覃子姗,顾　璠,秦晓科,等.基于GPU平台的有效字典压缩与解压缩技术[J].计算机科学与探索,2014,8(5):525-536.

ISSN 1673-9418　CODEN JKYTA8
Journal of Frontiers of Computer Science and Technology
1673-9418/2014/08(05)-0525-12
doi: 10.3778/j.issn.1673-9418.1309017

E-mail: fcst@vip.163.com
http://www.ceaj.org
Tel: +86-10-89056056

基于GPU平台的有效字典压缩与解压缩技术*

覃子姗[1,2],顾　璠[1,2],秦晓科[3],陈铭松[1,2+]
1. 华东师范大学 软件学院,上海 200062
2. 华东师范大学 上海市高可信计算重点实验室,上海 200062
3. 英伟达(NVIDIA),奥兰多 FL 32826

Efficient Dictionary-Based Compression/Decompression Techniques Using GPU*

QIN Zishan[1,2], GU Fan[1,2], QIN Xiaoke[3], CHEN Mingsong[1,2+]
1. Software Engineering Institute, East China Normal University, Shanghai 200062, China
2. Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai 200062, China
3. NVIDIA Corporation, Orlando, FL 32826, USA
+ Corresponding author: E-mail: mschen@sei.ecnu.edu.cn

QIN Zishan, GU Fan, QIN Xiaoke, et al. Efficient dictionary-based compression/decompression techniques using GPU. Journal of Frontiers of Computer Science and Technology, 2014, 8(5): 525-536.

Abstract: Compression techniques are widely used in data storage and transmission. However, due to the inherent sequential nature, most existing dictionary-based compression/decompression algorithms are designed for sequential execution on CPUs. To explore the potential performance improvements of compression and decompression processes using graphic processing unit (GPU), by investigating the techniques of coalescing memory access and parallel assembling, this paper studies two parallel implementations of dictionary-based techniques based on CUDA (compute unified device architecture), stateless compression/decompression and LZW compression/decompression. The experimental results demonstrate that, compared with traditional sequential implementations based on single core, the two proposed approaches can improve the performance of existing sequential dictionary-based compression/decompression algorithms drastically.

* The National Natural Science Foundation of China under Grant No. 61202103 (国家自然科学基金青年项目); the Specialized Research Fund for the Doctoral Program of Higher Education of China under Grant No. 20110076120025 (高等学校博士学科点专项科研基金); the Open Project of SW/HW Co-design Engineering Research Center of MOE under Grant No. 2013001 (软硬件协同设计技术与应用教育部工程研究中心开放课题).

Received 2013-09, Accepted 2014-02.

CNKI网络优先出版:2014-03-03, http://www.cnki.net/kcms/doi/10.3778/j.issn.1673-9418.1309017.html

Key words: graphic processing unit (GPU); compute unified device architecture (CUDA); dictionary-based compression/decompression

摘 要: 压缩技术被广泛应用于数据存储和传输中,然而由于其内在的串行特性,大多数已有的基于字典的压缩与解压缩算法被设计在CPU上串行执行。为了探究使用图形处理器(graphic processing unit,GPU)对压缩与解压缩过程潜在性能的提升,结合合并内存访问与并行组装的技术,基于CUDA(compute unified device architecture)平台研究了两种并行压缩与解压缩方法:基于字典的无状态压缩和基于字典的LZW压缩。实验结果表明,与传统的单核实现比较,所提方法能够显著改善已有的基于字典的串行压缩与解压缩算法的性能。

关键词: 图形处理器(GPU);统一计算设备架构(CUDA);基于字典的压缩与解压缩

文献标志码: A **中图分类号:** TP311.52

1 引言

数据压缩被广泛应用于不同领域,这些领域通常涉及到大规模数据的处理与存储,或者涉及到远距离的网络数据传输。数据压缩能够有效地降低数据的存储空间,并且可以减少网络的传输时间。如图1所示,由于采用了串行的方法,传统的数据压缩与解压缩过程非常耗时,成为系统运行的瓶颈。如果能够有效利用目前硬件平台的并行能力,降低压缩与解压缩所需的时间,系统性能(例如视频解码速度、实时网络响应时间等)将大大提高。

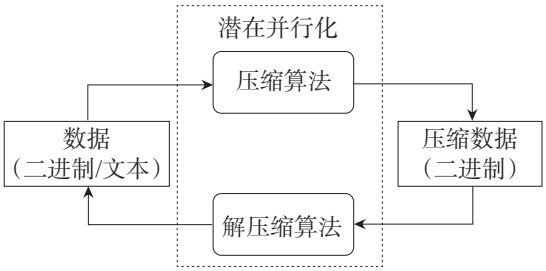


Fig.1 Conventional compression/decompression flow

图1 传统的压缩/解压缩流程

为了获得比串行压缩与解压缩算法更好的性能,许多常用的压缩与解压缩工具都已经采用了并行的实现方法。例如,WinZip和Winrar已经使用块级并行对多核系统进行优化,以实现数据级并行(data level parallelism, DLP)和线程级并行(thread level parallelism, TLP)。这些工具背后的基本思想是将输入数据分成多个小块,再使用不同CPU核对每个数据块进行并行处理。

图形处理器(graphic processing unit,GPU)支持大规模的并行计算,被广泛用于图形图像的加速处理中。在当今的桌面系统与移动终端上,GPU已经成为了基本配置。因此如何利用GPU的计算能力来加速现有的压缩与解压缩算法成为了研究热点。为了简化GPU的编程模式,NVIDIA开发了统一计算设备架构(compute unified device architecture, CUDA)平台,使得CPU与GPU能够协同处理。然而因为基于GPU的CUDA是在单指令多数据(single instruction, multiple data, SIMD)模式下实现的,所以将常规的块级并行压缩技术直接移植到GPU上并非易事。

