文章编号:1671-4598(2024)02-0317-08 DOI:10.16526/j.cnki.11-4762/tp.2024.02.045 中图分类号:TP309.2 文献标识码:A

一种 ARGB 数据无损压缩解压算法的 FPGA 设计

尹 明[1],孙国庆[2]
(1. 青岛科技大学 信息学院,山东 青岛 266100;2. 西安电子科技大学广州研究院,广州 510000)

摘要:提出了一种 ARGB 数据的无损压缩优化算法以及 FPGA 实现方法;为了避免对整个文件的解压和压缩,采用了 Deflate 算法的相关方法对图像按块进行压缩和解压,极大提高了存储器的访问效率;利用了 Deflate 算法对小块进行压缩,发挥了 Deflate 中 LZ77 压缩的 Huffman 压缩技术来优化压缩算法;通过 VIVADO HLS 将算法实现成 FPGA 电路,采用多张图片进行了实际应用,证实了算法的有效性,并分析了其功耗和时序信息。

关键词:Deflate 算法;无损压缩;图像压缩;仿真测试

FPGA Design of Lossless Compression and Decompression Algorithm for the ARGB Data

YIN Ming[1], SUN Guoqing[2]
(1. College of Information Science and Technology, Qingdao University of Science and Technology, Qingdao 266100, China;
2. Research Institute of Guangzhou, Xi'an Electronic Science and Technology University, Guangzhou 510000, China)

Abstract: A lossless compression optimization algorithm for the ARGB data and its FPGA implementation method are proposed. In order to avoid the decompression and compression of the whole file, the related methods of Deflate algorithm are used to compress and decompress the image block by block, which greatly improves the access efficiency of the memory. The Deflate algorithm is used to compress small blocks, and the Huffman compression technology of LZ77 compression in the Deflate algorithm is fully utilized to optimize the compression algorithm. The algorithm is achieved as an FPGA circuit through the VIVADO HLS tool, and the practice application with multiple images proves the effectiveness of the algorithm, and analyzes its power consumption and timing information.

Keywords: Deflate algorithm; lossless compression; image compression; simulation test

0 引言

在 GPU、AI 等芯片设计领域,存储器访问往往是系统性能的瓶颈,提高存储器的访问效率对提升芯片性能的意义重大,其中对颜色缓冲区数据(AGRB)的频繁读写性能的影响很大[1]。从数据压缩算法的角度,通过减少访问颜色缓冲区的数据量来提高存储器的带宽和访问效率[2-3]。

图像压缩是一种通过使用更紧凑的方式来存储和传输图像数据的技术。在进行无损图像压缩时,通常会对图像的各个通道(包括红色通道(R)、绿色通道(G)、蓝色通道(B)以及透明度通道(A))进行处理,这些通道数据通常是基于 AGRB 格式。数据压缩的原理来自于信息内存在的数据冗余。文献[4]提出一种多级流水的 Gzip 压缩设计架构。利用并行处理窗口的设计实现数据压缩的并行处理,通过匹配选择消除了并行处理的相关性[4]。文献[5]使用了计算速度与拟合精度更有优势的自注意力机制模型,同时引入字典编码的优势,以字节对编码(Byte-pair Encoding)的形式生成字典,以模型计算数据集统计信息,辅以算术编码 AC 进行压缩,能够达到吞吐率与压缩比都更高的无损压缩模型[5]。文献[6]提出了一种在 FPGA 平台上实现 Huffman 编码以及高速存入 DDR3SDRAM 存储器的研究方案[6]。文献[7]根据 LZ77 和 LZW 压缩算法的基本原理,通过分析多种影响 LZ77 和 LZW 压缩性能的关键因素,提出诸如双 Hash 函数查找方式、并行匹配处理方法、更有效的 LZ77 数据存储格式、高效数据拼接器以及并行 Hash 函数查找方式等多种加速方法及其硬件结构,并设计出相应的 LZ77 和 LZW 硬件压缩电路[7]。但是 Deflate 压缩算法有缺陷,Huffman 的存储结构也需要进行优化。根据香农提出的信息论中的率失真理论,可以利用这种冗余来实现图像的压缩,即在保持图像质量的前提下,减小图像数据的存储或传输所需的空间或带宽。通过压缩算法和技术,可以消除图像中的冗余信息,包括空间冗余(由于图像中的相邻像素之间的相似性)和统计冗余(由于像素值分布

收稿日期:2023-10-16; **修回日期**:2023-11-23。
基金项目:国家自然科学基金项目(62105176)。
作者简介:尹 明(1980-),男,博士,讲师。
通讯作者:孙国庆(2000-),男,硕士研究生。
引用格式:尹 明,孙国庆.一种 ARGB 数据无损压缩解压算法的 FPGA 设计[J].计算机测量与控制,2024,32(2):317-324.

的规律性），从而实现高效的图像压缩。

对于 GPU 而言，对图像的访问并不限于整张图片。GPU 具有强大的随机访问能力，因此在处理压缩后的图像时，可以仅改变部分区域而无需修改整个图像[8]。为了实现这一点，压缩后的文件需要提供每个压缩区域的位置信息。通过解压缩文件并根据索引找到特定位置，可以修改相应区域的像素数据。为了满足这一需求，开发了一种图像单元的压缩算法，该压缩算法能够对图像的特定区域进行快速压缩和解压缩操作。通过这种压缩算法，能够快速改变图像的特定区域，并且有效减小传输带宽的占用。

