杭州电子科技大学

硕 士 学 位 论 文

题　目:基于多级并行的JPEG二次压缩图片解码研究与实现

研 究 生　盛海翔

专　　业　电子信息

指导教师　盛庆华　副教授

完成日期　2023年5月

杭州电子科技大学硕士学位论文

基于多级并行的 JPEG 二次压缩图片解码研究与实现

研 究 生：盛海翔

指导教师：盛庆华　　副教授

2023 年 5 月

Dissertation Submitted to Hangzhou Dianzi University
for the Degree of Master

Study and implementation of JPEG secondary compression image decoding based on multilevel parallel

Candidate: Haixiang Sheng

Supervisor: Vice Prof. Qinhua Sheng

May, 2023

摘　要

联合图像专家组(Joint Photographic Experts Group，JPEG)编码标准目前已经成为了最为常用的图像编码标准之一。2018 年，谷歌发起了开源项目 Brunsli，该项目对基于 JPEG 编码的图片进行了二次无损压缩。经过二次压缩后得到的 brn 文件相比原始的 JPEG 文件节约了 22%左右的存储空间，通过二次压缩可以有效的降低 JPEG 文件的存储成本与传播成本。然而，由于目前 Brunsli 项目存在着二次压缩后解码速度过慢的问题，使得其在面向实际应用特别是移动终端等低算力场景时，面临较大的挑战。

在二次压缩的解码过程中，主要包含非对称系统(Asymmetric Numeral System，ANS)解码与熵编码两个环节，本文针对其中的熵编码环节进行了并行化研究。本文从线程并行化、数据并行化以及指令并行化三个层面对熵编码算法进行并行化优化，实现了不同程度的解码速度的提升。本文的主要的优化内容包括：

(1) 提出了一种解码器中的多线程并行化熵编码方案，通过引入“哈夫曼标记词”，在二次压缩的编码环节提取解码环节所需的参数信息，实现了多线程熵编码过程中的子图均衡分割与非字节对齐数据的写入。实验结果显示，使用两个线程熵编码速度为单线程熵编码的 1.6 倍以上，编码速度明显提升。

(2) 在 X86 与 ARM 架构下，本文分别设计并实现了相应的单指令多数据(Single Instruction Multi Data，SIMD)数据并行化熵编码方案。通过使用 SSE2 指令集与 ARM NEON intrinsic 指令集，分别完成了相应架构下对熵编码算法代码的数据并行化重构，完成了对熵编码的数据并行化优化。对比试验的结果显示，在 X86 架构与 ARM 架构下 SIMD 指令对于熵编码速度的提升幅度分别在 55%与 40%左右。

(3) 研究使用指令并行(Instruction Level Parallelism，ILP)进行熵编码算法的加速设计，充分利用现代 CPU 内部乱序执行(Out-Of-Order Exection，OOOE)与超标量(SuperScalar，SS)架构的硬件特点。通过对编码算法的改造使得熵编码过程中指令并行化的概率得到提升，最终实现了熵编码速度的提升。对比实验显示，经过指令并化改造后的算法相较于原有的算法的编码速度提升了 10%左右。

本文通过研究利用线程并行化、数据并行化以及指令并行化等三个层面的技术，结合熵编码算法进行了针对性的设计，探索了不同并行技术对熵编码模块运行性能的优化表现。不同层面的优化方案可以进行组合使用，以满足不同应用场景下的编码并行化需求。最后，通过对比试验验证了上述改进方案在不同 CPU 架构下，对不同 JPEG 模式下编码速度提升的幅度，为该技术用于实际应用提供了参考依据。

关键词：JPEG 编解码、熵编码、多线程、SIMD、指令级并行

ABSTRACT

The Joint Photographic Experts Group (JPEG) coding standard has become one of the most commonly used image coding standards. In 2018, Google launched an open-source project called Brunsli, which performs a second lossless compression on images based on JPEG coding. The resulting Brunsli files save about 22% of storage space compared to the original JPEG files, effectively reducing the storage and transmission costs of JPEG files through secondary compression. However, the Brunsli project currently faces a significant challenge in practical applications, especially in low computing power scenarios such as mobile terminals, due to the slow decoding speed after secondary compression.

The decoding process of secondary compression mainly involves two steps: ANS decoding and entropy coding. This article focuses on parallelization research of the entropy coding step. The entropy coding algorithm is parallelized optimized from three levels: thread parallelization, data parallelization, and instruction parallelization to achieve decoding speed improvement. The main optimization contents of this article include:

(1) Proposed a multi-threaded parallel entropy coding scheme in the decoder, which extracts the parameter information required for the decoding process in the secondary compression coding process by introducing "Huffman mark words," achieving subgraph equalization and non-byte-aligned data writing during multi-threaded entropy coding. Experimental results show that using two threads can improve the entropy coding speed by more than 1.6 times compared to single-threaded entropy coding, significantly improving the encoding speed.

(2) Designed and implemented corresponding SIMD data parallelization entropy coding schemes on X86 and ARM architectures, respectively. By using the SSE2 instruction set and ARM NEON intrinsic instruction set, the entropy coding algorithm code is data parallelized reconstructed for the corresponding architecture, completing the data parallelization optimization of entropy coding. Comparative experimental results show that the SIMD instruction can improve the entropy coding speed by about 55% and 40% respectively on the X86 and ARM architectures.

(3) Studied the acceleration design of the entropy coding algorithm using Instruction Level Parallelism (ILP), fully utilizing the hardware characteristics of modern CPU's Out-of-Order Execution (OOOE) and SuperScalar (SS) architectures. By modifying the encoding algorithm to increase the probability of instruction parallelization during the entropy coding process, the entropy coding speed is ultimately improved. Comparative experiments show that the algorithm transformed

by instruction parallelism can improve the encoding speed by about 10% compared to the original algorithm.

This paper conducts targeted design by researching the use of thread parallelization, data parallelization, and instruction parallelization technologies, combined with the entropy coding algorithm, and explores the optimization performance of different parallel technologies on the operation performance of the entropy coding module. The optimization schemes at different levels can be combined for encoding parallelization requirements in different application scenarios. Finally, through comparative experiments, the above improvement schemes are verified to provide a reference basis for the use of this technology in practical applications under different CPU architectures and different JPEG modes.

Keywords: JPEG codec, entropy coding, multithreading, SIMD, instruction level parallel

目　录

第 1 章 绪论

1.1 课题来源

本课题来源于华为公司委托的横向课题《图像原子化融合技术创新》，本文的部分优化成果已经成功应用于华为手机与华为云平台的图像压缩算法。

1.2 研究背景与意义

图片相较于文字在传播信息上具有直观、准确、生动等特征，伴随近些年智能手机的快速普及以及移动互联网、云存储等技术的快速发展，通过拍摄照片快速传播信息已经成为人们日常生活中不可或缺的信息传播方式。据统计全世界移动社交网络每小时传播的图片超过 1 亿张，每日的平均传播量超过 30 亿张。每时每刻都有海量的图片通过互联网中进行传播，如此大的图像数据传播量给整个计算机世界的网络带宽带来了巨大的挑战。

另一方面，随着近些年来手机摄像技术的快速发展，单张图片的分辨率实现了从百万级到千万级乃至亿级像素的快速跨越。更高的图像分辨率也意味着单张图片需要更大的存储空间，以未经压缩的 BMP 图片为例，在分辨率为 12032×9024 编码方式为 RGBA4448 的情况下，该文件所需的内存大小为 12032×9024×4byte = 414MB，即使是通过 JPEG 压缩后，生生成的 JPG 文件占用内存的大小也超过了 40MB。为了满足人们对于图片越来越高的清晰度追求，图像压缩算法必须能够与时俱进，不断提升编解码的速度与压缩率。

图像编码理论和算法一直在不断的发展，深度学习图像压缩挑战赛(Challenge on Learned Image Compression，CLIC)每年都会涌现出许多优秀的编码算法，但是这类算法往往仅针对特定的硬件环境或受限于专利版权问题，因而推广成本高昂且耗时漫长，最终难以应用到实际场景。因此评价一个图像编解码算法的优劣，除了分析算法的压缩效率外，也要关注在实际使用场景中的普及成本。

联合图像专家组(Joint Photographic Experts Group，JPEG)编码标准[1]自 1992 年被确立为国际标准以来，凭借其优秀的压缩率以及通用浏览器与操作系统的广泛支持，目前已经成为网络图片以及智能手机最主要的编码格式。伴随着时间的累计，不断地积累的 JPG 图片占用着巨大的社会存储资源，特别是针对云存储平台而言，若能够通过二次压缩实现对 JPEG 图片压缩率的提升，则必能大幅降低图片信息的存储与传播成本。

本文正是基于当前背景，为了实现在实际应用领域中降低图像信息的存储与传输的成本的目标，研究对于 JPEG 图片的二次压缩方案。在关注二次压缩算法对 JPEG 图片压缩率提升的同时，本文也考虑到了实际编解码场景中对于图片解码时效性的要求。特别是针对计算资源通常相对有限且对于功耗的控制极为敏感的嵌入式设备。本文重点研究了在二次压缩算法的解码环节引入不同的并行化技术手段，提升二次压缩后图像文件的解码速度。

1.3 国内外研究现状

1.3.1 JPEG 图像二次无损压缩研究现状

第一代 JPEG 编码标准于上世纪 80 年代被提出，但直到 2014 年才迎来第一次更新，足见 JPEG 编码本身的性能优异性。在这期间也出现过类似 JPEG-2000[2,3]的替代性编码方案，但往往由于新方案在性能的提升幅度上并不突出，且重新推广的成本高昂，从而难以实现 JPEG 编码标准一般的普及程度。