本文探究了基于字典的两种无损压缩技术:有状态(stateful)的压缩与无状态(stateless)的压缩。这两种压缩技术的区别在于在压缩或解压缩的过程中是否需要维持相关的内部状态(即字典)。无状态的压缩/解压缩算法,例如简单的字典压缩方法[1]、哈弗曼编码(Huffman coding)等,是基于预先计算好的字典进行数据的压缩与解压缩。而有状态的算法,如LZW(Lempel-Ziv-Welch)和算术编码,在压缩/解压缩过程中需要动态地构建字典(或相似的数据结构)。无状态的压缩/解压缩技术明显更加简单和快速,而有状态的压缩/解压缩技术虽然较慢,但往往能产生更好的压缩结果。

由于基于字典的压缩算法需要频繁地访问字典,而通常字典又是存放在全局内存中,这就使得压缩过程的全局内存访问负载很大。高代价的非合并内存访问(non-coalescing memory access)模式带来的

延迟损失，将会使得在单个warp（warp是GPU执行程序时的调度单位）下利用多线程处理不同数据块带来的改进效果优势无存。传统的块级并行处理方式往往会带来不好的内存访问模式。除此之外，每一个并行的压缩块采用的字典内容不同，由于CUDA平台的SIMD特性，在遇见条件分支的时候将使得等待控制分支化解（control divergence resolution）的时间变得很长，并行的GPU计算将串行执行，大大影响GPU的性能。因此开发出一种能充分利用GPU的计算能力和内存带宽，而不牺牲性能的高效压缩技术是当前的一大挑战。本文提出了一种有效的GPU并行压缩/解压缩框架，这种框架可以应用到有状态和无状态压缩技术中，并结合了常规的块级并行方法与GPU体系结构的特性优势。

本文组织结构如下：第2章介绍了并行压缩与解压缩算法的相关工作；第3章提出了一个GPU友好的并行压缩/解压缩框架及其相关技术；第4章详细介绍了利用前述框架的基于字典的压缩技术和LZW的实现；实验结果在第5章进行了介绍；第6章对本文进行了总结。

2 相关工作

压缩/解压缩技术的并行化已经被广泛地研究。Franaszek等人[2]提出了一种基于块引用压缩算法和字典协作构建的并行压缩技术，采用多个压缩器共同构建出字典。虽然该方法对压缩性能有着巨大的改善，但是在压缩率上不如LZ77方法。Nagumo等人[3]提出了基于两个静态字典压缩策略的并行化算法。Smith等人[4]引进了一种基于文本替代的数据无损压缩并行算法。这种算法可以很容易地实现为n-MOS电路。Storer等人[5]描述了一种针对Nagumo方案的大型的并行体系结构。Henriques等人[6]也提出了一种并行算法和体系结构来实现LZ数据压缩技术。Lee等人[7]研究了怎样将数据压缩技术应用到科学数据集的迁移中。Farach等人[8]从一些非常基础的问题考虑并行性，如字典匹配和字符串压缩。这也对并行多媒体数据压缩产生了相当大的影响。Shen等人[9]对这个研究领域进行了调研综述。

在解压缩时，并行化方法通常可以增加解码的带宽。根据底层编码系统的不同，该领域已有的相关工作可以被大致分为两类：第一类是基于定长的编码。对于这一类压缩技术而言，因为连续编码的边界是固定的，所以并行解码的过程相对非常容易。相对于变长编码而言，定长编码的唯一缺陷在于压缩效率不高。第二类方法处理了变长编码的并行解码问题[10]的并行粒度，这些方法可以进一步划分为块级并行和编码级并行。第一类方法常常将压缩的数据组织成块，并且将块标识符存储在文件头或索引表中。利用这种方法，并行解码器可以被应用在每一个块上。这些方法对于媒体数据如JPEG、MPEG-2[11-12]或H.264[13]而言是十分适合的，因为大多数压缩的媒体数据已经按块结构（宏块）表示出来了。然而这些方法的缺点在于，常用的块标识技术如块头、字节对齐或索引表等方案会引入一些开销，使得压缩效率降低。Klein等人[14]利用自动块边界检测方法详述了这一问题。基本思想是每个解码器对应不同的块起始点，而这个起始点也与准确的边界值足够接近。正确的编码边界的检测是通过检验重叠区域上处理器的输出来实现的。利用这种方法，可以有效地避免存储部分的块边界信息。

编码级别的并行目前也可以用在多解码器解压缩多个连续压缩数据块的过程中。与之前的方法不同，这种方案下的并行解码十分有挑战性，因为不知道下一个编码将从何处开始。目前已有多种方法能够很好地解决这个问题。比如Nikara等人[15]在输入缓冲的每个可能的位置上，采用多个推测的并行解码器进行操作，并只输出正确的解码结果。但这种方法对于大的输入缓冲区会引入显著的硬件开销。Qin等人[16]对这个问题则引入一个位置协议来预测不同解码器要处理的代码起始地址。然而，这些编码级别的方法需要解码器间的深度同步，目前的GPU还没有相关的有效支持。在这种情况下，适当的结合块级别和编码级别的并行可能会更有效地解决这个问题。

与本文工作相似，Wu等人[17]探究了多个基于CUDA平台的压缩算法。他们利用块级别的并行策

略加速CUDA上的LZ77。然而由于该方法采用了代价较高的非合并内存访问模式,从而在多核系统的CUDA上的改进受限。而且该方法并没有提供足够的关于压缩数据块的组装过程的信息,由于设备内存和主机内存的低带宽连接,这可能是另一个瓶颈。