1 无损压缩算法的设计思路

1.1 优化 Deflate 压缩算法流程图

Deflate 压缩算法的实现流程如下：

1）通过使用 LZ77 算法，找到输入数据中的重复片段，并用指针和长度来表示这些重复片段。

2）使用哈夫曼编码对指针和长度进行编码，将它们转换为更短的比特序列。

3）在进行哈夫曼编码之前，还会构建一个动态的哈夫曼树，根据输入数据的频率来调整编码方式，以实现更高效地压缩。

4）将编码后的数据和一些元数据一起打包成压缩数据格式，并输出压缩结果。

通过这一系列的步骤，Deflate 压缩算法能够实现对数据的高效压缩，以减小数据的存储空间占用。

将图像的特定区域称为“tile”，并实现了对该区域的压缩和解压缩功能。设置了不同大小的 tile，包括 4×4、8×8 和 16×16 像素的块，对应的字节大小分别为 64、256 和 1 024字节。主要目的是实现对图片区域的压缩和解压缩，所以每次压缩的数据量并不大，仅限于这 3 种字节大小。

针对 Deflate 压缩算法，在进行 LZ77 压缩之后，如果按照传统方式使用 Huffman 算法进行压缩，保存 Huffman 编码表所需的内存空间较大。所以 Huffman 再对 LZ77 压缩的结果进行压缩提升的压缩率不明显。为此，对 Deflate 算法进行了优化，将 LZ77 和 Huffman 算法的执行过程并行化，以提高压缩效率[9]。

本算法的流程如图 1 所示。

使用 v4 头部格式的位图文件。v4bmp 是一种位图文件格式的扩展，它具有更丰富的信息和功能，特别适用于支持 RGBA 颜色空间的图像。包含 AGRB 共 4 个颜色的通道。对文件的压缩和解压基于此文件展开压缩和解压的。

算法开始之后，对 bmp 文件的文件头进行读取并保存，获得了图片的高度和宽度，然后按顺序读取所有的 AGRB 像素数据并保存。再从保存的 AGRB 数据中读取一个图像区域，对其进行差分预测编码之后再进行 LZ77 压缩，并且对此图像区域进行 Huffman 压缩，比较两个压缩算法完成后的文件大小，判断其是否都小于 256 字节，如果都小于 256 字节，那么就将其中较小的写入压缩文件，否则就直接把原始的图像区域写入压缩文件，这样可以获得较小的压缩率。

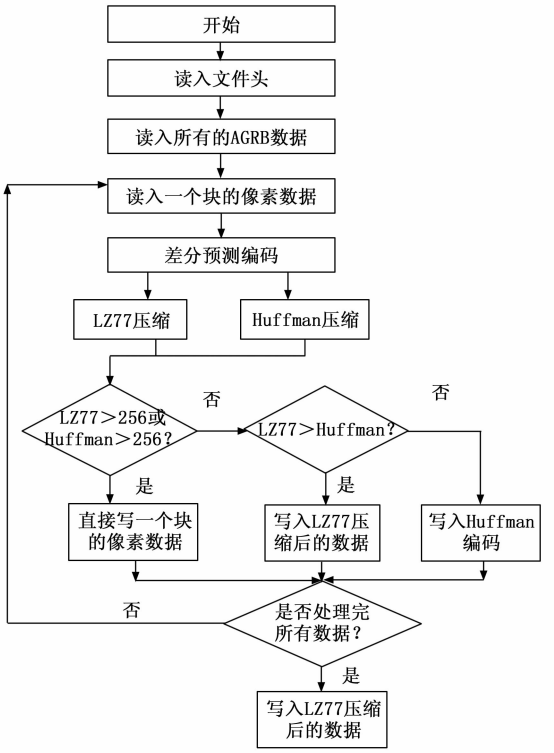


图 1 该压缩算法流程图

压缩后的文件给出了图像的宽度和高度，以便解压时可以完整地还原原始的图像信息。应该注意到压缩后的文件中的 TileCount 和 DataInfo 两个数据，正如之前所说的 GPU 具有强大的随机访问的性能，在某些领域的图像更改不需要改变所有的图像信息。因此，当某个图像区域需要改变时，只需要知道压缩后其所在压缩图像的位置，将其从压缩后的文件中取出后解压并改变像素数据即可。TileCount 和 DataInfo 便是找到压缩后图像数据所在位置的关键信息，他能够起到索引作用。

1.2 差分预测编码

对于给定的图像，可以将其 AGRB 数据进行区域划分，主要介绍按照 8 * 8 像素的单位进行划分。这意味着将图像划分为许多 8 * 8 的小块，如果需要对图像进行修改，也需要按照这个 8 * 8 像素的单位进行变动[10-11]。图像划分格式如图 2 所示。

每个图像的“tile”，也就是 8 * 8 的小块，包含了 64 个像素数据。按照 8 * 8 像素的方式对图像进行读取，这样就可以获取 256 个字节的 AGRB 数据。需要注意的是，这 64 个像素并不是按照顺序排列的，而是按照图 2 所示的方式进行读取。采取这样的读取方式是因为对于一张图片来说，这样的区域内的像素之间具有较高的相关性。因此，获取并压缩这样的数据单元可以达到较好的压缩效果。在得到一个单元的 AGRB 数据之后，进行差分预测编码。