为了实现更高的压缩率，JPEG 编码巧妙地利用了人类眼球的生理学特点，将图片中对于人眼而言较为不敏感的色度信息与高频信息进行了滤除，实现了在不影响人眼视觉效果的前提下，大幅提升了算法的压缩率。2016 年由武汉大学张雅媛，孔令罔进一步结合人类眼球的生理学特点，通过人眼对于亮度对比度敏感函数设计了一种新的亮度量化表，基于该改进型亮度量化表提出了 JPEG-HVS 压缩算法[4]。经过试验测试 JEPG-HVS 的压缩比相较于原有 JPEG 压缩比提高了 53.56%左右，并且 JPEG-HVS 在编码时间上要明显低于传统的 JPEG 编码。Lina Guo, Xinjie Shi, Dailan He 等人提出了一种基于深度学习的 JPEG 二次压缩方案，通过深度学习技术在 DCT 域上运行，提出了一种多级跨通道的熵模型，实现了对 JPEG 图片的无损二次压缩[5]。

2014 年由 Jarek Duda 提出了非对称系统(Asymmetric Numeral System，ANS)[6]，极大的推动了有限状态熵(Finite State Entrope，FSE)技术的发展；2018 年，由谷歌发起了开源项目 Brunsli[7]，该项目基于 ANS 的编码理论对 JPEG 图片进行了二次压缩，经过二次压缩后生成的 brn 文件相比原始的 JPEG 文件节约了 22%左右的存储空间，且 brn 文件通过解码可以无损的还原为原始的 JPEG 文件。对于海量的 JPEG 图片生成设备，该项目可以有效的降低存储图片所需的存储空间，并且在分享与转发图片时可以将 brn 文件还原到 JPEG 文件进行顺利的流通，不需要额外的编码标准推广成本。

虽然经过 Brunsli 项目二次压缩后的图片文件能够在保证流通性的同时有效的提升 JPEG 图片的压缩率，但是二次压缩在原有的 JPEG 编解码的基础上带来了额外的解码时间使得该项目目前的解码速度并不理想。经过测试，一张分辨率为 2976*3968 大小为 5.84MB 的 JPEG 图片，由二次压缩带来的额外解码时间为 566ms(AMD Ryzen 7 4800U 平台下)。解码时间的增加导致该方案难以被推广到大多数的图片编解码的使用场景中，因此本文对二次压缩方案中的解码环节进行了并行化研究，通过对解码过程中的熵编码环节并行化加速，缩短解码时间。

1.3.2 JPEG 编码并行化研究现状

伴随着计算机硬件架构的飞速发展以及各个领域对于并行化计算需求的不断提升，计算机并行化方案呈现出百花齐放的状态。Flynn 分类法[8]（Flynn's Taxonomy）根据计算机的指令流与数据流的执行方式将计算机系统分为四大类：

（1）单指令单数据(Single Instruction Single Data，SISD)[9,10]，该类计算机的特点是 CPU 在一个时钟周期内仅能够操作一条指令或一个数据（不考虑多核多线程的情况）。由于该方案

在软件与硬件上的设计都较为简单因此上世纪绝大多数的个人电脑均采用 SISD 的架构，这类架构的计算机无法实现真正意义上的并行化计算。

（2）单指令多数据(Single Instruction Multi Data，SIMD)[11,12]，该类计算机特点是 CPU 在一个时钟周期内可以实现对多个数据元素的并行处理。该类架构主要设计用于大规模的矩阵运算，在图像、音频、视频等多媒体领域有着巨大的优势，GPU 架构就是最为典型的 SIMD 架构。

（3）多指令单数据(Multi Instruction Single Data，MISD)[13,14]，该类计算机的特点是包含多个控制单元，可以并行执行多个指令，不同的指令可以并行操作同一个数据元素。该类型架构的应用场景限度较小，通常用于一些特殊用途的计算任务如对针对单个信号的多频滤波器处理单元。

（4）多指令多数据（Multi Instruction Multi Data，MIMD)[15,16]，该类计算机内部包含多个控制单元，在一个时钟周期内可以完成多个指令并行操作不同的数据元素。根据内存与存储方式的分布特点可以进一步划分为并行向量处理机（PVP）、对称对多处理机（SMP）、大规模并行处理机（MPP）、工作机群（COW)、分布式共享存储系统（DSM）。该类架构主要应用于仿真、建模以及通信交换机等多个领域。

目前，对于并行化 JPEG 编解码的研究主要集中在两个方面，一是利用 SIMD 架构实现数据并行化加速，如国内的张敏华团队利用 GPU 对于图像处理方面的并行化优势，基于 OpenCL 实现了并行化的 JPEG 编码[17]，有效提高了算法的执行效率,大幅降低了算法的执行时间；另一个主要的研究方向利用多核处理器的硬件特点实现并行编解码，如廖醒宇、余水来利用 ARM 架构的特点,基于 JFIF 数据交换格式语法层可并行化特点提出一种 JPEG 并行编码策略，实现了多核并行编码 JPEG 图片，大幅提升了超高分辨率 JPEG 图片的编解码速度[18]。

JPEG 编码主要包含离散余弦(Discrete Cosine Transform，DCT)变换、量化、熵编码三个基本环节，其中 DCT 变换与熵编码需要耗费较大计算资源，也是造成高像素 JPEG 图片编解码卡顿的主要原因。国内外有许多的图像编解码加速方案的研究也正是围绕这两个环节进行探索与尝试。

在 DCT 变换方面，由 da Silveira 带领的团队通过求解一个多准则优化问题，得到了四个低时间复杂度的 8-point DCT，有效的降低了 DCT 运算的时间复杂度[19]；由 Brahimi 教授带领的团队将空元素引入指定的整数 DCT 变换，实现了通过加法与移位运算代替原有的乘法运算，有效的降低了算法的时间复杂度[20]；Singhadia 教授的团队基于 CUDA GPU 与 SMP 架构，使用 2D-DCT 并行化方案，有效的提升了 JPEG 编码速度[21]；国内的王成友教授通过使用全相位双正交变换（APBT）来代替离散余弦变换，提出并行 APBT-JPEG 大幅的降低了因为 DCT 变换造成的编码时间成本过高问题[22]。

传统的 JPEG 编码在熵编码环节使用的是行程(Run Length Encoding，RLE)编码+可变长整数(Variable Length Integer,VLI)编码+哈夫曼编码或算术编码的编码方案。由朱福顺、严华团

队基于 GPU 对 JPEG 的熵编码环节进行了并行优化，极大的提升了熵编码的速度[23]；Sodsong，Jung 等人提出了一种在异构多核上用于 JPEG 解压缩的并行熵解码的新方法 JParEnt，实现了熵解码速度的巨大提升[24]。

1.4 研究的主要内容及本文的组织结构

1.4.1 研究的主要内容

JPEG 标准中的编码模式主要可以分为四种，分别是基于离散余弦变换变换的连续编码模式、基于离散余弦变换变换的渐进模式、无失真模式以及分级模式。目前绝大多数的 JPEG 图片采用的是基于离散余弦变换的连续模式，在部分网页图片中使用的是基于离散余弦变换的渐进模式。本文之后所研究的 JPEG 二次压缩也都是针对这两个编码模式下的 JPEG 图片。

本文的研究基于谷歌的 Brunsli 项目，该项目已经实现了对 JPEG 图片的二次无损压缩[25]，但受限于二次压缩算法的解码时间成本较高，该项目目前难以应用到实际场景中。为了让二次压缩方案在压缩率的提升的同时，尽可能的减少二次压缩带来的额外解码时间，本文对 Brunsli 项目解码过程中的熵编码环节做了进一步的优化。从线程级并行化、数据级并行化以及指令级并行化三个层面实现了对熵编码算法的并行化优化，每一种优化方案都可以实现不同程度的编码速度提升，且不同方案间可以实现组合使用，大幅提升编码的速度。本文针对不同硬件架构（ARM、X86）以及不同的 JPEG 编码模式（Baseline 与 Progressive）设计了不同并行化优化方案，增强了并行化二次压缩方案对于不同平台的兼容性。

1.4.2 本文的组织结构

本文共分为六个章节，其中前两个章节主要介绍本文的研究目标以及编解码的理论基础，第三、四、五章节分别介绍了三种并行化技术具体的优化方案以及优化后的实验结果，第六章对本文目前的实验结果进行了总结并展望了后续的相关工作。

第一章， 为引言，主要介绍了本文的研究背景与研究方向，分析了当前二次压缩方案存在的主要问题，以及本文的主要研究内容；

第二章， 介绍了二次压缩的基本环节以及相应的理论基础，对三种熵编码进行了具体的介绍与比较，然后提出了本文并行化方案的整体软件框架与优化方向；

第三章， 介绍了多线程并行化方案的主要技术难点及对应的解决方案，通过实验说明了多线程并行化方案的加速效果；

第四章， 介绍了在不同架构下数据并行化熵编码方案的设计难点以及相应的编码方案，最后通过对比实验得到数据并行化后熵编码的提速效果；

第五章， 介绍了指令并行化实现的基本原理以及如何通过软件优化提升编码过程中的指令并行化概率，最后通过对比实验说明了提升指令并行化改了的可行性；

第六章， 总结与展望。

第 2 章 JPEG 二次压缩理论基础

本章介绍了并行化二次压缩方案相关的理论基础，主要包含了三个方面：1）基本框架，介绍了并行化二次压缩方案的基本框架，简要的说明了二次压缩的基本流程以及优化的主要内容；2）编码理论，介绍了 JPEG 编解码的基本环节与相关理论基础，重点分析了并行化二次压缩涉及的熵编码环节；3）数据段结构，对 JPEG 文件中的数据段结构进行了必要的分析，为后续并行化方案的具体实现做了铺垫。

2.1 JPEG 二次压缩方案总体框架

本小节首先介绍了 Brunsli 项目的编解码框架，通过该框架说明了二次编解码的基本流程。然后引入了本文并行化二次压缩并行化方案的总体框架，说明了三种并行化方案主要的优化内容以及优化的具体环节。