3 基于GPU的有效压缩/解压缩框架

本文提出了一个基于GPU的有效的压缩/解压缩框架。如图2所示,本文的并行数据压缩/解压缩流程主要包含以下4个步骤。

压缩步骤1:文件切割与压缩。原始文件被切分成多个数据块,通过相应的压缩算法使用GPU进行并行压缩处理。

压缩步骤2:数据块组装。压缩后的数据块被组装到一个单独的内存区域(即单个文件)中。

解压缩步骤1:数据拆分。在解压缩时,根据压缩文件中数据块的起始位置信息,将压缩文件重新拆分。

解压缩步骤2:解压缩与拼接。通过相应的解压缩算法使用GPU进行并行解压缩处理,得到解压缩后的数据块,然后经过数据块拼接,得到原始文件。

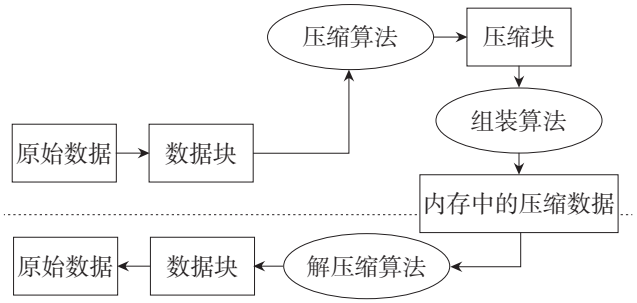


Fig.2 Parallel compression/decompression workflow

图2 并行压缩/解压缩工作流

压缩解压缩过程中数据移动的详细信息如图3所示。原始文件拆分成 k 个大小相等的部分,并行压缩得到新的数据块(图中阴影部分),将得到的新的数据块组装即可得到压缩文件。拆装压缩文件即可还原出源文件。

基于字典的压缩通常被认为在本质上是串行的。但是通过对传统基于分块的并行方法的分析,在基于字典的压缩过程中,只有很小的一部分是不

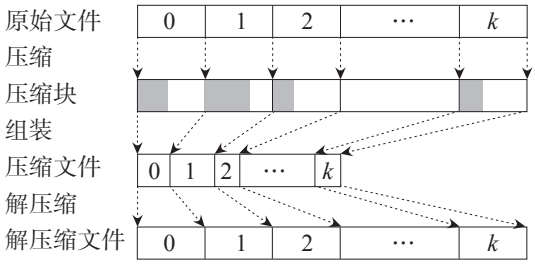


Fig.3 Data movement in Fig.2

图3 图2中的数据移动

能并行化的。在整个流程中,唯一的串行步骤是组装过程。为了使得整个工作流高效,需要解决以下两个问题:

(1)怎样使得多个编码器/解码器在GPU上高效地并行工作。

(2)怎样将该“串行”组装过程并行处理。

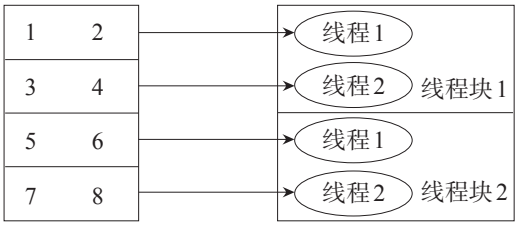
3.1 高效内存访问

一般说来,让多个编码器/解码器在CUDA平台上高效地工作是十分困难的,因为处理的数据块与使用的字典内容不同,压缩/解压缩线程的行为可能会不同。例如,由于不同线程处理不同数据块,线程间内存访问的地址不连续,这将导致多次访问内存,引起内存访问时间过长,即非合并内存访问的问题。

图4(a)给出了传统块级别并行性情况下的内存访问模式。在这个例子中有4个数据块,每个数据块都包含两个符号(比如块1包含1、2两个符号)。两个线程块中的4个线程被用于并行压缩这些数据块。线程1和线程2获得的数据地址是不连续的,因此不能同时访问内存区域。在这种情况下,多次地访问内存会使得压缩算法效率下降。

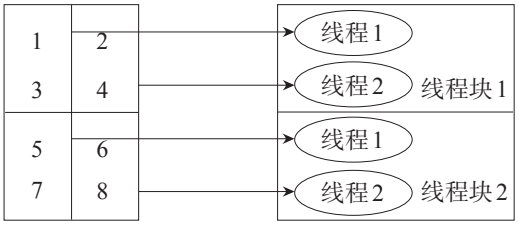
针对这个问题,本文采用了“垂直划分”的思想来避免非合并内存访问的情况,提高了访问效率。如图4(b)所示的改进方法,将数据块1~数据块4进行“垂直”划分,线程1与线程2就可以同时访问块1和块2中的内容,采用这种方法,内存访问的效率将大大提高。当划分恰当,所有输入数据访问全局内存时都是连续的地址,合并内存访问在并行处理过程中,可以获得更高的吞吐量。

在解压缩时也可以采用相同的技术,尽管有时候效果并不是很明显,但是当线程块中的所有线程



(a)General memory access scheme

(a)普通内存访问模式



(b)Vertical memory access scheme

(b)垂直内存访问模式

Fig.4 Different parallel memory access schemes

图4 不同的并行内存访问模式

都同步后，写操作也可以有效地联合。目前GPU支持同一个线程块内的同步，因此这种写操作的联合非常容易实现。

3.2 并行组装

在块级别的压缩过程中，组装过程被认为是所谓的“串行”工作，通常由CPU完成。然而由于数据从GPU设备内存(device memory)传输到主机内存(host memory)上的代价很高，基于CPU的组装过程十分低效。再者，在传输时，设备内存中的压缩数据还未组装，基于CPU的组装方法需要将所有GPU设备缓冲区的内容传递回主机。因为缓冲区并未完全利用，在实际过程中，发送回主机的信息量总会比最终的压缩数据大得多，导致性能低下，所以在GPU上执行完成这个组装工作是十分有优势的。

在基于块的压缩方法中，压缩结果的组装过程主要包含以下两个任务：

(1)对所有的压缩数据块进行偏移量计算。

(2)将压缩数据块放置到正确的压缩文件对应的位置。

显然，第一个任务必须由单线程执行，不过一旦得到所有偏移量后，第二个任务在本质上就是一个并行任务。因此，采用两个核(kernel)来组装压缩的数据块。第一个核只采用一个线程，根据数据块的大小一个一个地计算每个压缩数据块的偏移量。第二个核上采用大量的线程和线程块来执行数据迁移与组装。每个线程块负责对一个压缩数据块执行操作。在这种方式下，压缩数据块的组装就在设备内存中被高效地完成。之后，只需要将组装好的压缩数据传回到主机上即可。

