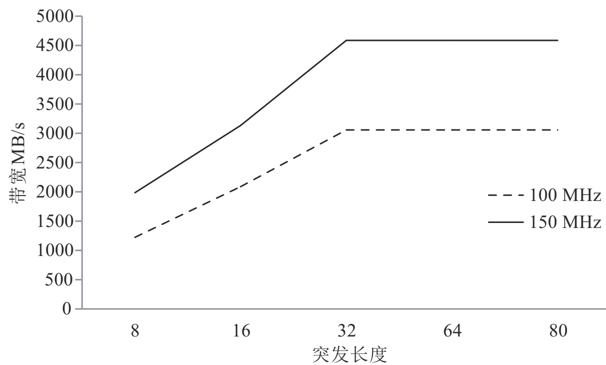

图13　在不同频率下FPGA与DDR通信带宽

从图13中可以看出随着突发长度的增长,通信带宽成正比增长,当突发长度大于32时,通信带宽在测试板卡中趋于平稳,即使在100 MHz的时钟频率下,带宽达到3 057 MB/s,因此,将DDR与FPGA的突发传输长度设计为32,可满足数据压缩的通信传输要求.

在单通道下,对Snappy算法的RTL实现进行综合布线,实现频率为148 MHz,各主要模块所占资源结果如表1所示. 从表1的资源占用可以计算出Snappy核心算法模块占总资源的1.1%,资源占用量较小.

表1　主要模块资源占用

模块	LUT	FF	Slice	Block RAM
数据压缩	1 547	1 551	790	52
数据编码	351	535	156	2
Snappy核心	1 898	2 086	946	54
顶层控制	1 243	2 709	553	13.4
数据通信	723	2 037	558	10

在单通道计算模式下对不同数据进行压缩,压缩速度如表2所示. 从表2可以看出在FPGA上优化实现的算法在性能上基本达到了实际的频率,即采用串-并混合的优化结构达到了与全流水线相同的性能.

以单通道算法的资源使用量为参考,可以计算出FPGA芯片的总资源满足四通道并行的Snappy算法,实现结果的资源占用如表3所示.

表2　不同数据压缩速度

数据类型	压缩速度MB/s
网络数据包	146.21
字典文件	146.51
数据库文件	146.95
FPGA下载文件	147.00
Html文件	147.00

表3　四通道下Snappy算法资源占用

模块	LUT	FF	Block RAM
Snappy0	2 417	3 861	54
Snappy1	2 849	3 862	54
Snappy2	2 793	3 862	54
Snappy3	2 381	3 862	54
顶层控制	1 462	2 198	43.5

由于核心算法增加了3倍,使软件自动化布线的复杂度增加,需要增加资源的占用从而增加布线的成功率,因此,在四通道下资源比单通道下占用较多,但是,与整片芯片相比仍然具有较小的资源占用率,最后实现的频率为139 MHz,压缩性能与不同CPU进行对比,结果如图14所示.

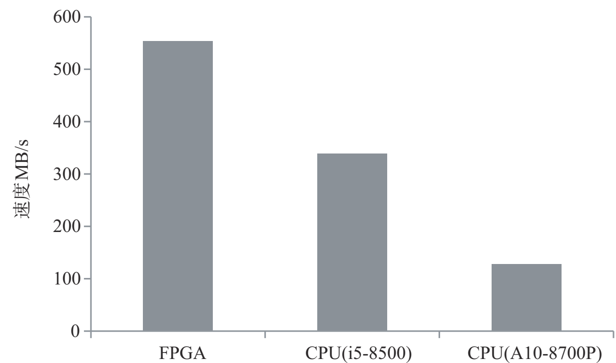

图14　不同芯片性能对比

从图14中可以看出本文实现的结果与CPU相比具有较高的性能优势.