2.1.1 Brunsli 编解码方案

本文的二次编解码框架以谷歌的 Brunsli 项目框架为基础，如下图 2.1 所示 Brunsli 项目主要包含了编码过程和解码过程两个主要部分。编码过程负责对原有 JPEG 图片进行二次压缩，二次压缩后得到以 brn 为后缀的 brn 文件，该文件作为最终的存储文件相较于原有的 JPEG 图片节约了约 22%的存储空间。解码过程是编码过程的逆过程，通过相应的解码操作将 brn 文还原为原有的 JPEG 图片。该项目保证编解码前后的 JPEG 图片的图像数据二进制码流完全一致，不会产生画质变化。

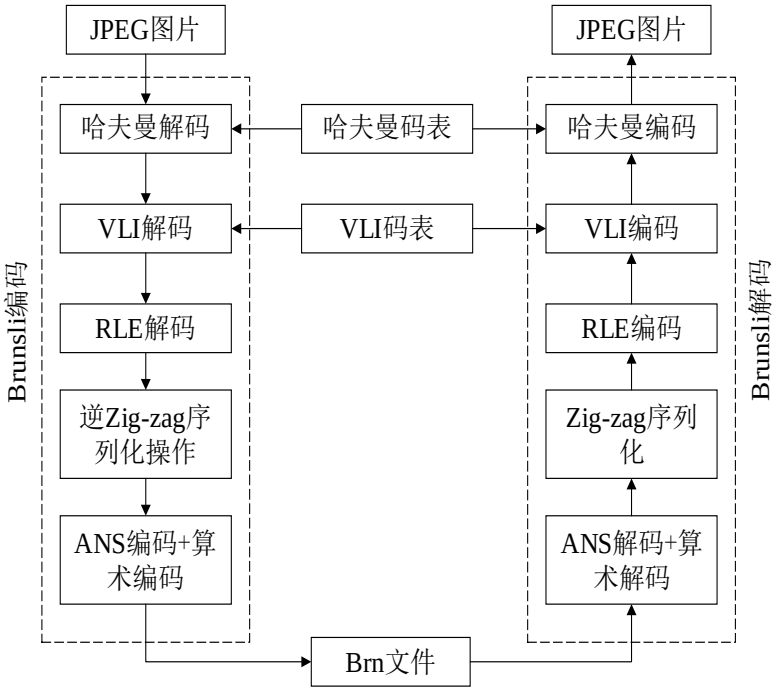


图 2.1 Brunsli 编解码基本框架

Brunsli 项目的编码过程主要涉及两个主要部分：首先对原有 JPEG 图片进行哈夫曼解码、VLI 解码、RLE 解码以及逆 Zig-zag 序列化操作，使得码流恢复到 JPEG 编码量化后的状态；然后对解码后码流中的 AC 系数和 DC 系数分别采用 ANS 编码与算术编码生成新的二进制码流写入到 brn 文件。由于 ANS 编码与算术编码相较于原有的哈夫曼编码有更高的压缩率，因此经过二次压缩生成的 brn 文件相较于原有 JPEG 文件需要的存储空间更小。

Brunsli 解码过程为编码过程的逆过程同样包含了两个主要的组成部分：首先是对 brn 文件进行 ANS 解码与算数解码，恢复到 JPEG 量化后的码流；然后对 ANS 解码后的码流重新进行 Zag-zig 序列化、RLE 编码、VLI 编码以及哈夫曼编码，恢复到原有的 JPEG 编码格式。

相较于重新设计一种新的图片压缩方案，Brunsli 二次压缩方案的优势在于：

（1）从存储角度看，对于目前已经存在的海量 JPEG 图片该方案都可以实现二次压缩，降低图片的存储成本，并且二次压缩前后的 JPEG 图片保持了相同的图像质量。

（2）二次压缩方案的推广成本极低，由于二次压缩解码后可以还原成原始的 JPEG 文件，这使得该图片依旧可以在不支持 brn 解码的平台上使用。若是完全重新设计一种新的图片编码方案，则需要花费巨大的时间成本与经济成本，以实现新编码方案的普及与使用。

2.1.2 解码并行化技术

伴随图像数据量快速增长，传统的单处理器串行解码方案以及难以满足对图像数据的实时处理。现代 CPU 均支持异构并行计算技术，凭借其高效的并行计算效率以及良好的功耗控制，目前已成为了主流的计算模式。因此本文为了提升二次编码的解码效率，对解码算法进行了并行化优化。目前对于算法并行化的研究主要集中在三个方向，分别是：

（1）多线程并行化，使用多线程并行化是目前最为主流也是最常见的并行化手段，主要是利用了现代 CPU 内部的多核架构，通过将串行任务分割为若干子任务交由不同的处理核心并行执行[26,27]实现算法的并行化；

（2）算法优化：通过算法裁剪、以空间换时间以及算法自适应降级等手段实现降低算法时间复杂度[28,29]，该方案不需要额为的硬件支持，优化后的运行成本很低，但算法优化的设计难度普遍较高且提升的效果相对有限。

（3）硬件加速：通过 CPU、GPU 内部的硬件架构[30]的支持，对原有算法进行 SIMD 数据并行化[31]、Cache miss 优化以及 GPU share 等技术手段的针对性改造，以实现算法在特定硬件架构下的并行化加速。

通过对原有 Brusli 项目代码的分析本文最终选取了多线程并行化、SIMD 数据并行化以及指令并行化三个技术方向对原有代码进行优化。从线程级、指令级以及数据并行级三个维度分别进行并行化优化，针对不同的编解码使用场景的特点可以进行灵活的组合，以实现硬件性能的最大程度发挥。比如在云存储等高并发场景下，可以使用多线程+数据并行化方案(指令并行化需要额为的内存因此不适用于高并发场景)，而在对于功耗较为敏感的终端设备，如智能手机等可以使用数据并行化方案加指令集并行化(多线程会带来额外的功耗)。

2.1.3 二次压缩并行化方案

Brunsli 方案通过对 JPEG 文件的二次压缩降低了图像存储的空间成本，但同时也增加 JPEG 图片编码与解码过程中的时间成本。日常浏览 JPEG 图片过程中都需要对 JPEG 文件进行解码操作，Brunsli 在解码过程中需要先将 brn 文件还原成 JPEG 文件，因此解码过程中需要额外的解码时间，该额外的解码时间在很大程度上会决定整体的解码算法使用体验。经过实测在 CPU 为 AMD Ryzen 7 4800U 操作系统为 win11 的环境下，一张分辨率为 2976*3968 大小为 5.84MB 的图片经过 Brunsli 二次压缩后生成的 brn 文件大小为 4.51MB，而带来的额外解码时间为 566ms。对于频繁的图片浏览操作，这一解码时间显然不是很理想,为此本文从提升解码速度的角度出发，通过不同级别的并行化解码手段共同作用来提升整体的二次压缩方案的解码效率。

本文的并行化方案框图如图 2.2 所示，在 brn 文件解码过程中主要包含四个编码环节分别是：Zig-zag 序列化、RLE 编码、VLI 编码以及哈夫曼编码，本文将这四个编码环节合并为熵编码环节，并对该环节进行统一的并行化操作。基于 Brunsli 的总体框架，本文方案的主要改进点在于：

（1）将 Brunsli 方案中使用算术编码与 ANS 编码分别编码 DC 系数与 AC 系数的方案替换为所有系数采用 ANS 编码，以适配后续的并行化操作。

（2）将 Brunsli 解码方案中的 ANS 解码过程与熵编码过程进行并行化优化，以满足算法在不同架构、不同环境下的并行化需要。

（3）为了配合解码过程，对编码过程进行了一定的改进，引入了哈夫曼切换词、Bitmap 等数据结构，以适配相应的并行化操作。

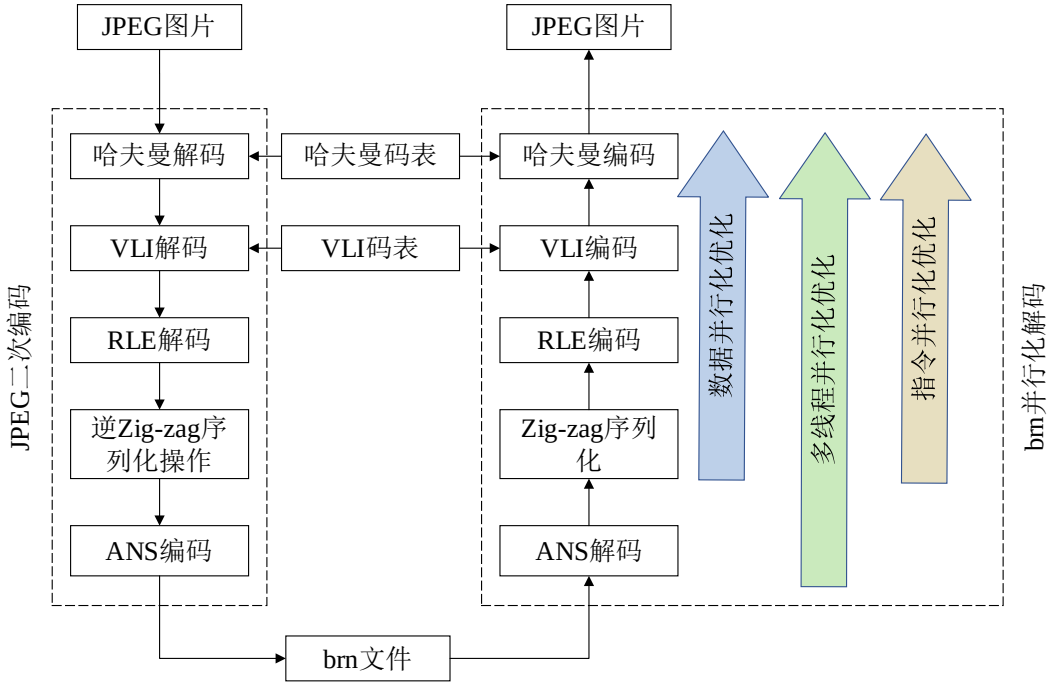


图 2.2 JPEG 二次压缩方案总体框架

2.2 JPEG 编解码基本流程

由于二次压缩过程中涉及对于原有 JPEG 图片的编解码过程，因此本小节对 JPEG 编解码流程做了详细介绍。JPEG 编码按照其内部编码方式应属于混合编码，即通过两种或两种以上的编码技术从不同理论方向对信源数据进行组合编码。在编码过程中，JPEG 主要通过对色彩空间变换后数据的降采样，离散余弦(Discrete Cosine Transform，DCT)变换[32-34] 后对数据进行量化以及熵编码技术来实现图像数据的压缩，其中降采样与量化过程会删除图像中对于人眼而言的冗余信息以实现数据量的减少，删减后的数据会经过多种熵编码[35]环节实现进一步的二进制数据压缩。与之对应的在解码环节需要使用各个编码环节的逆过程来实现图像的还原。JPEG 编码的基本编解码流程如下图 2.3 所示，本小节后续的内容会对 JPEG 编码过程中主要环节进一步展开介绍。