差分预测编码的实现：差分预测编码（Differential Predictive Coding）是一种常用的无损压缩技术，用于压缩数字

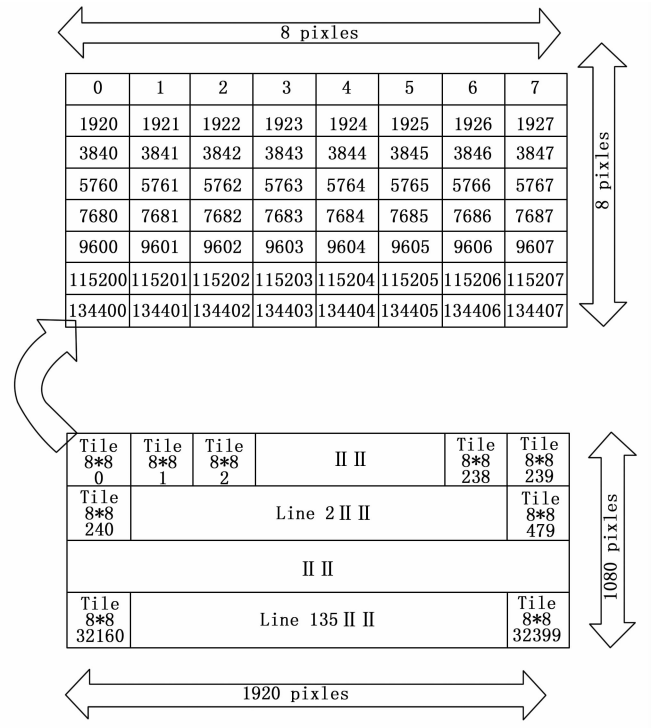


图 2　压缩块示例

信号和图像数据。差分预测编码的基本思想是通过对信号或图像中的样本进行预测，然后用预测值与实际值之间的差异进行编码。它利用了信号或图像中的相邻样本之间的相关性，将差分值进行编码，而不是直接对原始样本进行编码。

观察到这些数据没有重复项，这意味着在压缩算法中存在很少的冗余。为了充分利用数据的特点，采用了差分预测编码的方法。通过差分预测编码，可以对每个像素值与其相邻像素值之间的差异进行编码。这样可以进一步提高数据的冗余性，并提高压缩算法的效率。差分预测编码是一种算法，在该算法中，通过获取一个像素值后，利用预测方法来预测下一个像素的值。采用了使用上一个像素的值作为本次读取像素的预测，并计算本次读取的像素值与上一次读取的像素值之间的差值。这种方式可以有效地减少数据的冗余性，并实现对图像数据的压缩。通过差分预测编码，引入了更多的重复数据，增加了数据的冗余度。然而，这种冗余度的增加使得压缩变得更加容易。利用这些重复的数据，可以更有效地进行压缩，因为重复的数据可以被更紧凑地表示和存储，从而达到更高的压缩率。

1.3　LZ77 压缩算法

Deflate 压缩算法是由 Abraham Lempel 和 Jacob Ziv 提出，是基于字典的无损压缩算法的基础，也是无损压缩领域中的主流算法。LZ77 算法基于字典匹配的原理，通过寻找重复出现的数据序列并用指向其前一个出现位置和长度的指针来表示，从而实现数据的压缩[12-13]。

LZ77 算法会先创建一个字典区和待编码区。LZ77 压缩是一个循环迭代的过程，LZ77 编码器在字典区查找，直到找到匹配的字符串。匹配字符串的开始位置与离字典区右边的距离称为“偏移值”，匹配字符串的长度称为“匹配长度”。LZ77 在编码时，会一直在字典区中搜索，直到找到最大匹配字符串，然后输出一个三元组（off，len，char），off 表示偏移值，len 表示匹配长度，char 为待编码区第一个等待编码的字符。编码输出为（0，0，A），接着，文本序列会向前移动，编码输出为：（1，1，A）。文本序列继续前移一位，编码输出为：（0，0，B）。文本序列继续前移一位，编码输出为：（3，1，B）。文本序列继续前移，直到待编码区为空。所以最后的文本输出为：

（0，0，A），（1，1，A），（0，0，B），（0，0，C），
（2，1，B），（3，1，B），（5，3，A）

经过上面的编码会发现，实际输入的文本共有 9 个字节，而输出共有 7×3＝21 个字节，这就会导致压缩后的数据膨胀，并没有实现压缩的功能。以上只是 LZ77 算法的基本原理，根据以上原理进行了算法的改动。

对于上述的流程可以发现，字典区越长，待编码区的字符在字典区能够被搜索到的概率越高，也就是匹配的概率越高。然而也并非将待编码区的字符全部替换成（off，len，char）合适。因为保存一个 off 要一个字节，保存一个 len 要一个字节，保存一个 char 也要一个字节，这样压缩之后的数据就会变得膨胀。为此，设置一个界限，当 len 的长度大于 2 的时候再进行编码，否则进行原样输出。这样一来，压缩的效率会大大提高。

同时，算法所针对的压缩数据也就是一个图像区域，也就是 256 个字节。倘若再进行上节所讲的字典区为 5，待编码区为 3 的算法压缩，那么很难找到长度大于 2 的编码。为此将 256 字节的一部分作为字典区剩余的另一部分作为待编码区，这样就极大地增加了字符被编码的概率。

获得一个 tile 之后，先将前两个字符设置为字典，然后再到后面的字符中取字符，在前面的字典区查找，如果找得到，那么就保存其 off 偏移值，len 长度，以及字符 char。

在这里，定义以下 3 个 uchar（unsigned char，下同）数组：

ucharcomplbuff [256]；
ucharcompsignbuff [256]；
ucharcompslbuff [256]；

其中 complbuff 储存的是字典区匹配到的字符，compsignbuff 是标志位字符，compslbuff 是存储 len 和 off 的数组。对以上 3 个数组进行解释，如果待编码区的字符在字典区没有找到，那么就将其保存在 complbuff 中，并将标志位符设置为 0，表示字典区查找不到。如果能在字典区查找到该字符，并且能匹配到的长度大于二，那么就对其保存 off 和 len，并将标志符设置为 1，表示能在字典区找到。

1.4　hash 表进行对算法进行加速

根据上述的匹配机制可知，如果直接对字典区和待编码区进行暴力匹配，那么算法的复杂度是 $O(mn)$，它的算

法复杂度为 $O(mn)$，其中 m 是待匹配字符串的长度，n 是目标字符串的长度。在 hash 算法匹配之前，也尝试了 KMP 算法进行加速，但是效果不是很好，而 hash 匹配函数，可以较好地完成算法加速的功能[14-15]。