3.3 块大小对压缩/解压缩技术的影响

对于基于字典的压缩/解压缩技术而言，如果块过小，压缩率将因为字典很小受到很大影响；如果块过大，完成压缩/解压缩所需的线程数量会较少，不能完全利用GPU的能力。因此，块粒度决定了基于字典的压缩/解压缩技术的效率。在实际实现时，由于有很多分支，而且每个数据块都是不同的，分歧不可避免。然而，并行执行大大减少了压缩/解压缩所需的内存访问时间。尽管基于字典的串行方法使用了一些“指令级别并行”(instruction-level parallelism, ILP)，在遇到较长的内存访问延迟时，这种方法还是会使整个压缩/解压缩过程进入停滞状态。对于GPU的流水线处理而言，频繁的零消耗warp交换将降低内存延时，并且整体执行时间会减少。5.2节将分析块大小在LZW方法中的影响。

4 两种基于字典的方法的并行化实现

本章基于第3章提到的高效内存访问、并行组装、构建合适的块大小的思路，介绍了两种基于字典的不同类型压缩算法，即基于字典的无状态压缩与基于字典的有状态的LZW压缩，并且介绍了在GPU环境下的相关并行技术。在接下来的讨论中，用“符号”(symbol)这一术语来指代一个未压缩的位串，用“编码”(code)来指代利用压缩算法处理“符号”得到的压缩结果。

4.1 基于字典的无状态压缩

对于传统的基于字典的有状态压缩方法，N 位长的输入数据会被分成 n 个，每个长度为 w 位的符号，即 $N=n\times w$ 。其基本思想就是将频繁出现的符号用通过字典编码过的符号进行替代。字典中经过编码的符号长度小于原始的符号，因此可以达到相

应的压缩效果[18]。字典在整个压缩与解压缩过程中扮演着非常重要的角色。在压缩过程中,原有的文件需要进行两次遍历:第一次遍历将会获得原始位串的频率统计信息,高频率的符号将会用以生成相关的字典;第二次遍历将会基于生成的字典对原始数据进行替代,生成压缩文件,在这次遍历中,所有的符号将会被添加一个压缩标记位,用以指示当前的符号是否在字典中出现。图5给出了在第二次遍历过程中的编码方法。如图5(a)所示,在字典中没有对应的符号,其符号的压缩标记位将被置为“0”。如图5(b)所示,如果符号在字典中出现,其符号的压缩标记位将被置为“1”。

压缩标记位 (“0”)	未压缩符号 (w)

(a)Uncompressed symbol

(a)未压缩符号

压缩标记位 (“1”)	字典索引 ($\text{lb}\,d$)

(b)Symbol compressed with dictionary index

(b)用字典索引压缩过的符号

Fig.5 Coding formats in stateless dictionary-based compression

图5 基于字典的无状态压缩中的编码格式

图6展示了一个基于字典的压缩方法的实例。这个例子使用两个条目的字典,最频繁出现的符号被存储到字典中。在压缩时输入数据中的这些符号被词典索引所代替,压缩的数据不仅包含编码,还包含这个字典。在这个例子中,基于字典的压缩实现了97.5%的压缩率。

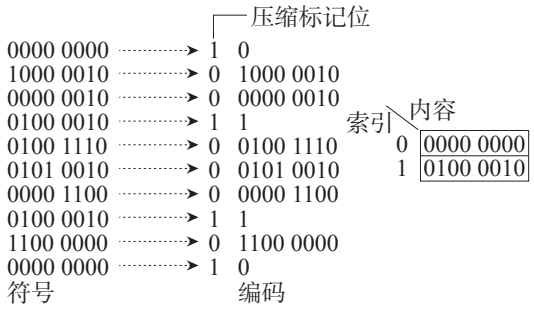


Fig.6 Stateless dictionary-based compression

图6 基于字典的无状态压缩

在GPU的并行环境下,对于基于字典的无状态压缩本文主要考虑的是数据块的划分大小。结合3.3节关于块大小对压缩效率的讨论,基于字典的无状态压缩的压缩效果依赖于符号长度 w 和字典大小 d 。对于一个给定的字典大小,较大的字符长度使得压缩符号的数量减少,同时匹配率降低。另一方面,较短的字符长度会使得有更大概率获得重复的符号,尽管这样会导致待压缩符号的数量增加,但可能会使压缩更加有效。一旦字典建立,压缩算法就仅仅是一个符号匹配过程,而解压缩过程就可以以查找表的方式来实现。同时,高效的内存访问也是基于字典的无状态压缩过程中需要着重考虑的问题,详见3.1节。

4.2 基于字典的LZW压缩

LZW[17]是一个通用无损数据压缩算法,它是LZ78算法的一个改进。下面结合第3章提到的高效内存访问、并行组装和合理设置块大小的思想,阐述如何在基于CUDA的GPU平台上提高LZW压缩/解压缩算法处理文本文件时的性能。

4.2.1 LZW压缩与解压缩

本文将标准的256个ASCII字符作为符号,每个符号占用1 Byte。LZW压缩方法将文本符号串映射成数字代码。起初,字符串中可能出现的所有符号都要一个代码。符号串与代码间的映射存储在字典中。字典中的每个条目都有两个属性:key和code。压缩时,代码表示的符号串被存储到key中。LZW压缩算法采用LZW规则来压缩符号串,它重复寻找未编码部分的最长前缀p。如果在剩余的字符串中有下一个符号c,那么就将pc作为下一个key存入字典,并记录一个新的code信息。

LZW解压缩过程是从压缩文件中一个接一个地获取代码,并按照代码表示的文本进行替换。解码一个压缩文件前需要预先了解字典的初始配置,剩余的字典条目在解压缩过程中可以进行重建。