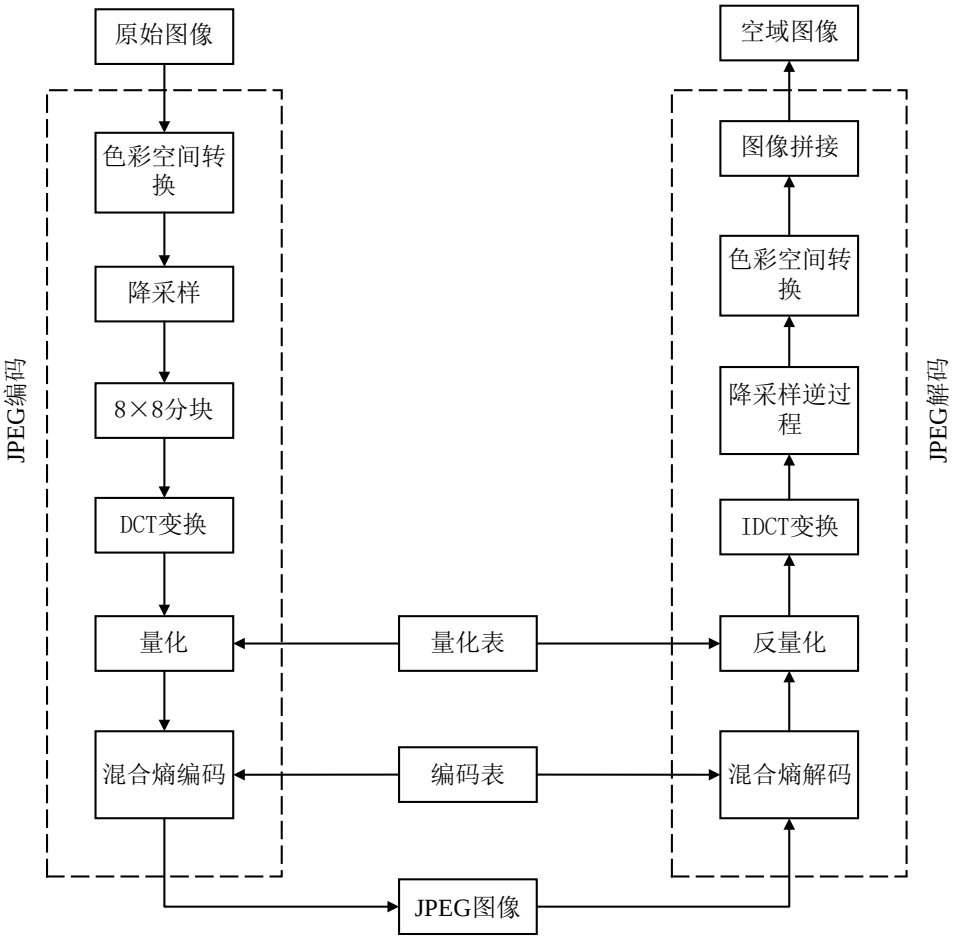


图 2.3 JPEG 编解码基本框架

2.2.1 色彩空间变换与降采样

JPEG 编码流程中色彩空间变换[36,37]是降采样操作的前提，而实现对指定色域的降采样也是色彩空间变换的根本目的，因此本小节将会对两个环节一起进行介绍。

色彩空间是为了通过数字对颜色进行量化而人为设定的数学模型，常见的色彩空间包括 CIE 模型、RGB 模型以及 YUV 模型。每一种色彩空间模型都可以实现将颜色通过数值进行

量化，以及通过对应的色度数值还原回原来的颜色。比如一张使用 RGB 模型的彩色图片是通过将图片分解为红、绿、蓝三种颜色分量的色度图（色彩饱和度图），然后对每一个像素的色度使用数值进行描述，因此图片文件实际存储的是红、绿、蓝三种颜色的数据矩阵，通过这三个数据矩阵所代表的色度信息的叠加又可以还原回到原始的彩色图片。同一张图片对应不同的色彩空间可以得到针对不同的数据矩阵，但本质上来说这些色度矩阵描述的都是同一张图片，因此不同的色彩空间之间可以通过数学的方式实现色彩空间之间的转换。

人类的眼球拥有约 1 亿个视杆细胞（负责亮度信息检测）以及红、绿、蓝三种颜色各约六百万个视锥细胞（负责相应的色彩信息检测），由于这一生理学上的特点人类对于图像色度信息的敏感程度要远低于亮度信息。JPEG 编码使用色彩空间转换的目的就是将由 RGB 色彩空间描述的图片转换为由 $YCrC_b$ （亦称 YUV）色彩空间进行描述，其中 Y 分量表示图片的亮度信息而 Cr、Cb 分别表示红色和蓝色的色度信息。使用 YCrCb 色彩空间的优点在于通过利用人眼对于亮度信息敏感而对色度信息并不敏感的特点，对色度信息进行降采样，减少图像数据中对于人眼而言存在的图像冗余信息。由 RGB 色彩空间与 YC_rC_b 色彩空间相互转换公式如式（2.1）与式（2.2）所示：

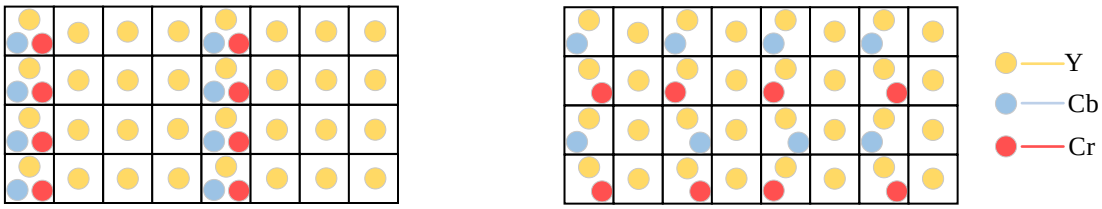
$$\begin{cases} \mathrm{Y} = 0.299R + 0.587G + 0.114B \\ \mathrm{C}_b = -0.169R - 0.3319G + 0.5B + 128 \\ \mathrm{C}_r = 0.5R - 0.419\mathrm{G} - 0.081B + 128 \end{cases} \tag{2.1}$$

$$\begin{cases} R = Y + 1.402(C_r - 128) \\ G = Y - 0.34414(C_b - 128) - 0.7141(C_{\mathrm{r}} - 128) \\ B = Y + 1.772(C_{\mathrm{b}} - 128) \end{cases} \tag{2.2}$$

在完成色彩空间的转换后，JPEG 会根据不同的采样格式对不同的色度进行采样。JPEG 常用的采样格式包括 $YCrC_b$4:1:1、$YCrC_b$4:2:2、$YCrC_b$4:2:0 以及全采样 $YCrC_b$4:4:4，不同的采样格式表示对不同色彩空间数据的采样精度。以采样格式为全采样 $YCrC_b$4:4:4 为例，它表示为了描述原图中分别对应于横向和纵向的 2×2 个像素，对于亮度信息 Y 以及色度信息 Cr 与 Cb 均采集全部的四个数据，因此在该采样格式下描述该位置四个像素所需要的数据量为 12。而对于采样格式为 YCrCb4:2:0 的图片而言，为了描述原图中分别对应于横向和纵向 2×2 个像素，对于亮度信息 Y 的数据每采集四个数据即全采样，而对于相同位置的红色色度 Cr 与蓝色色度 Cb 仅采集一个数据，因此描述该位置四个像素所需要的数据量为 4+1+1 共 6 个数值。通过该采样格式后实现了对 Cr 与 Cb 色域的降采样，使得降采样后的数据量的规模缩减为原有的数据规模的一半，有效的减少了图片中对与人眼而言存在的冗余信息。

降采样环节除了实现对整体数据规模的缩减外还决定采样的方式，比如采样格式 YCrCb4:1:1 与 YCrCb4:2:0 均会对 4 个像素采集 4 次亮度信息，1 次色度信息 Cr 以及一次色度信息 Cb，但两种方式的选择采集位置有所不同如下图 2.4 所示，YCrCb4:1:1 选择的是对横向 4 个像素进行降采样，而 YCrCb4:2:0 采用的式对 2*2 的像素区间内进行交替采集。