Hash 加速如下：

依然是使用上面的字符为例，依旧按照上面所述的流程，将字符的前两个保存在 hash 表中，当待编码区的 2 过来时，发现 hash 里面有 2，其位置为 2，那么待编码区的字符 2 直接到字典区的 2 的位置进行匹配，发现匹配长度为 1，不能作为 LZ77 编码的输出，那么将待编码区的 2 保存在 hash 表中，数据前移。

继续重复上述的过程，发现字符在字典区没有找到，那么直接将其插入到 hash 表中。当下一个字符 2 进行编码时，在 hash 表中发现其有位置 2、3，那么程序会执行以下过程，首先在 hash 表中找字符 2 的位置，为 2，那么就从字典区的第二个位置找起。字典区的指针向后移动一个位置，待编码区的指针也向后移动一个位置，直到两个指针位置后面的字符不再相等，就停止移动，并退出字符匹配。同时更新 hash 表。该表表示将待编码区已经被匹配的 2，2 字符也插入到了 hash 表中。

对于已经被匹配的字符 2，2 使用 compslbuff [] 保存其位置 2 和长度 2。并设置标志字符为 1，并保存在数组 compsignbuff [] 中。如果要对其解压，就读取 compsignbuff [] 数组，如果值为 1，就直接读取 compslbuff [] 两个字节即可，读取的第一个字节为位置，第二个字节为长度。

1.5 对 off 和 len 的优化

对于这样的一个字符串，a b a b a b a b a b a b a b a b a b a b a b a b 会发现如果按照上面的方式进行压缩，那么输出的编码在 compslbuff [ ] 的储存为 [0，2] [0，4] [0，8]..... 这就导致每来一个 a，b 字符就会输出一个位置和长度，然而匹配的大小为 2，这就导致 LZ77 算法对这样的一个字符无能为力。为此需要对 off 和 len 重新设计[16]。

做出如下的改变，使用新的变量 jmp 和 lgth。将压缩后的文件写成 ab [0，12]，这个的含义是在字典区的位置为 0 的字符开始，这样的字符执行了 12 次。

这样就对于 off 和 len 舍弃使用新的变量进行压缩和解压缩的表示。

寻找最长匹配，假如说找到这样的一个序列：

字典区：a b c d e f b c d e f g

待编码区：a b c d e f g h

目前最佳匹配串：a b c d e f

根据前面设定的 marethan 的大小，发现最佳匹配串大于了 morethan 的值，那么就将其保存在对应的数组之中。那么待编码区的字符还剩下 g h 两个字符，因为其长度小于 marethan，那么就会被算法保存在 complbuff 之中。最终的 a b c d e f b c d e f g h a b c d e f g 的编码就会变成 a b c d e f [5，5] g h [13，5] g h 。中括号前面的 5 表示距离前面的匹配的字符的距离，后面的 5 表示匹配长度。会发现，源字符的后 7 个字符可以压缩成 a [8，6]。最终的编码为 a b c d e f [5，5] g h a [8，6]，这样压缩后的字符编码会比之前的更小。

之所以会造成这样的浪费是因为算法只关心前面的这个字符，并不关心将这个字符后面的字符作为匹配，为此对算法的另一个步骤就是寻找最长匹配。

当前字符在匹配时产生的匹配长度（简称当前匹配长度）和下一个字符在匹配时产生的长度（简称下一个长度）有以下几种关系：

1）当前匹配长度＞下一个长度：

例：当前上文为 a a aa b bb c cc 当前下文为 a a a b bb，当前最佳匹配串：a a a b bb

下一个上文为：a a a b bb c cca 下一个下文为：a a b bb，下一个最佳匹配串为 a a b bb

在这种情况下，下一个最佳匹配串的长度小于当前最佳匹配串的长度。

2）当前长度＝下一个长度：

例：当前上文为 a a a b bb a a b bbc，当前下文为 a a a b bbc，当前最佳匹配串：a a a b bb

下一个上文为：a a a b bb a a b bbc 下一个下文为：a a b b b c，下一个最佳匹配串：a a b bb c

此时下一个最佳匹配串的长度与当前最佳匹配串的长度相等。

3）当前长度＜下一个长度：

例：当前上文为 a a a b bb a a b bb c d，当前下文为 a a a b bb c d，当前最佳匹配串：a a a b bb

下一个上文为：a a a b bb a a b bb c d a 下一个下文为：a a b bb c d，下一个最佳匹配串：a a b bb c d

此时下一个最佳匹配串的串长大于当前最佳匹配串的串长。

显而易见的是：

①在第一种情况下，输出下一个最佳匹配串将是多余的。

②在第二种情况下，输出任何最佳匹配串都不会造成浪费。

③在第三种情况下，输出当前最佳匹配串将是浪费的。

因此，只需要对第三种情况进行判断即可。在获取当前最佳匹配串的同时，获取下一个最佳匹配串，并进行判断。

为此，将此次的字符放入 hash 表中，并使用下一个字符作为待编码区的起始字符。这样比较此次字符作为待编码区的首字母的匹配长度和下一字符做待编码区首字符的编码长度，这两个长度哪个长就是用哪一个长度保存在压缩的数组之中。