加速比(Speed up)是衡量加速效果的指标之一,指的是程序串行运行的时间与并行运行的时间的比值[13],计算如式(10):

$$\mathrm{Speedup}=\frac{\mathrm{time}_{\mathrm{CPU}}}{\mathrm{time}_{\mathrm{FPGA}}}=\frac{\mathrm{data}/\mathrm{speed}_{\mathrm{CPU}}}{\mathrm{data}/\mathrm{speed}_{\mathrm{FPGA}}}=\frac{\mathrm{speed}_{\mathrm{FPGA}}}{\mathrm{speed}_{\mathrm{CPU}}} \tag{10}$$

其中data为待压缩数据的数据量,通过计算可得FPGA与CPU(i5-8500)的加速比为1.634,具有较高的加速效果.

研究者对于RTL级的Snappy算法研究较少,而LZ4算法与Snappy算法具有类似的压缩方法,因此,将Snappy算法与LZ4算法进行对比,如图15所示.

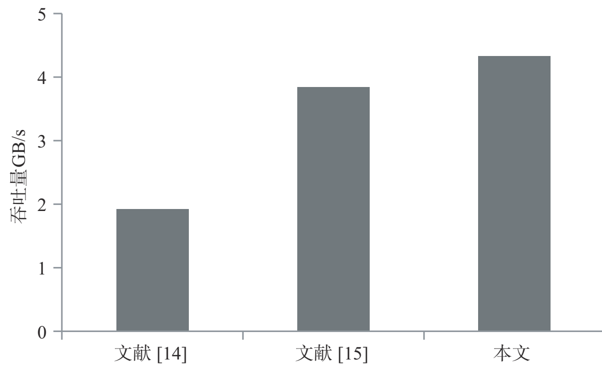

图15 性能对比

图15中不同文献采用的芯片为28 nm,与同类型算法相比,在相同工艺的硬件上,本文实现的压缩算法在性能上具有较大的优势.

Xilinx公司给出了在FPGA上实现Snappy单个执行核的资源结果,与本文实现的单通道结果对比如表4所示.

表4 不同板卡单核算法对比

设计	Xilinx Snappy Streaming	本文
LUT	3 000	1 898
FF	3 500	2 086
频率	300 MHz	148 MHz
吞吐量	260 MB/s	148 MB/s
PRR	0.087	0.078
芯片工艺	16 nm	28 nm

表4中PRR是性能资源比(Performance Resource Ratio),表示单个LUT资源的性能,数值越大,资源利用率越高.由于所使用的芯片工艺差别较大,本文实现的性能频率略低,使得PRR略低于Xilinx的结果,但是在资源占用方面,LUT资源与Xilinx给出的结果下降了36.7%,具有较大的资源优势.

Xilinx在Alveo U200上通过多通道并行实现了高性能的Snappy压缩算法,与本文的结果进行对比,如表5所示.

表5 不同板卡对比

设计	吞吐量	价格	性价比
Alveo U200	8 192 MB/s	99 500	0.082
Zynq-7035	554 MB/s	3 700	0.15

从表5中可以看出本文的实现性能略低,但是综合的性价比要高于U200,表明本文所提方案具有较大的可行性和实际应用价值.

5 结束语

随着信息技术的迅速发展,网络上的数据量呈爆炸式增长,同时,FPGA作为计算设备、网络设备、存储设备得到了广泛应用,为了解决网络负载和数据存储问题,在有限带宽下提高数据发送量,同时增加数据存储量.提出了在FPGA上实现Snappy算法,通过多种FPGA优化方法进行RTL编码,实现的结果占用面积较少、性能较高.通过对算法的扩展,可将算法应用到边缘设备、数据中心等要求性能更高的数据处理领域.

参考文献

[1] 周俊, 沈华杰, 林中允, 等.边缘计算隐私保护研究进展[J].计算机研究与发展, 2020, 57(10): 2027-2051.
ZHOU Jun, SHEN Hua-jie, LIN Zhong-yun, et al. Research advances on privacy preserving in edge computing [J]. Journal of Computer Research and Development, 2020, 57(10): 2027-2051. (in Chinese)

[2] HOSSAIN K, RAHMAN M, ROY S. IoT data compression and optimization techniques in cloud storage: Current prospects and future directions[J]. International Journal of Cloud Applications and Computing, 2019, 9(2): 43-59.