YCrCb4:2:0 是目前 JPEG 图片在 BaseLine 模式下最常见的采样格式。



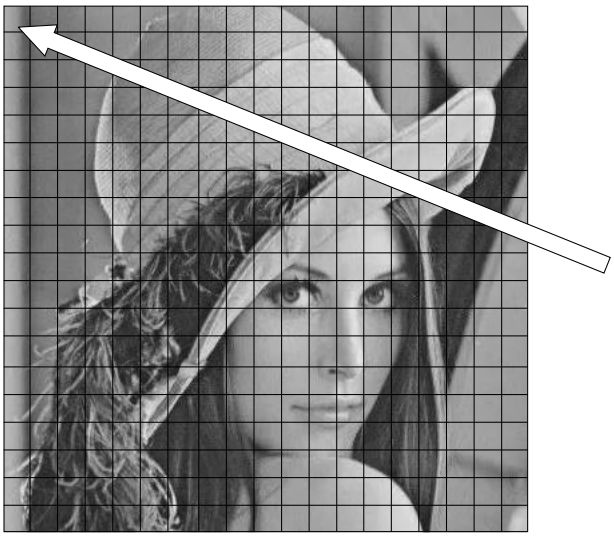
（a）YCbCr4:1:1　　（b）YCbCr4:2:0

图 2.4 采样模式对比

2.2.2 DCT 变换与量化

DCT 变换与量化环节间的关系与色彩空间变换与降采样环节间的关系类似，两者间也是相互绑定的关系，DCT 变换为量化创造了条件，而量化则是进行 DCT 变换的最终目标。因此本小结将会一起介绍这两个环节。

由于 DCT 变换标准[38]处理单元必须是一个 8×8 的矩阵，因此在进行 DCT 变换前图像数据首先会被分割为若干个 8×8 大小的 Block 数据块作为编码的基本单元如图 2.5 所示，后续所有的编解码操作都会以 Block 数据块作为操作的基本对象。



Block数据块

99	21	24	6	43	23	65	6
78	98	5	32	13	4	5	3
23	34	54	23	45	42	43	23
65	76	87	89	43	32	21	54
37	54	32	32	43	6	2	3
58	84	43	32	54	6	65	56
32	43	58	78	32	83	56	42
45	65	42	4	32	6	89	30

图 2.5 Block 数据块

离散余弦变换是傅里叶变换将离散数据变换为仅含有余弦分量的一种可逆数学变换。与色彩空间变换的过程相类似，JPEG 流程中的 DCT 变换同样是利用了人类眼球生理学上对于图像中高频信息不敏感的特点，将原有图像信息在指定的频率范围上进行频域展开，并通过后续的量化操作实现对图像中高频数据的有效滤除，最终实现图像数据的整体规模缩减。每一个 8×8 的 Block 数据块经过 DCT 变换后，会对原有的 Block 数据在指定频域进行展开并生成一个 8×8 的系数矩阵，矩阵中的每一个系数分别表示对应频率的余弦分量的系数。系数矩阵左上角即（0,0）位置为该 8×8 子图的直流（Direct Curren，DC）系数，DC 系数表示该子图范围内的像素均值，剩余的 63 个系数被称为该子图的交流（Alternating Current，AC）系数，AC 反映了该子图的具体的边缘与细节信息，越接近矩阵右下角位置的 AC 系数反映的图像信

息越高频。将系数矩阵重新存放到对应的 Block 数据块中， 实现了使用频域的系数矩阵来代替原有 Block 数据对子图进行描述。与编码过程中的 DCT 变换相对的解码过程包含 DCT 变换的逆过程离散余弦逆变换(Inverse Discrete Cosine Transform，IDCT)，关于 DCT 变换与 IDCT 变换的数学表达如式（2.3）与式（2.4）所示， $f(i,j)$ 表示输入 DCT 变换的 Block 数据，其中 $0 \le \mathrm{i}, j \le 7$ ， $F(u,v)$ 表示经过 DCT 变换后得到的频域系数值，其中 $0 \le u,v \le 7$ 。

$$F(u,v)=\frac{1}{4}C_uC_v\sum_{i=0}^{7}\sum_{j=0}^{7}f(i,j)\cos\left(\frac{(2i+1)u\pi}{16}\right)\cos\left(\frac{(2j+1)v\pi}{16}\right) \tag{2.3}$$

$$F(i,j)=\frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7}C_{\mathrm{u}}C_V f(u,v)\cos\left(\frac{(2i+1)u\pi}{16}\right)\cos\left(\frac{(2j+1)v\pi}{16}\right) \tag{2.4}$$

$$\text{其中,}\begin{cases}C(u)=C(v)=\dfrac{1}{\sqrt{2}},u=v=0\\ C(u)=C(v)=1,\text{其他}\end{cases}$$

DCT 变换实现了对每个 Block 数据块的频域展开，接下来的量化操作会对频域系数进行量化操作。由于人眼对于图像中的高频信息并不敏感，因此可以通过一次“低通滤波”实现对高频信息的部分滤除，而量化的过程正是对应着这个“低通滤波器”。根据人眼对于色度信息以及高频细节的识别能力较弱的特点，量化过程通过使用不同量化矩阵来实现有选择的数据滤除。量化矩阵的量化过程可以通过公式（2.5）来表示，其中 $F(u,v)(0 \le u,v \le 7)$ 表示 DCT 变换后得到的频域系数矩阵， $Q(u,v)$ 表示一个 8×8 的量化矩阵， $F^Q(u,v)$ 表示量化后的得到的新的频域系数矩阵。

$$F^Q(u,v)=round\left(\frac{F(u,v)}{Q(u,v)}\right) \tag{2.5}$$

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

标准亮度系数量化表

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

标准色度系数量化表

图 2.6 量化系数表

量化矩阵 $Q(u,v)$ 并不是一个固定值的矩阵，量化矩阵内部的参数是由被量化对象的标准量化表与该图像当前的压缩质量因数(Quality Factor，QF)共同决定。由于人眼对于亮度信息与

色度信息的敏感程度不同，因此对于亮度频域系数使用的标准量化表与对于色度频域系数使用标准量化表是不同的，如下图 2.6 所示，可以明显看出亮度系数的量化表中的值相较于色度系数量化表偏小，这样设计是为了减小量化操作对于亮度信息的影响。两张标准量化表对于靠近右下角的高频系数均采用较大值进行削弱，而对于靠近左上角的低频系数均使用较小值进行保留，这样做的目的是为了实现“低通”的目标。

影响量化矩阵的因素除了标准量化表外还有图像当前的压缩质量因数(Quality Factor，QF)，QF(1≤QF≤100)值的大小直接影响到图像的压缩质量与压缩率，QF 的具体大小根据实际压缩需要进行设定，对于 QF 大于等于 50 与小于 50 的量化矩阵计算方式有所不同，如式子(2.6)所示，QT(u,v)表示当前的标准量化表。

$$Q(u,v)=\begin{cases}\dfrac{1}{100}\left(\dfrac{QT(u,v)\times 5000}{QF}+50\right) & QF<50\\ \dfrac{1}{100}\left[QT(u,v)\times(200-2\times QF)+50\right] & QF\geq 50\end{cases} \tag{2.6}$$

2.2.3 熵编码

熵编码是一种常用的编码压缩手段，它通过对信源字符序列的概率分析，建立一个基于概率分布的信源字符于二进制符号间的映射模型，通过该映射模型最终实现了对信源信息的无损压缩。JPEG 中的熵编码以 8×8 的 block 块作为编码的最小单位，对于每一个 block 块都需要经过 Zig-Zag 序列化、行程长度(Run-Length Encoding，RLE)编码[39]、变长度整数编码(Variable Length Integer ,VLI)以及哈夫曼编码四个环节如下图 2.7 所示。

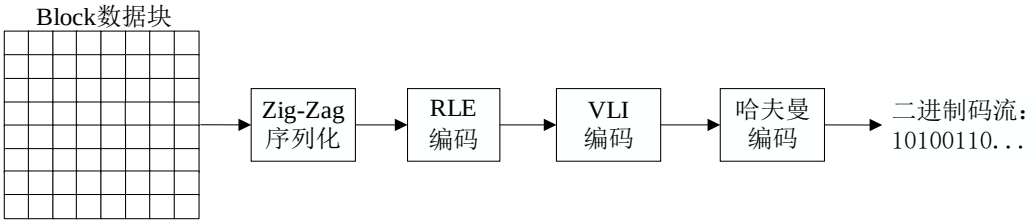


图 2.7 Block 熵编码流程图

Zig-Zag 序列化按照如下图 2.8 所示的顺序进行，将原来的 Block 数据转化为一维的数据以方便后续的编码处理。经过序列化后的系数可以被划分为直流(Direct Current，DC)分量和交流(Alternating Current,AC)分量，直流分量即 Block 中第一个系数交流分量即 Block 中剩余的 63 个系数。对于 DC 系数而言在进行 RLE 编码前需要进行差分脉冲编码调制(差分脉冲编码调制,DPCM)即当前 Block 的 DC 系数的编码值应为与上一个 Block 的 DC 系数的差分结果。

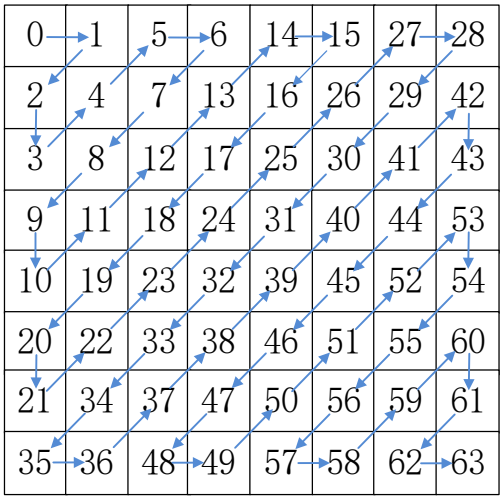


图 2.8 Zag-zig 序列化

Block 数据经过量化后高频 AC 系数会产生大量的 0 数据，RLE 编码通过统计零系数的个数以减少后续编码系数的个数。 VLI 编码的目的是采用非字节对齐的方式对非 0 系数进行编码，对于正数系数而言 VLI 码表的结果为当前系数二进制值去除无效前导 0 数字的个数，对于负数系数而言编码的结果为当前系数绝对值的二进制值反码，VLI 码表如下表 2.1 所示。