在图 2 所示的代码中，对两个匹配长度进行比较，得到长度最长的字符串。lgth1 是下一字符作为待编码区开头

字符时得到的匹配长度，lgth 是此次字符作为待编码区开头字符时得到的匹配长度。

1.6　优化 Huffman 算法

霍夫曼压缩算法的简要步骤如下：

1）统计符号频率：遍历待压缩的数据，统计每个符号（如字符、字节等）出现的频率。

2）构建霍夫曼树：根据符号频率构建一颗霍夫曼树。频率较低的符号作为叶子节点，频率较高的符号位于树的较低层。通过合并最小频率的节点来构建树，直到只剩下一个根节点。

3）分配编码：从根节点开始，沿着树的路径为每个符号分配唯一的二进制编码。一般而言，向左子树走表示编码为 0，向右子树走表示编码为 1。

4）生成编码数据：遍历原始数据，将每个符号替换为对应的编码，生成压缩后的编码数据。

5）存储压缩数据：将压缩后的编码数据保存到文件或传输给接收方。

6）解压缩时，使用相同的霍夫曼树和编码表，根据编码逐步还原原始数据。

霍夫曼压缩算法通过根据符号频率赋予不同长度的编码，实现了高频符号使用短编码的效果，从而在保证数据完整性的前提下，实现了较高的数据压缩率。它广泛应用于文件压缩、图像压缩、音频压缩等领域。

以下是 Huffman 压缩编码的一个举例：

假设我们有以下字符串需要进行压缩："ABBCCCDDDDEEEEE"。

1）统计符号频率：统计每个符号的出现频率。

'A'：1

'B'：2

'C'：3

'D'：4

'E'：5

2）构建霍夫曼树：根据符号频率构建霍夫曼树。

从最小频率开始构建树：

①合并频率为 1 的 'A' 和 'B'，得到新节点 'AB'：频率＝3

②合并频率为 3 的 'AB' 和 'C'，得到新节点 'ABC'：频率＝6

③合并频率为 4 的 'D' 和 'ABC'，得到新节点 'DABC'：频率＝10

④合并频率为 5 的 'E' 和 'DABC'，得到新节点 'E(DABC)'：频率＝15

得到以下霍夫曼树：

3）分配编码：从根节点开始，沿着左子树走为 0，沿着右子树走为 1，为每个符号分配编码。

'A'：编码为：0000

'B'：编码为：0001

'C'：编码为：001

'D'：编码为：01

'E'：编码为：1

4）生成编码数据：使用编码表，将原始数据替换为对应的编码。

原始数据："ABBCCCDDDDEEEEE"

替换为编码："00000001 _ 00010010 _ 01001010 _ 10101111 _ 11"

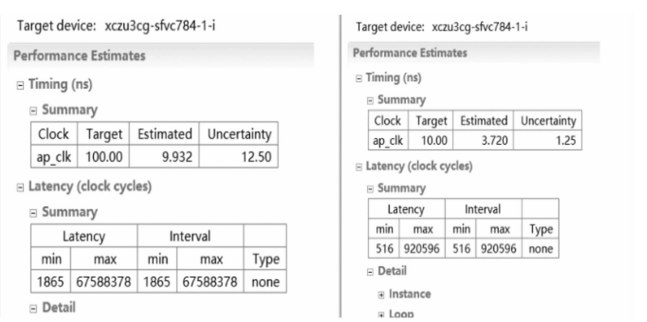
通过霍夫曼压缩算法，原始字符保存下来共需要 15 个字节，现在只需要 5 个字节就可以。现在原始字符串被压缩为较短的编码字符串，从而实现了数据的压缩。在解压缩时，使用相同的霍夫曼树和编码表，根据编码逐步还原原始数据。

主要是改进 Huffman 的存储结构，由于是对某一图像区域进行压缩，因此这个区域的字节数不会太大，对于 8＊8 的 tile 块，设定的是 256 字节。如果根据上面的分析可知，压缩和解压缩都需要得到字符出现的概率，以便构建 Huffman 树和获得 Huffman 编码。但是由于如果保存一个字符，一个出现的概率，那么如果 256 个字节中出现了 N 种字符，那么就需要至少 $2N$ 的字节保存 Huffman 字符频率，如果 N 大于 128，那么编码表都比原字符要大了，这样就造成了浪费。于是设置一个标记位，在统计到 $2N+0.5N>256$ 时，就不在使用 Huffman 压缩算法，从而保证了压缩速度。同时计算出 Huffman 文件的大小，如果压缩后的数据大于 tile 的大小，直接给（＊outputsize）赋予一个大于 tile 的值。

2　压缩算法的 FPGA 设计

2.1　FPGA 平台

该算法实现平台是 Xillinx 的 VivadoHLS2019.1 高级综合工具。接下来主要介绍总体设计结构的实现和优化、功能仿真结果和时序仿真结果。将前几章阐述的算法结构进行实现和优化，通过添加 C 仿真验证算法的正确性，在此基础上进行硬件电路的实现和优化[17-18]。在 HLS 工具中用到的 FPGA 是 Xilinx 的 Zynq UltraScale＋ MPSoC（多处理器系统级芯片）系列（xczu3cg-sfvc784-1-i)[19-20]。将压缩设计的结构直接通过高级综合工具默认的约束进行综合，得到的时延结果如图 3 所示。