4.2.2 基于GPU的LZW并行实现

原始的LZW压缩与解压缩算法只使用了一个字典,并且字典在压缩和解压缩过程中会分别进行构建和重建。换言之,这两个过程都取决于动态生成

的字典。由于压缩/解压缩算法是内在串行连续的，每个块只可以用一个线程进行处理，而且每个线程都需要一个较大的内存空间来存放字典，因此本文没有使用共享内存来加快内存访问时间，所有的内存操作都在全局内存中进行。对于每一个大小为 k 的文件块，单位为Byte，分配大小为 k 的内存空间来存储原始文件，$2k$ 的内存空间来存储压缩文件（假设压缩率小于2）。对于压缩而言，需要 $16k$ 的内存空间来存储以树型数据结构组织的压缩字典。对于解压缩来说，需要 $8k$ 的内存空间来重建字典。也对所有数据块采用一个全局的整型数组来存储压缩文件块的大小。所有数据结构都存储在全局内存中。

在数据块压缩过程中，压缩单元为1 Byte。最初需要8位来对初始配置进行编码，因为这里有对应于256个ASCII字符的256个条目（从0开始）。LZW压缩过程是边压缩边生成字典的，编码位数可能随着内容增长而增加，因此初始配置使用8位进行编码后，剩下的内容就需要9位或以上的位数进行编码。以此类推，每当位数不够用时，需要再增加一位编码位来完成编码。数据块压缩完成后，压缩数据块的大小可能不是8位（1 Byte）的整数倍，因此需要填充一些0，使得字节对齐。

4.3 压缩与解压缩算法框架

结合以上两种基于字典的压缩算法，提出了一个通用的压缩与解压缩算法框架。基于字典的无状态压缩和基于字典的LZW压缩方法都可以基于这一框架进行压缩。

算法1 Dictionary-based Compression

Input: Original file F

Output: Compressed file F'

Begin

CudaMemcpy(F, cudaMemcpyHostToDevice)；

$\{f_1, f_2, \cdots, f_n\} \leftarrow$ Cut(F)；

$\{(f_1', dic_1), (f_2', dic_2), \cdots, (f_n', dic_n)\} \leftarrow$

GPUComp($\{f_1, f_2, \cdots, f_n\}$)；

DicSet $\leftarrow \{dic_1, dic_2, \cdots, dic_n\}$；

OT $\leftarrow$ GPUComputeOffset($\{f_1', f_2', \cdots, f_n'\}$)；

$F' \leftarrow$ GPUAssem(OT, DicSet, $\{f_1', f_2', \cdots, f_n'\}$)；

CudaMemcpy(F', cudaMemcpyDeviceToHost)；

return F';

End

压缩过程如算法1所示，首先CUDA将原始文件从主机内存拷贝到设备内存上，使用Cut函数将文件等分切割成 n 个文件块后，GPUComp函数将这些文件块在GPU上进行并行压缩。在这里GPUComp函数可以是基于字典的无状态压缩，或是LZW压缩。由于压缩后的文件块大小不一，需要记录每个文件块在合并后文件中的位置信息。本文使用GPUComputeOffset函数计算每个小文件的偏移，得到一个偏移表OT。GPUAssem函数把偏移表、字典集和压缩的文件块组装成压缩文件 F'。压缩后的文件最终将被拷贝到主机内存。

解压缩可以看做压缩的逆向过程。如算法2所示，首先压缩文件从主机内存拷贝到设备内存上，通过Deassem函数拆装得到偏移表、字典和压缩文件块。GPUDessem函数结合字典，将 n 个这样的压缩文件块还原成 n 个大小相等的文件块。由于得到的文件块是等长的，不需要使用偏移表和字典就能使用GPUAssem函数直接进行组装。解压缩后的文件最终将被拷回主机内存。

算法2 Dictionary-based Decompression

Input: Compressed file F'

Output: Original file F

Begin

CudaMemcpy(F', cudaMemcpyHostToDevice)；

(OT, D, $\{f_1', f_2', \cdots, f_n'\}$) $\leftarrow$ Deassem(F')；

$\{f_1, f_2, \cdots, f_n\} \leftarrow$ GPUDecomp($\{f_1', f_2', \cdots, f_n'\}$, D)；

$F \leftarrow$ GPUAssem(NULL, NULL, $\{f_1, f_2, \cdots, f_n\}$)；

CudaMemcpy(F, cudaMemcpyDeviceToHost)；

return F;

End

5 实验

为了验证所提出的基于字典的并行压缩框架的有效性，基于C语言与CUDA平台开发了相应的基于字典的无状态压缩及LZW压缩算法，并实现了CPU串行程序与GPU并行程序。本文通过使用Gtgraph

Table 1 Dictionary-based compression/decompression results

表1 基于字典的压缩/解压缩

File	File size/MB	Compression ratio/(%)		Compression time/ms		Decompression time/ms	
		CPU	GPU	CPU	GPU	CPU	GPU
text1	1.98	63	63	25.2	19.9	18.5	25.0
text2	15.85	63	63	171.0	29.6	139.0	42.3
text3	100.00	63	63	1 113.0	110.8	913.0	148.3

工具[19]生成了3个不同大小的文本文件,并对其进行了压缩与解压缩实验和结果分析。实验运行环境为使用NVIDIA Tesla C1060GPU的工作站,其操作系统为Linux,CPU型号是Intel Xeon E5462,主频为2.80 GHz。

5.1 基于字典的无状态压缩

在基于字典的无状态压缩/解压缩实验中,采用了16个条目的字典以及8位的符号,每一个符号块的大小设为16k。如前所述,每个线程块被指定压缩/解压缩一个符号块,在符号块中连续的线程处理连续的符号(为了高吞吐量,以4 Byte为分组大小)。对于压缩代码块的并行组装,采用128个线程块,每个块中使用256个线程。