表 2.1 VLI 码表

Value		size	Bits	
0		0	-	
-1	1	1	0	1
-3,-2	2,3	2	00,01	10,11
-7,-6,-5,-4	4,5,6,7	3	000,001,010,011	100,101,110,111
-15,...,-8	8,...,15	4	0000...,0111	1000,...,1111
-31,...,-16	16,...,31	5	0 0000,...,01111	1 0000,...,1 1111
-63,...,-32	32,...,63	6	00 0000,...	...,11 1111
-127,...,-64	64,...,127	7	000 0000,...	...,111 1111
-255,...,-128	128,...,255	8	0000 0000,...	...,1111 1111
-511,...,-256	256,...,511	9	0 0000 0000,...	...,1 11111 1111
-1023,...,-512	512,...,1023	10	00 0000 0000,...	...,11 1111 1111
-2047,...,-1024	1024,...,2047	11	000 0000 0000,...	...,111 1111 1111

为了进一步提高熵编码的压缩比，JPEG 编码在上述编码流程完成后还要在进行一次哈夫曼编码。JPEG 将当前系数前的零系数个数与 VLI 编码的实际长度合并为一个字节，并将该数据进行哈夫曼编码。

2.3 熵编码的比较

目前主流的熵编码方式主要有哈夫曼编码和算术编码两种方式，这也是目前 JPEG 的熵编码主要方式。2014 年由 Jarek Duda 提出了非对称系统，为有限状态熵编码提供了新的编码

选择，本小节主要介绍了三种熵编码各自的实现方式以及具体的性能特点。

2.3.1 哈夫曼编码

哈夫曼编码[40]是速度最快的编码实现方式，也是主流 JPEG 熵编码中的一部分。哈夫曼编码的核心是根据待编码字符出现的频次生成哈夫曼树进而得到哈夫曼编码表，最后实现字符与对应的二进制编码符号间的映射。如下图 2.9 所示，哈夫曼树的建立，主要通过以下三个基本环节：

(1)统计所有待编码字符的出现频次并根据出现频次进行升序排序生成字符频次队列。

(2)将字符队列前两个节点进行合并，合并后的节点的出现频次等于左右子树频次之和。

(3)将合并后的节点重新加入队列并重新排序，重复(2)、(3)操作直到所有字符挂载到同一棵树。

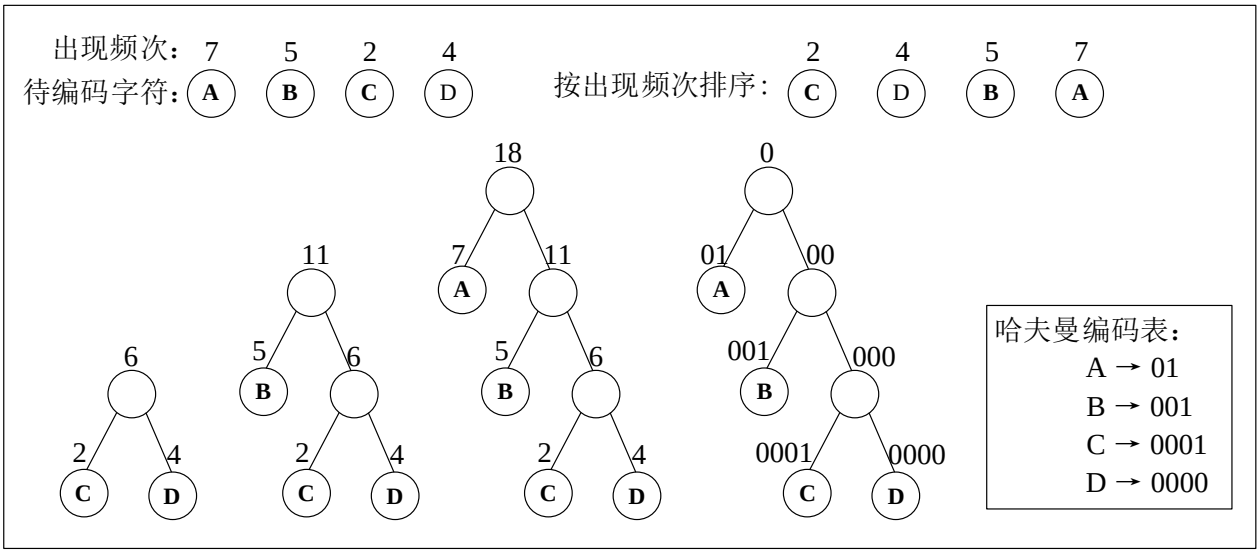


图 2.9 哈夫曼编码

哈夫曼编码的实现逻辑清晰且编解码速度极快，因此是目前最为常见的熵编码方式之一，但是由于哈夫曼编码的码长划分并不是严格按照字符出现频率的进行等比例划分，因此哈夫曼编码从理论上就无法无限逼近香侬熵[41,42]，从而无法实现理论最佳编码与最大的压缩比。

2.3.2 算术编码

算术编码[43,45]通过一个[0,1)区间内的小数 X 实现对信源字符序列的编码，它的基本实现方式如图 2.10 所示，通过将每个字符按照出现的频次在[0,1)区间内划分出对应的区间，然后按照字符出现的顺序逐次乘上对应区间所占的比例不断的缩小区间，最后得到一个包含所有区间信息的浮点数，在解码时通过该浮点数所在的区间不断进行逆向的除法操作即可实现解码。算术编码的优点是他在理论上可以无限接近于熵编码的最优结果(相对于哈夫曼编码可以减少 22%左右的存储空间)，但由于算术编码需要的解码时间相对哈夫曼编码更长以及算术编码本身的专利限制，目前绝大多数的 JPEG 编码在熵编码环节使用的都是哈夫曼编码。

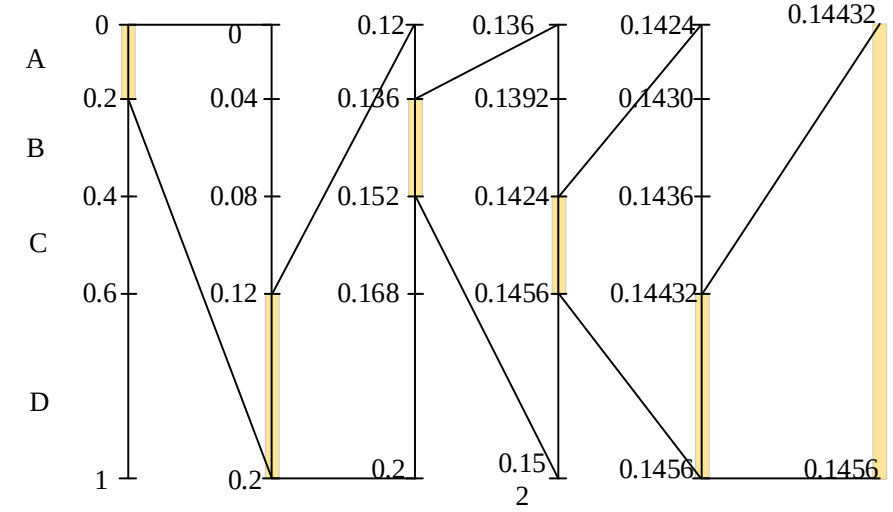


图 2.10 算术编码

2.3.3 ANS 编码

ANS 编码[46]的实现逻辑与算术编码类似，通过当前编码的状态 X_i 与当前字符 S 的概率分布推导出加入当前字符后的序列状态值 X_{i+1},以此类推最终得到整个序列的编码状态值即为结果。ANS 编码与算术编码的不同之处在于 ANS 的编码的初始状态值为零且编码结果始终为正整数，随着字符的增加编码的结果值单调递增而算术编码的结果是随着字符增加单调递减的浮点数。ANS 的编码公式如式子（2.7）所示，其中 2^n 表示设置的帧长度， f_s 为当前字符 s 对应的区间长度，x_i 表示未包含当前字符的编码结果，其中 $x_0=0$。CDF[s]=f_0+f_1+f_2+…+f_{s-1} 表示当前字符 s 区间的起始偏移位置，$x_i \bmod f_s$ 表示取余操作确定当前字符的具体区间内偏移量。通过对应的逆向解码公式（2.8）就可以得到相应的解码结果。ANS 编码与解码公式如下所示：

$$C(x_i, s_{i+1}) = [\frac{x_i}{f_s}] * 2^n + CDF[s] + (x_i \bmod f_s) \tag{2.7}$$

$$x_i = D(x_{i+1}) = f_s * [\frac{x_{i+1}}{2^n}] - CDF[s] + (x_{i+1} \bmod 2^n) \tag{2.8}$$

ANS 编码过程中需要使用字符分布表以及状态转移表，字符分布表根据字符的分布概率映射到了对应的字符区间，状态转移表根据当前状态与字符跳转到下一个状态。下图与表为 ANS 编码示例，其中图 2.11 表示根据待编码字符概率分布生成的字符分布区间，表 2.2 为对应的状态转移表。

如图 2.11 所示，在本例中设定每一帧区间长度为 10，待编码字符根据自身的占比可以得到各个字符的区间长度为 fs={3,3,2,1,1}区间的起始位置 CDF[s]={1,4,7,9,10}。假定需要编码的字符串为“BABDCE”则基于编码状态转移表初始状态 x0=0，初始 B 字符对应结果为 4，

x1=4 时 A 字符对应结果为 12,以此类推可以得到最终的编码过程为 0→4→12→44→449→2248→22490，这也正对应编码公式（2.6）的结果。ANS 的提出者 Jarek Duda 在论文[47]中通过实验详细的比较了三种熵编码的编码速度与压缩率，实验结果说明了相较于前面两种熵编码，ANS 编码的优势在于：在压缩率上 ANS 编码的压缩与算术编码类似可以无限接近熵编码的理论值，而在解码的速度方面 ANS 编解码的速度接近于哈夫曼解码的速度。因此 ANS 的出现彻底解决了熵编码在解码速度于压缩率之间的权衡问题，除此之外由于 ANS 编码本身完全开源，因此不存在由于专利限制影响算法普及的成本问题。