[3] WENCHANG L, BOYUAN G, GUOHUI L. Communication scheduling in data gathering networks of heterogeneous sensors with data compression: Algorithms and empirical experiments[J]. European Journal of Operational Research, 2018, 271(2): 462-473.

[4] AL-HAFEEDH A, CROCHEMORE M, ILIE L, et al. A comparison of index-based lempel-Ziv LZ77 factorization algorithms[J]. ACM Computing Surveys, 2012, 45(1): 1-17.

[5] HASSAN K, ALAA A, VIKTAR U, et al. New modified RLE algorithms to compress grayscale images with lossy and lossless compression[J]. International Journal of Advanced Computer Science and Applications, 2016, 7(7): 250-255.

[6] DATTA A, NG K F, BALAKRISHNAN D, et al. A data reduction and compression description for high throughput time-resolved electron microscopy[J]. Nature Communications, 2021,12: 664.

[7] FANG J, CHEN J Y, AL-ARS Z, et al. Work-in-progress: A high-bandwidth snappy decompressor in reconfigurable logic[C]//2018 International Conference on Hardware/Software Codesign and System Synthesis(CODES+ISSS). Turin: IEEE, 2018: 1-2.

[8] XIA MING, HUANG ZUNKAI, TIAN LI, et al. SparkNoC: An energy-efficiency FPGA-based accelerator using optimized lightweight CNN for edge computing[J]. Journal of Systems Architecture, 2021, 115(4): 101991.

[9] COUSINS D, ROHLOFF K, SUMOROK D. Designing an

FPGA-accelerated homomorphic encryption co-processor [J]. IEEE Transactions on Emerging Topics in Computing, 2017, 5(2): 193-206.

[10] ZHANG B, SANDER P V, TSUI C Y, et al. Microshift: An efficient image compression algorithm for hardware [J]. IEEE Transactions on Circuits and Systems for Video Technology, 2019, 11(29): 3430-3443.

[11] TRUONG N M, AOKI M, IGARASHI Y, et al. Real-time lossless compression of waveforms using an FPGA [J]. IEEE Transactions on Nuclear Science, 2018, 65(9): 2650-2656.

[12] INC Xilinx. Xilinx Snappy-Streaming Compression and Decompression[EB/OL]. (2021-09-22) [2022-02-28]. https://xilinx. github. io/Vitis_Libraries/data_compression/2021.1/source/L2/snappy.html.

[13] 王超, 王腾, 马翔, 周学海.基于FPGA的机器学习硬件加速研究进展[J].计算机学报, 2020, 43(6): 1161-1182.
WANG Chao, WANG Teng, MA Xiang, ZHOU Xue-hai. Research progress on FPGA-based machine learning hardware acceleration[J]. Chinese Journal of Computers, 2020,43(6): 1161-1182. (in Chinese)

[14] LIU W, MEI F, WANG C, et al. Data compression device based on modified LZ4 algorithm[J]. IEEE Trans Consumer Electronics, 2018, 64(1): 110-117.

[15] KIM J, CHO J. Hardware-accelerated fast lossless compression based on LZ4 algorithm[C]//Proceedings of the 3rd International Conference on Digital Signal Processing. Jeju Island: ACM, 2019: 65-68.

作者简介

陈晓杰 男,1993年12月出生,河南武陟人.现为战略支援部队信息工程大学博士研究生,从事可重构计算、信息安全方面的有关研究.
E-mail: 740248499@qq.com

李 斌(通讯作者) 男,1986年12月出生,河南郑州人.现为郑州大学计算机与人工智能学院讲师,主要从事可重构计算、信息安全方面的有关研究.
E-mail: cctvlibin@163.com

周清雷 男,1962年9月出生,河南郑州人.教授、博士生导师.主要从事自动机理论、信息安全和计算复杂性理论方面的有关研究.