基于字典的压缩/解压缩结果如表1所示。第2列显示的是这次实验中使用的原始文件的大小。第3列和第4列表示的是分别使用CPU和GPU时的压缩率。可以看到输入数据的分块情况对压缩率几乎没有影响,这是由于实验中字典是独立建立的,相关索引信息基本可以忽略。第5列和第6列显示的是压缩时间。可以看到相比于CPU,在原始文件较大的情况下,GPU可以获得10倍左右(即1 113/111)的加速比。值得注意的是,在原始文件较小的情况下,GPU可能比CPU消耗的时间略长。在解压缩时间中也可以看到类似的情形。这主要是由于GPU压缩的过程需要一个较长的准备时间,例如数据从主机内存传输到GPU的设备内存时间。每个GPU的线程在获得第一个数据前需要等待一段相当长的时间。因此,在数据输入较小的情况下,这个准备阶段占据总时间的大部分。

为了更好地理解表1的时间消耗情况,将压缩/解压缩的时间划分为4个部分。如表2所示,H2D是在设备内存上建立数据结构的时间,EXE是GPU真正的运行时间,ASSMB是并行组装算法消耗的时间,D2H是从GPU将结果写回主机的时间。最后一列显示的是总体运行时间。在表2中,前3行展示了压缩过程中各阶段的时间构成情况。后3行展示了解压缩过程中各阶段的时间构成情况。可以看到,主机与设备内存间的数据传输的代价是非常高的。对于大文件(如text3),这部分的时间有可能比实际的压缩/解压缩时间更长。

Table 2 Breakdown of compression/decompression time in Table 1

表2 表1中压缩/解压缩时间的分解

ms

File	H2D	EXE	ASSMB	D2H	Total
text1c	0.9	18.1	0.2	0.7	19.9
text2c	6.0	18.4	0.8	4.4	29.6
text3c	37.8	41.9	5.2	25.8	110.8
text1d	0.6	22.3	—	2.0	25.0
text2d	3.7	23.0	—	15.5	42.3
text3d	24.0	27.9	—	96.5	148.3

图7给出了采用本文方法与采用传统的块级并行压缩技术的时间消耗情况的比较。因为传输和组装时间对于两种方法都是一样的,这里只给出GPU的总运行时间。值得注意的是,为了体现压缩与解压缩的整体性能,图中的每一个竖块代表着不同大小文件采用不同方法的压缩与解压缩的时间总和。结果显示本文方法总是比块级并行方法更优。对于200 MB的原始文件text3,本文方法比没有使用合并内存访问的块级别并行方法快4倍。其主要原因在于,块级别使用的非合并内存访问在当前的GPU架构上代价很高,这在待压缩的原始文件大小越大时效果越明显。

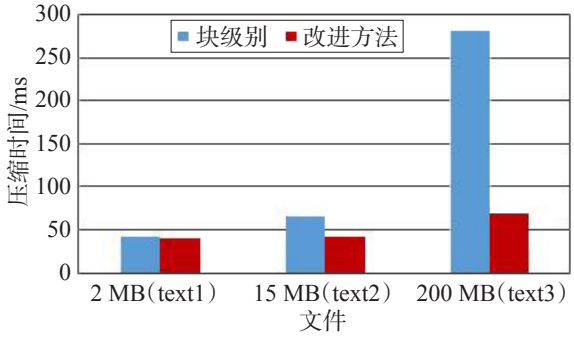


Fig.7 Comparison of compression time with block level parallelism

图7 与块级并行方法的压缩时间比较

将本文并行组装技术与传统的基于CPU的组装技术进行了对比，结果如图8所示。对于每个输入数据大小，左列代表了使用基于CPU组装技术的压缩时间，右列代表了使用并行组装技术的压缩时间。每一列底部的颜色块（块级并行方式用蓝色表示，GPU并行方式用绿色表示）显示了压缩数据从GPU设备内存到主机内存的传输与数据组装所消耗的时间的总和（ASSMB时间与D2H时间之和），顶部的颜色（块级并行用红色表示，GPU并行用黄色表示）显

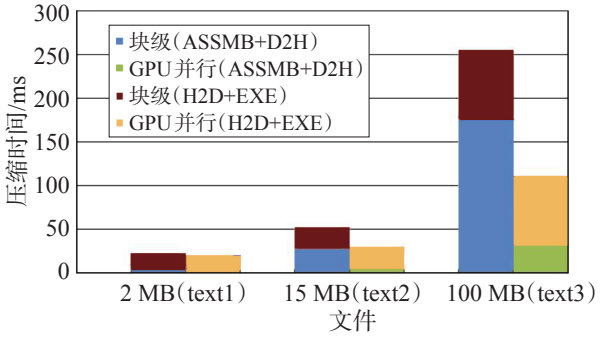


Fig.8 Comparison of compression time between different assembling approaches

图8 不同组装方法的压缩时间比较

示了从主机内存到设备内存的时间与压缩时间总和（即H2D与EXE之和）。

根据数据压缩传输时间（从设备到主机）和组装过程计算的消耗时间可以看出，尽管计算时间没有改变，但当大量的压缩数据由CPU组装时，总的时间消耗会明显增长。

由于从设备到主机的传输速度比到其他方向（例如主机到设备）的速度要慢得多，其影响更大。本文并行组装方法由于仅仅将写回主机的数据进行组装与传输，这样所花时间更少，从而成功地减少了传输时间。

5.2 LZW

本书展示了采用LZW压缩/解压缩实现方式的实验结果。将该方法应用到前文提到的3个文本文件中，每个文件将块大小设置为1 024 Byte。对每个数据块使用一个线程进行压缩/解压缩。对于每个线程，需要22个寄存器来压缩，21个寄存器进行解压缩。每个线程需要52 Byte共享内存。为每个线程块分配128个线程。