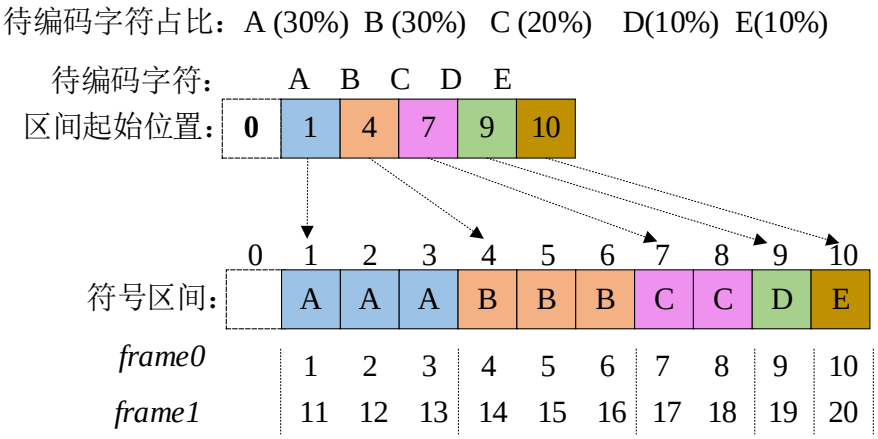


图 2.11 字符分布区间表

表 2.2 ANS 编码状态转移表

		Current state															
		strt	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
A	3/10	1	2	3	11	12	13	21	22	23	31	32	33	41	42	43	…
B	3/10	4	5	6	14	15	16	24	25	26	34	35	36	44	45	46	…
C	2/10	7	8	17	18	27	28	37	38	47	48	57	58	67	68	77	…
D	1/10	9	19	29	39	49	59	79	89	99	…						
E	1/10	10	20	30	40	50	60	80	90	…							

2.4 JPEG 数据段结构分析

JPEG 文件除了存储图像的压缩数据外还会额外存储一些数据，这些数据包含如量化表、哈夫曼表等图像编解码的过程信息以及图像的具体尺寸、编码模式等图像的参数信息。因此 JPEG 设计了不同的数据段分别存储不同的数据，并对整体的数据段结构做了规定，本小节主要分析整体的数据段结构并对每一段的存储内容做简要的介绍。

2.4.1 数据段布局

JPEG 文件包含了七个不同的数据段分别存储不同的图片信息，为了对每个数据段进行区分在每个数据段的起始位置包含了数段标记代码。标记代码由两个字节的数据构成，第一个字节为 0xFF，当读取文件发现 0xFF 时就会对紧接着的下一个字节进行分析，若第二个字节

为非 0x00 或 0xFF 的数据，则当前位置为一个段起始标准位。各个段标记与标记代码的对应关系如下表 2.3 所示:

表 2.3 JPEG 数据段标记

标记	标记代码	意义
SOI	0xFFD8	图像开始
APP0	0xFFE0	应用程序保留标记 0
DQT	0xFFDB	定义量化表
SOF	0xFFC0	帧图像开始
DHT	0xFFC4	定义哈夫曼表
SOS	0xFFDA	扫描开始
EOI	0xFFD9	图像结束

2.4.2 数据段内容介绍

在众多的数据段中本文主要涉及 SOF0 和 SOS 段，因此对这两段做重点介绍，而对其他段做一个简要介绍。SOI(Start Of Image)段表示 JPEG 图像开始，仅包含标记代码数据不包含其他数据。APP0(Application 0)段仅用于存储当前图像的缩略图信息，DQT(Define Quantization Table)段存储的量化与反量化过程所需要使用的量化表，DHT(Difine Huffman Table)段保存哈夫曼编解码过程中所使用的哈夫曼表，EOI (End of Image)仅用于表示图像结束。具体的图像压缩数据存储在图像帧数据中，图像帧数据由两个数据段构成，分别是帧头描述段 SOFn(Start of Frame)和扫描数据段 SOS (Start of Scan)。

SOFn 表示一个颜色分量的帧描述信息，其中 n 表示了对应的 SOS 段采用的编码模式，具体的对应关系如下表 2.4 所示，不同的编码模式采用的编码过程与熵编码方式有所不同。

表 2.4　SOFn 标记说明

标记	编码过程	熵编码
SOF0	基线顺序编码	哈夫曼编码
SOF1	扩展顺序编码	哈夫曼编码
SOF2	渐进式编码	哈夫曼编码
SOF3	无损编码	哈夫曼编码
SOF9	扩展顺序编码	算术编码
SOF10	渐进式编码	算术编码
SOF11	无损编码	算术编码

SOFn 段除了表示具体编码方式还包含了当前颜色分量的描述信息，如下表 2.5 所示，SOF0 段包含了当前颜色分量系数的位长信息，图像的高度与宽度信息，图像总的颜色分量信息，当前颜色分量的 ID、对应的量化表 ID 和与之对应的水平与垂直采样因子，这些参数都是完成编解码过程的必要信息。

表 2.5 SOF 段数据说明

字段号	意义	长度	值说明
0	标记代码	2 个字节	固定值 0xFFC0
1	数据长度	2 个字节	1~6 字段的总长度
2	精度	1 个字节	每个数据样本的位数，通常为 8
3	图像高度	2 个字节	图像高度（单位：像素）
4	图像宽度	2 个字节	图像宽度（单位：像素）
5	颜色分量数	1 个字节	当前图片的颜色分量数（通常为 3）
6	颜色分量信息	颜色分量数×3 （通常为 9 个字节）	颜色分量 ID 水平与垂直方向采样因子 当前颜色分量量化表 ID

SOS 段除了保存压缩的图像数据外也会保留一部分描述性信息，这些描述信息一部分与 SOF 段相同另一部分则是对 SOF 段进行的补充。如下表 2.6 所示，在 SOS 段中包含对应的颜色分量 ID、DC 与 AC 系数所使用的哈夫曼表 ID 以及最后的图像压缩数据段。

表 2.6 SOS 段数据说明

字段号	意义	长度	值说明
0	标记代码	2 个字节	固定值 0xFFDA
1	数据长度	2 个字节	1~4 字段总长度
2	颜色分量数	1 个字节	与 SOF 字段 5 值相同
3	颜色分量信息	2 个字节	颜色分量 ID DC/AC 系数表号
4	压缩数据图像数据	3 个字节	普选择开始标记 0x00 普选择结束标记 0x3F 普选择在基本 JPEG 中固定为 0x00

2.5 本章小结

本章首先介绍了并行化二次压缩方案的基本框架，说明了二次压缩的主要流程以及并行化优化的主要内容。然后详细的分析了二次压缩方案的理论基础，包含 JPEG 编码标准以及二次压缩涉及的三种熵编码算法，从理论层面阐明了二次压缩方案可以实现更高压缩率的原因。最后简要的分析了 JPEG 文件中各数据段的存储格式与存储内容，以便于理解后续并行化实验所操作的对象。

第 3 章 多线程并行化

为了对二次压缩解码过程中的熵编码环节进行了多线程并行化优化，本章首先对多线程编码方案实现过程中的主要问题进行了分析，主要问题包括：编码环节间存在的依赖问题、线程间任务量的均衡化问题以及对于编码结果中非字节对齐数据的写入问题。然后，基于上述的问题分析，本章设计了基于“哈夫曼切换词”的子图分割式多线程并行化编码方案。最后，通过实验验证了本章多线程并行化编码方案的整体提速效果。

3.1 多线程并行化难点分析

本节首先分析了多线程编码方案实现过程中存在的编码环节依赖、线程任务均衡以及对于编码结果中非字节对齐数据的写入问题。针对这些问题，本文通过分析图像数据的分布特点以及引入“哈夫曼切换词”，解决了上述改造过程中遇到的问题。

3.1.1 并行化方案选择

多线程并行化主要是利用多核 CPU 的并行处理来实现，目前多线程实现并行化主要有两个实现方向：

(1)通过对任务进行分割，将整个编码任务分割为若干子任务，然后将每个子任务交由不同的线程单独完成；

(2)通过对流程进行分割，将整个编码流程分割为若干的子流程，然后将每个子流程并交由相应的线程实现；

brn 文件解码程中的串行熵编码过程如下图 3.1 所示，包含了 Zig-zag 序列化、RLE 编码、VLI 编码以及哈夫曼编码四个子环节，每个子环节都依赖于上一个子环节的编码结果，因此环节间存在着依赖问题。在整个串行编码过程中均是以 Block 数据块作为最小的编码单元，通过对 Block 数据块中的系数逐个编码实现由 brn 文件解码为 JPEG 文件的过程。

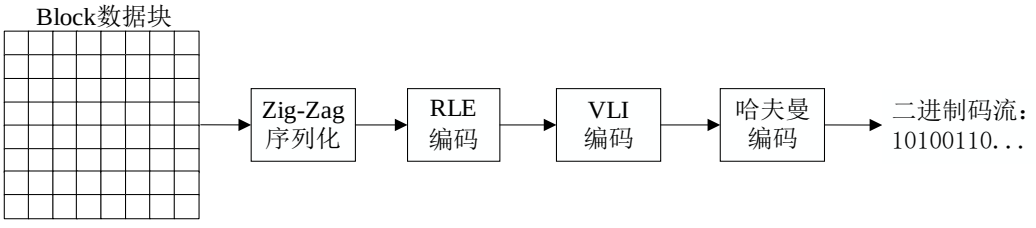


图 3.1 Block 熵编码

本文分别设计并比较了两种多线程并行化优化方案：一是通过对编码流程的分割实现类似 CPU 流水线式的并行化优化；二是通过对整体编码任务进行分割，将整个图片分割为若干的子图数据，将每个子图数据分别交由一个线程完成子图的编码任务。下面将对两个并行化方向进行具体的展开介绍。