(a)压缩单元的HLS默认的约束综合结果　(b)解压缩单元的HLS默认的约束综合结果

图 3　时延结果

2.2 压缩算法的 FPGA 功能仿真

对 VIVADO HLS 生成的电路进行功能仿真。设置输入的块数据为：

Unsigned char input[256]={

1,5,1,9,2,0,3,8,3,3,8,4,4,7,8,111,

8,1,4,4,3,8,9,66,74,47,1,3,7,9,5,1,

32,38,47,43,40,1,1,1,1,1,1,2,2,38,1,5,

1,9,2,0,3,8,3,3,8,4,4,7,8,111,1,6,

6,7,8,1,6,9,66,74,47,1,3,7,32,38,47,43,

40,1,1,1,1,1,2,2,3,7,8,100,102,10,15,222,

33,8,65,60,63,44,52,12,14,18,19,1,5,1,9,2,

0,3,8,3,3,8,4,4,7,8,111,1,6,6,7,8,

1,6,9,74,47,1,3,72,7,3,5,9,5,1,32,38,

47,43,40,1,1,1,1,1,1,2,2,3,7,8,1,5,

1,9,2,0,3,8,3,3,8,4,4,7,8,111,1,6,

6,7,8,1,6,9,4,4,3,8,1,3,7,8,2,7,

3,5,9,5,1,32,38,47,43,40,1,1,1,1,1,1,

1,5,1,9,2,0,3,8,3,3,8,4,4,7,8,111,

1,6,6,7,8,1,6,9,4,4,3,8,9,6,78,47,

1,3,7,8,2,7,3,5,9,5,1,32,38,47,43,40}

压缩后的数据为：

nsigned char refer[256]={

20,118,82,1,5,1,9,1,4,251,71,189,25,8,217,1,

255,2,5,0,254,0,41,241,2,99,187,30,207,247,251,3,

4,255,251,46,216,254,42,210,0,6,250,253,4,252,214,0,

253,7,252,5,248,1,7,29,250,7,39,218,29,219,0,37,

227,255,252,107,152,2,248,7,6,3,254,36,230,239,251,2,

1,6,1,255,27,4,3,62,193,4,4,58,193,44,210,7,

50,243,223,246,253,0,149,3,73,189,36,98,122,94,208,245,

66,254,193,2,2,255,8,39,217,4,0,2,0,0,34,64,

32,0,16,36,85,14,194,15,183,180,65,18,210,1,2,11,

2,28,5,14,2,52,3,36,3,11,2,50,5,48,3,28,

5,65,2,93,2,88,5,27,2,36,3,72,3,98,3,85,

6,46,2,50,5,119,2,49,3,61,2,113,6,112,3,93,

2,138,6,179,2,139,3,36,3,72,2,98,4,85,5,11,

2,50,5,49,3,113,6,112,3,104,2,138,6,139,3,0,

0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,

0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}

首先利用 C 语言进行仿真，得到的结果是正确的，然后进行 FPGA 仿真。

压缩电路的功能仿真结果如图 4 和图 5 所示。从图 4 和 5 可以看出功能仿真完成正确。

2.3 解压缩算法的 FPGA 功能仿真

对于 C 仿真，使用的测试输入数据如下：

Unsigned char input[256]={

20,118,82,1,5,1,9,1,4,251,71,189,25,8,217,1,

255,2,5,0,254,0,41,241,2,99,187,30,207,247,251,3,

4,255,251,46,216,254,42,210,0,6,250,253,4,252,214,0,

253,7,252,5,248,1,7,29,250,7,39,218,29,219,0,37,

227,255,252,107,152,2,248,7,6,3,254,36,230,239,251,2,

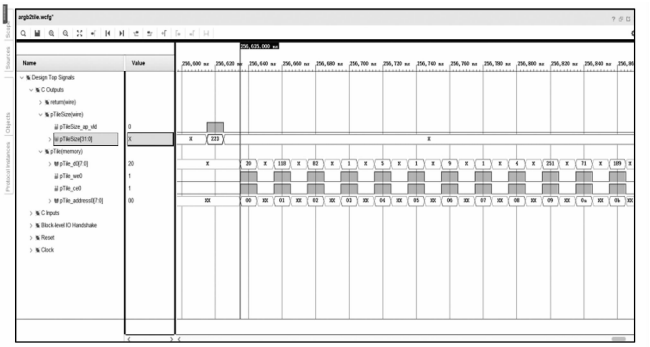


图 4 对测试数据 1 的功能仿真结果

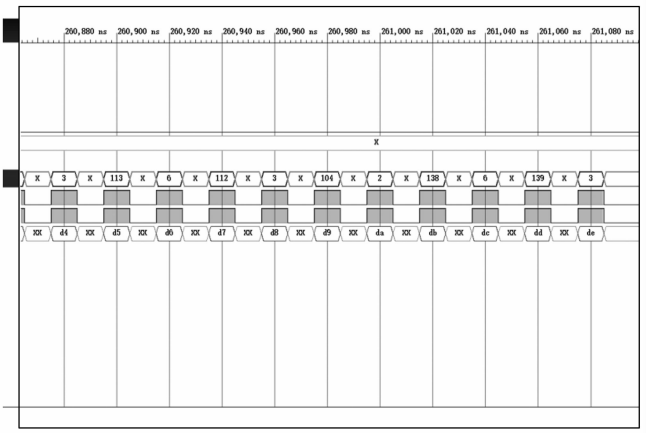


图 5 对测试数据 1 的功能仿真结果

1,6,1,255,27,4,3,62,193,4,4,58,193,44,210,7,

50,243,223,246,253,0,149,3,73,189,36,98,122,94,208,245,

66,254,193,2,2,255,8,39,217,4,0,2,0,0,34,64,

32,0,16,36,85,14,194,15,183,180,65,18,210,1,2,11,

2,28,5,14,2,52,3,36,3,11,2,50,5,48,3,28,

5,65,2,93,2,88,5,27,2,36,3,72,3,98,3,85,

6,46,2,50,5,119,2,49,3,61,2,113,6,112,3,93,

2,138,6,179,2,139,3,36,3,72,2,98,4,85,5,11,

2,50,5,49,3,113,6,112,3,104,2,138,6,139,3}

解压后的数据如下：

Unsigned char refer[256]={

1,5,1,9,2,0,3,8,3,3,8,4,4,7,8,111,

8,1,4,4,3,8,9,66,74,47,1,3,7,9,5,1,

32,38,47,43,40,1,1,1,1,1,1,2,2,38,1,5,

1,9,2,0,3,8,3,3,8,4,4,7,8,111,1,6,

6,7,8,1,6,9,66,74,47,1,3,7,32,38,47,43,

40,1,1,1,1,1,2 ,2,3,7,8,100,102,10,15,222,

33,8,65,60,63,44,52,12,14,18,19,1,5,1,9,2,

0,3,8,3,3,8,4,4,7,8,111,1,6,6,7,8,

1,6,9,74,47,1,3,72,7,3,5,9,5,1,32,38,

47,43,40,1,1,1,1,1,1,2,2,3,7,8,1,5,

1,9,2,0,3,8,3,3,8,4,4,7,8,111,1,6,

6,7,8,1,6,9,4,4,3,8,1,3,7,8,2,7,

3,5,9,5,1,32,38,47,43,40,1,1,1,1,1,1,

1,5,1,9,2,0,3,8,3,3,8,4,4,7,8,111,
1,6,6,7,8,1,6,9,4,4,3,8,9,66,78,47,
1,3,7,8,2,7,3,5,9,5,1,32,38,47,43,40}