表3展示了LZW方法的结果。在所有的实验中，GPU仅仅处理一个1 024 Byte的小块。实验结果表明，GPU在压缩3个文本文件时可以获得大概1.5~2倍的加速比。从表3中可以看出，当原始文件较大时，LZW压缩/解压缩可获得更佳的性能。尽管GPU的压缩率比CPU差一些，但这些压缩率的降低在大多数应用情况下还是可容忍的。然而，并行压缩/解压缩要优于传统LZW方法，在压缩时间上可以获得1.5~2倍的加速比，在解压缩时获得最多3.2倍的加速比。

本文通过调整块大小研究了线程被分配的数据块的大小的影响。在表3中的文件text2上进行了实

Table 3 LZW compression/decompression results

表3 LZW压缩/解压缩结果

File	File size/MB	Compression ratio/(%)		Compression time/ms		Decompression time/ms	
		CPU	GPU	CPU	GPU	CPU	GPU
text1	1.98	40	46	454	263	127	89
text2	15.85	41	43	3 562	1 845	927	352
text3	100.00	44	46	21 890	11 906	5 874	1 812

验。图9显示了不同块大小情况下的压缩率。可以发现,随着块大小的增加,压缩率变得更加理想,这是因为字典在压缩时可以容纳更多的条目,这样可以进一步压缩。

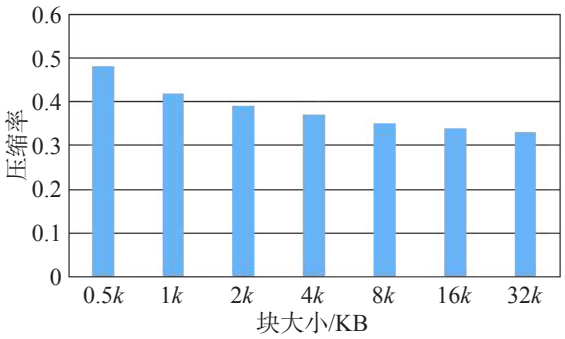


Fig.9 Compression ratio for different chunk sizes

图9 不同块大小情况下的压缩率

图10展示了不同块大小情况下的压缩时间。它表明随着压缩块大小的增加,时间也会相应增加。特别对于16*k*大小的块来说,时间增加幅度较大。这是因为在GPU下,很难为了存储压缩字典建立哈希表。不同于建立哈希表的作法,本文使用树结构和二分查找的方法来存储字典。因此当字典条目在一棵非平衡树上搜索时,搜索时间会大幅增加。

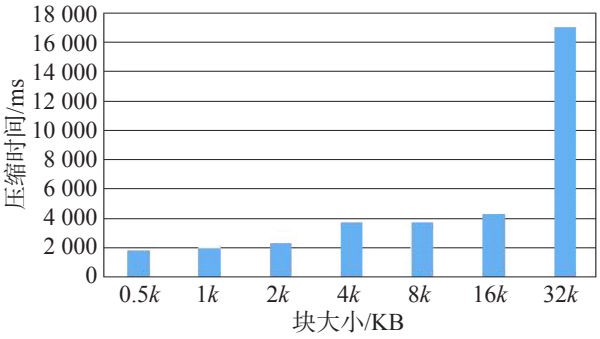


Fig.10 Compression time for different chunk sizes

图10 不同块大小情况下的压缩时间

图11显示了不同块大小情况下的解压缩时间。它说明解压缩时间在不同块大小情况下基本是一样的,但是块大小为1*k*和32*k*的情况除外。因为在LZW解压缩中,字典重构可能会被顺序地放在一个类数组的数据结构中。将一个code转换为对应的key没有额外的搜索消耗,因此解压缩时间基本是一致的。

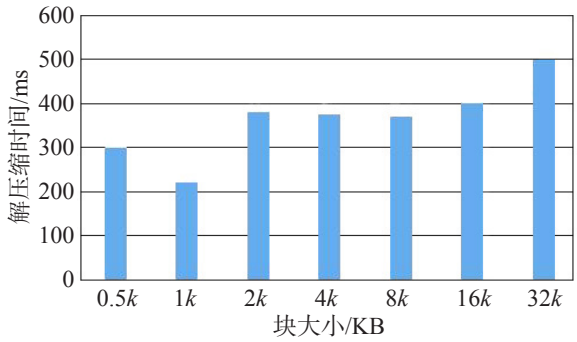


Fig.11 Decompression time for different chunk sizes

图11 不同块大小情况下的解压缩时间

5.3 基于GPU与多核CPU并行压缩的讨论

目前主流的GPU大多支持的是SIMD的体系结构,存在着分支化解、内存非合并访问等问题,因此GPU多线程执行时没有多核CPU灵活,效率会有所降低。

当前阶段,使用多核CPU也能实现基于字典的并行的压缩/解压缩算法。根据阿姆达尔定律,理论上,随着核数的增加,其加速比也越高。然而由于基于字典的压缩与解压缩处理数据频繁,内存带宽成为其主要瓶颈。目前最主流的多核CPU,如Intel Core i7-975,其带宽只有26.3 GB/s。相对于多核CPU而言,目前主流的GPU的内存带宽一般可以达到上百GB/s,如NVIDIA GeForce GTX 580,其带宽高达126 GB/s。由于存在内存带宽优势,随着GPU体系架构与编程模式的改进,GPU将非常适合处理基于字典的并行压缩与解压缩。

未来的处理器将会融合多种类型的计算核(例如传统的CPU与GPU),支持多种不同的编程模式。结合了CPU与GPU两者优势的并行压缩/解压缩技术,其性能将会有明显的提高。

6 结束语

压缩/解压缩的并行化对于数据的存储和传输来说都很重要。基于GPU的强大计算能力和内存带宽,将并行压缩迁移到GPU平台上是非常有吸引力的。现有的块级并行技术通常受制于非合并内存访问模式,这种模式使它们不能充分利用GPU的并行能力。本文提出了一种GPU友好的基于字典技术的

压缩/解压缩框架，它适用于有状态和无状态的压缩算法。通过对传统块级并行方法的适当修改，该框架能够有效减轻GPU内存访问过程中的不利影响。

将来计划基于GPU平台研究其他更多的压缩方法与技术。目前本文方法与技术仍然不能完全避免在写入不同的压缩数据块期间发生非合并的内存访问，因此希望能继续改进本文的方法与技术，进一步提升其性能。

References:

[1] Seong S W, Mishra P. Bitmask-based code compression for embedded systems[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2008, 27(4): 673-685.