解压电路的功能仿真如图 6 和图 7 所示。

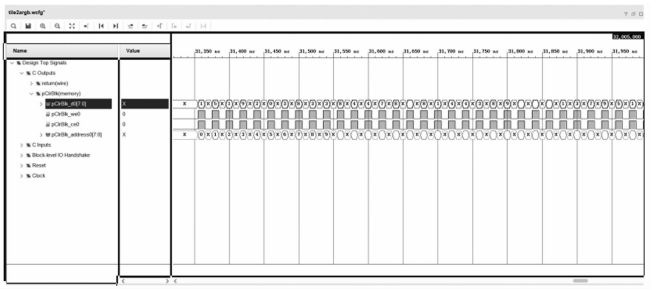


图 6　解压单元对测试数据 1 的功能仿真结果

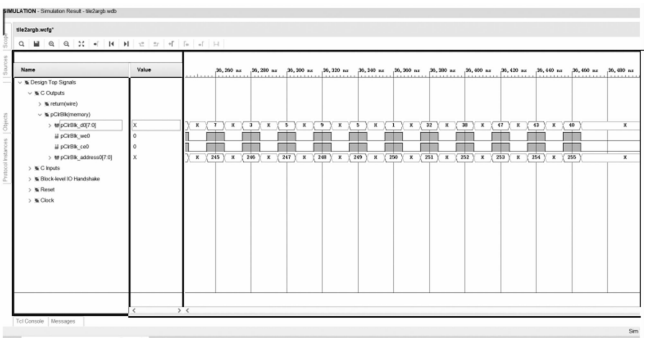


图 7　解压单元对测试数据 1 的功能仿真结果

2.4　时序仿真结果

HLS 导出 RTL 代码后进行了时序仿真，在这里设置的时钟频率是 100 MHz。

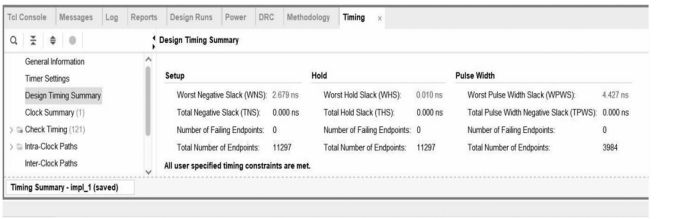


图 8　压缩单元的时序仿真结果

由图 8 可以看出，最坏路径的建立时间裕量为 2.679 ns，那么可以估计出单元的最高时钟频率约为 136 MHz。

同理对于解压缩单元，最坏路径的建立时间裕量为 6.264 ns，同样也可以估计出解压缩单元的最高时钟频率约为 267 MHz。

3　测试结果

使用了 33 张测试图片，在 Linux 系统下对本算法进行了测试，测试的图片和压缩率如表 1 所示。

在 Linux 系统下对 Deflate 算法进行了测试，测试的图片和压缩率如表 2 所示。

接下来对该算法与 Deflate 算法进行相应的比较。基于进行的图片压缩测试结果，本算法在处理那些具有丰富细节的图片时，其压缩性能表现相对较差。然而，当处理那些

表 1　图片和压缩率表

图片	1	2	3	4	5	6	7
压缩率/%	5.434 69	16.012 7	52.500 1	52.327 9	50.472	72.295 1	71.989 8
图片	8	9	10	11	12	13	14
压缩率/%	57.263 8	69.467 7	38.036 4	32.941 3	55.293 1	62.964 8	34.348 8
图片	15	16	17	18	19	20	21
压缩率/%	61.256	34.348 6	42.745 3	26.615 7	47.673 9	46.694 7	10.956 6
图片	22	23	24	25	26	27	28
压缩率/%	22.294 6	26.732 5	27.865 1	57.254 1	75.532 1	58.208 9	55.851
图片	29	30	31	32	33		
压缩率/%	73.260 2	68.298 7	73.521 4	40.002 1	78.951		
平均压缩率/%	50.270 045 45						

表 2　图片和压缩率表

图片	1	2	3	4	5	6	7
压缩率/%	10.554 7	20.454 1	50.547 1	49.983 4	47.532 1	68.954	70.845 1
图片	8	9	10	11	12	13	14
压缩率/%	59.452 1	70.514 5	43.750 8	60.151 7	58.261 4	60.571	40.769 8
图片	15	16	17	18	19	20	21
压缩率/%	60.784 1	40.251	45.776 1	30.203 1	45.820 1	44.001 2	23.971 4
图片	22	23	24	25	26	27	28
压缩率/%	22.294 6	26.732 5	27.865 1	57.254 1	75.532 1	58.208 9	53.851
图片	29	30	31	32	33		
压缩率/%	73.260 2	68.298 7	73.521 4	40.002 1	78.951		
平均压缩率/%	50.270 045 45						

相对平坦且细节不太丰富的图片时，该算法展现出了较为出色的压缩效果。而对于 Deflate 算法来说，其压缩率相对集中，对细节丰富的图片压缩得较好，然而平均压缩率不如该算法。

4 结束语

使用 C 语言实现了一种针对图像的区域压缩算法，并采用 Huffman 算法和 LZ77 算法分别对压缩区域进行压缩。通过选择较小的压缩文件来实现更高的压缩率。为了验证算法的功能可实现性使用 VIVADO HLS 工具进行功能仿真验证算法在不同压缩区域大小下的压缩和解压功能性，从而确保算法在硬件实现中的功能正确性，为图像压缩领域的进一步研究和应用提供了有力支持。

参考文献：

[1] DEEPU C J，HENG C H，LIAN Y. A hybrid data compression scheme for power reduction in wireless sensors for IoT [J]. IEEE Transactions on Biomedical Circuits and Systems，2017，11 (2)，245 - 254.

[2] YU Z，SUN Y，HUANG H L. Linearityanderrordistributionmeasurementofweightedcharge accumulation circuit for computing _ in _ memory [J]. Journal of Measurement Science and Instrumentation，2023，14 (2)：174 - 181.

[3] ZHANG Z J，HEY J，ZHANG B. A time efficient automatic circuit approximation method [J]. IEEE Transactions on Circuits and Systems I：Regular Papers，2020，67 (9)：3047 - 3055.

[4] 陈奕彤. 基于 HLS 的 Gzip 无损压缩硬件设计实现 [D]. 西安：西安电子科技大学，2017.

[5] 刘文顺. 基于字节对编码的无损压缩算法 [D]. 杭州：浙江大学，2023.

[6] 张颖超. 基于 FPGA 的 Huffman 编码并行实现及高速存储系统设计 [D]. 西安：长安大学，2015.

[7] 王超凡. 基于 FPGA 的 LZ77 和 LZW 混合压缩算法的实现 [D]. 南京：东南大学，2016.

[8] 李 冰，王超凡，顾 巍，等. Gzip 压缩的硬件加速电路设计 [J]. 电子学报，2017，45 (3)：540 - 545.

[9] 李 晶. CPU 和 GPU 协同运算下的 Deflate 算法性能加速研究 [D]. 长春：吉林大学，2013.

[10] 孙 圣. 基于 FPGA 的 Deflate 算法核心模块设计 [J]. 软件导刊，2010，9 (5)：63 - 64.

[11] 尤传亮. 一种图像无损压缩算法 JPEG-LS 的 FPGA 实现 [D]. 南京：东南大学，2019.

[12] 吴天钰. WebP 硬件加速解决方案的研究与实现 [D]. 西安：西安电子科技大学，2022.

[13] 马 杰，樊辉锦，宋金禹，等. 一种改进的 LZ77 算法及在车载北斗通信机上的应用 [J]. 舰船电子工程，2021，41 (8)：61 - 64.

[14] LEDWON M，COCKBURN B F，HAN J. High-throughput FPGA-based hardware accelerators for deflate compression and decompression using high-level synthesis [J]. IEEE Access，2020，99 (2)：1 - 10.

[15] Xilinx company. High-level-synthesisvivado design suite user guide [Z]. Xilinx company，2013.

[16] STORRER J A，SZYMANSKI T G. Data compression via textual substitution [J]. Journal of the Acm，1982，29 (4)：928 - 951.

[17] RIGLER S，BISJOP W，KENNINGS A. FPGA-based lossless data compression using huffman and LZ77 algorithms [C] // Electrical and Computer Engineering Canadian Conference on-IEEE，2007 (1)：1235 - 1238.

[18] CHARRI E. An fficentlossless compression scheme for ECG signal [J]. International Journal of Advanced Computer Science and Applications，2017，7 (7)：210 - 215.

[19] DEEPU CJ，LIANY. A joint QRS detection and data compression scheme for wearable sensors [J]. IEEE Transactions on Biomedical Engineering，2015，62 (1)：165 - 175.

[20] KIM DS，KWONJ S. A lossless multichannel bio-signal compression based on low-complexity joint coding scheme for portable medical devices [J]. Sensor，2014，14 (9)：17516 - 17529.

(上接第 316 页)

[13] CHONGCHEAWCHAMNAN M，PATISANG S，KRAIRIKSH M，et al. Tri-band Wilkinson power divider using a three-sectiontransmission-line transformer [J]. Microwave and Wireless Components Letters，IEEE，2006，16 (8)：452 - 454.

[14] 杨 洋. 微波多层超宽带功率分配网络的研究与设计 [D]. 南京：南京理工大学，2018：11 - 13，46 - 62.

[15] 张海兵，王亚松. Gysel 功率分配/合成器的设计与分析方法 [J]. 雷达与对抗，2007 (4)：42 - 46.

[16] HUANG X，WU K L. A broadband and vialess vertical microstrip-to-microstrip transition [J]. IEEE Transactions on Microwave Theory and Techniques，2012，60 (4)：938 - 944.

[17] 张国忠，李 伟. 一种微带线到带状线宽带垂直耦合过渡结构 [J]. 电子测量技术，2016，39 (8)：19 - 21.

[18] 王露莹，刘卫强，郭 超，等. Ku 波段微带到带状线垂直互联结构的研究与应用 [C] //中国电子学会，南京，2021：359 - 361.

[19] 陈顺利，王志刚. 基于 HTCC 收发组件前端无源部件的仿真设计 [C] //中国电子学会，南京，2020：315 - 317.

[20] 王韦皓. K 波段八波束有源相控阵接收组件设计 [D]. 南京：南京理工大学，2021：35 - 44.

[21] 杨雨林. X 波段瓦片式相控阵 T/R 组件微系统的关键技术研究 [D]. 成都：电子科技大学，2018：40 - 43.

[22] LI Z，WANG P，ZENG R，et al. Analysis of wideband multilayer LTCC vertical via transition for millimeter-wave system-in-package [C] // 2017 18th International Conference on Electronic Packaging Technology，Harbin，China，2017：1039 - 1042.