[2] Franaszek P, Robinson J, Thomas J. Parallel compression with cooperative dictionary construction[C]//Proceedings of the Conference on Data Compression (DCC '96), Snowbird, USA, 1996. Washington, DC, USA: IEEE Computer Society, 1996: 200-209.

[3] Nagumo H, Lu Mi, Watson K. Parallel algorithms for the static dictionary compression[C]//Proceedings of the Conference on Data Compression (DCC '95), Snowbird, USA, 1995. Washington, DC, USA: IEEE Computer Society, 1995: 162-171.

[4] Smith G, Storer J. Parallel algorithms for data compression[J]. Journal of ACM, 1985, 32(2): 344-373.

[5] Storer J A, Reif J H. A parallel architecture for high speed data compression[J]. Journal of Parallel and Distributed Computing, 1991, 13(2): 222-227.

[6] Henriques S, Ranganathan N. A parallel architecture for data compression[C]//Proceedings of the 2nd International Parallel and Distributed Processing Symposium (IPDPS '90), Dallas, USA, 1990. Washington, DC, USA: IEEE Computer Society, 1990: 260-266.

[7] Lee J, Winslett M, Ma Xiaosong, et al. Enhancing data migration performance via parallel data compression[C]//Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS '02), Fort Lauderdale, USA, 2002. Washington, DC, USA: IEEE Computer Society, 2002: 142.

[8] Farach M, Muthukrishnan S. Optimal parallel dictionary matching and compression[C]//Proceedings of the 7th ACM Symposium on Parallel Algorithms and Architectures (SPAA '95), Santa Barbara, USA, 1995. New York, NY, USA: ACM, 1995: 244-253.

[9] Shen Ke, Cook G W, Jamieson L H, et al. Overview of parallel processing approaches to image and video compression[C]// SPIE 2186: Proceedings of the Conference on Image and Video Compression, San Jose, CA, USA, 1994. [S.l.]: SPIE, 1994: 197-208.

[10] Xie Yuan, Wolf W, Lekatsas H. Code compression using variable-to-fixed coding based on arithmetic coding[C]// Proceedings of the Data Compression Conference (DCC '03), Snowbird, USA, 2003. Washington, DC, USA: IEEE Computer Society, 2003: 382-391.

[11] Iwata E, Olukotun K. Exploiting coarse grain parallelism in the MPEG-2 algorithm, CSL-TR-98-771[R]. Stanford University, 1998.

[12] Li Kai, Chen Han, Chen Yuqun, et al. Building and using a scalable display wall system[J]. IEEE Computer Graphics and Applications, 2000, 20(4): 29-37.

[13] Roitzsch M. Slice-balancing H.264 video encoding for improved scalability of multicore decoding[C]//Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT '07). New York, NY, USA: ACM, 2007: 269-278.

[14] Klein S T, Wiseman Y. Parallel Huffman decoding with applications to JPEG files[J]. The Computer Journal, 2003, 46(5): 487-497.

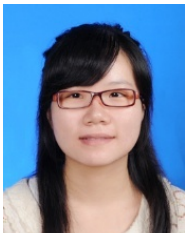
[15] Nikara J, Vassiliadis S, Takala J, et al. Multiple-symbol parallel decoding for variable length codes[J]. IEEE Transactions on Very Large Scale Integration Systems, 2004, 12(7): 676-685.

[16] Qin Xiaoke, Mishra P. A universal placement technique of compressed instructions for efficient parallel decompression[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2009, 28(8): 1224-1236.

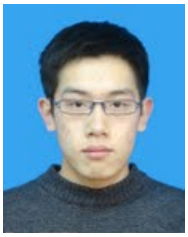
[17] Wu Leiwen, Storus M, Cross D. CUDA WUDA SHUDA: CUDA compression projects[R]. Stanford University, 2009.

[18] Welch T A. A technique for high-performance data compression[J]. IEEE Computer, 1984, 17(6): 8-19.

[19] Bader D, Madduri K. GTgraph: a suite of synthetic graph generators[EB/OL]. [2013-08-26]. http://www.cc.gatech.edu/~kamesh/GTgraph.



QIN Zishan was born in 1991. She is a master candidate at Software Engineering Institute, East China Normal University. Her research interests include software engineering and cyber-physical systems, etc.
覃子姗(1991—),女,湖南常德人,华东师范大学软件学院硕士研究生,主要研究领域为软件工程,信息物理融合系统等。



GU Fan was born in 1990. He is a master candidate at Software Engineering Institute, East China Normal University. His research interests include software testing and computer architecture, etc.
顾璠(1990—),男,江苏南通人,华东师范大学软件学院硕士研究生,主要研究领域为软件测试,计算机体系结构等。

QIN Xiaoke was born in 1982. He received the Ph.D. degree from University of Florida in 2012. Now he is a senior architect at NVIDIA. His research interests include embedded systems, computer architecture and formal method, etc.
秦晓科(1982—),男,山西人,2012年于佛罗里达大学获得博士学位,现为美国英伟达公司高级架构工程师,主要研究领域为嵌入式系统,计算机体系结构,形式化方法等。

CHEN Mingsong was born in 1982. He received the Ph.D. degree from University of Florida in 2010. Now he is an associate professor and Ph.D. supervisor at Software Engineering Institute, East China Normal University. His research interests include embedded systems, computer architecture and formal method, etc.
陈铭松(1982—),男,江苏泰兴人,2010年于佛罗里达大学获得博士学位,现在华东师范大学软件学院副教授、博士生导师,主要研究领域为嵌入式系统,计算机体系结构,形式化方法等。