CPU 流水线[48]式的并行化优化如下图 3.2 所示，将编码过程的每个子环节分别交由一个线程负责。当一个线程完成对当前 Block 的编码工作后，将当前编码完成的数据交由负责下一个环节的线程，然后当前线程开始对下个 Block 数据进行编码。

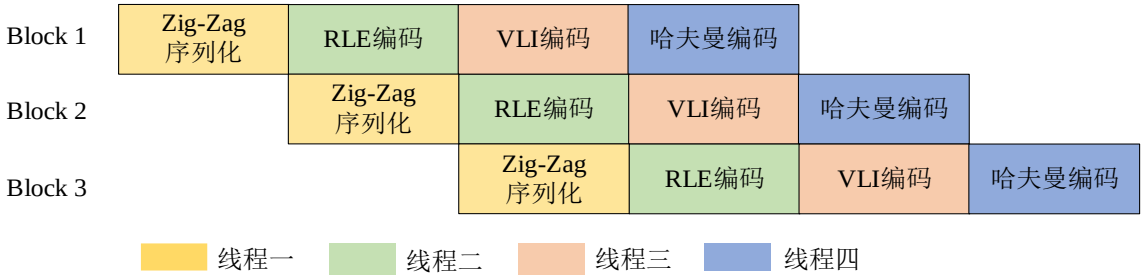


图 3.2 流水线式任务分割

流水线方案的优点是每个线程负责一个编码过程中的子环节，因此单个线程任务较为清晰，编码完成后的写入交由同一个线程进行不会产生写入对齐的问题，但是该方案的主要缺点是由于不同子环节计算量的差异，各个线程的完成时间无法一致，导致计算量小的子环节会频繁的进入等待状态，因此该方案需要解决线程间大量的同步问题。

多线程的另一种实现方式是通过对任务的分割，如下图 3.3 所示，通过将图像数据分割为若干个子图块，每个子图块的数据交由一个线程单独进行完整的编码流程。该方案的优点是由单个线程完成整个子图的编码工作，不存在线程与线程的任务因为需要衔接而产生的等待问题。但该方案同样存在着问题，首先要解决每个线程的图像分割问题，线程编码前必须知道自己所负责的编码区域，其次编码完成后每个线程都需要知道当前线程负责区域的编码结果在 JPEG 中的写入起始位置，最后由于每个线程的编码结果大概率为非字节对齐，因此要解决线程编码结束后的非字节对齐写入问题。

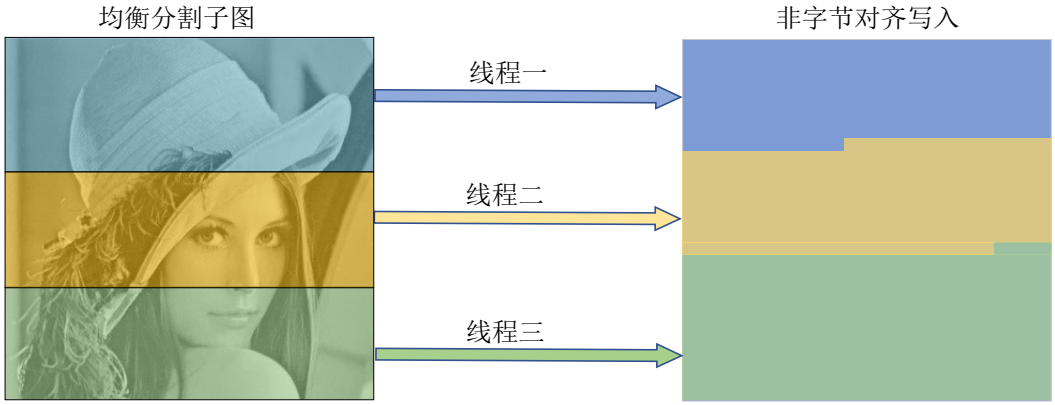


图 3.3 子图均衡分割

综上所述，对于熵编码过程而言多线程并行化的难点主要集中在三个方面：

(1) 如何解决流水线并行编码改造过程中遇到的子环节同步等待问题。

(2) 如何实现不同线程任务量的均衡性，以实现多线程并行化的最佳效率。

(3) 如何在子图分割式并行化下，解决对非字节对齐的子图编码结果的写入与拼接问题。

由于串行编码各子环节的编码时间难以实现统一，会导致多线程同步过程产生等待时间的问题。为了尽可能快的实现多线程并行化编码，最终本文选择通过图片分割的方式实现多线程并行化编码，下面两个小节会围绕图片分割式多线程方案中的子图均衡分割与非字节写入对齐问题进行具体展开并给出相应的方案。

3.1.2 子图均衡分割

为了使多线程并行解码的时间尽可能的缩短，需要使每个线程负责的编码范围大小基本一致。每个编码过程处理的数据是以最小编码单元(Minimum Coding Unit，MCU)作为编码的最小单位，MCU 是一个大小为 16×16 数据矩阵代表着相应位置的像素数据，因此通过划分编码的 MCU 范围即可以实现对图像的均衡分割。以一张 3840×2160 像素的图片为例，在采样模式为 YCrCb4:4:4（全采样）的情况下为例，如下图 3.4 所示整张图片被分割为了长为 240 宽为 135 的 MCU 矩阵，其中每个 MCU 包含了 16×16 个对应像素的数据，一个 MCU 内部相当于 2×2 个 Block 数据块。由 2.3 小节对 JPEG 数据段的分析可知，在 JPGE 标准中帧起始段(Start of Frame，SOF)包含了一张图片的长度与宽度的像素量信息，编码过程中可以通过这两个参数可以计算得到具体的 MCU 行数以及每一行所包含的 MCU 个数。

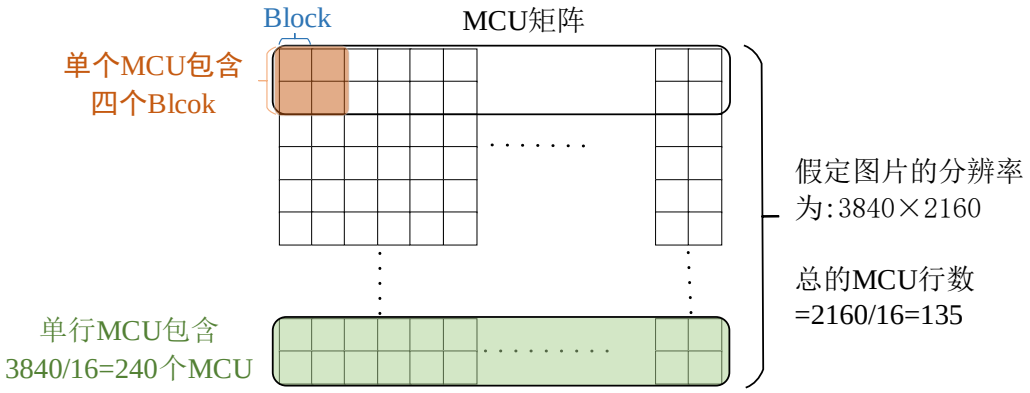


图 3.4 MCU 与 Block 布局

每个线程在编码前可以通过帧起始段参数计算出当前图片总的 MCU 行数与列数，因此每个编码区域的可以通过 MCU 行数据的平均分配实现。假定当前线程负责编码的子图所在的 MCU 行起始位置号为 N 以及结束行号为 M，则当前线程实际的编码任务量为(M-N) ×单个 MCU 行所包含的 MCU 个数。

综上所述为了尽可能的使 N 个线程间的任务量均衡，应该让每个线程在编码前了解到自己负责的 MCU 行区间。那么多线程并行方案中的任务均衡问题就转换成了如何让每个线程得到正确的 MCU 起始行参数与结束行参数的问题。

3.1.3 非字节对齐数据写入

由于在熵编码过程中涉及的 RLE 编码、VLI 编码以及哈夫曼编码都属于不定长编码，因此每个字符所对应的实际符号长度并不固定。这表示在熵编码编码完成后，如下图 3.5 所示，需要写入的编码结果大概率上是非字节对齐的（二进制位数不能被 8 整除），而计算机的存储

设备要求读写均必须要以字节为单位（每次写入 8 位二进制），这就带来了多线程并行编码的第二个主要问题即非字节对齐的写入问题。

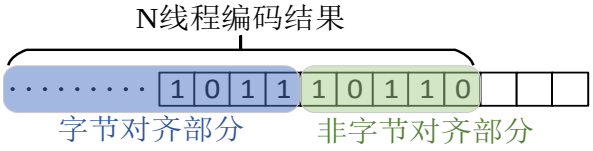


图 3.5 非字节对齐码流

在传统的单线程 JPEG 编码过程中也会面对非字节对齐问题，JPEG 标准选择的解决方案是通过对最后非字节对齐的部分通过补零进行补齐后进行写入。该方案对于单线程而言是可行的，而在多线程环境下相邻的两个子图的二进制流必须是严格对齐的，一旦中间出现多余的补齐字符就会导致最终生成的图片无法正常解码。因此在多线程写入过程中 N 线程不能直接写入非字节对齐部分，否则就会造成多余数据位的掺入。如下图 3.6 所示，除了第一个子图线程外，必须保证其他线程的写入起始位置与上一子图写入的结束位置能够完美的进行非字节对齐，且写入时只能够通过字节对齐的方式进行写入。

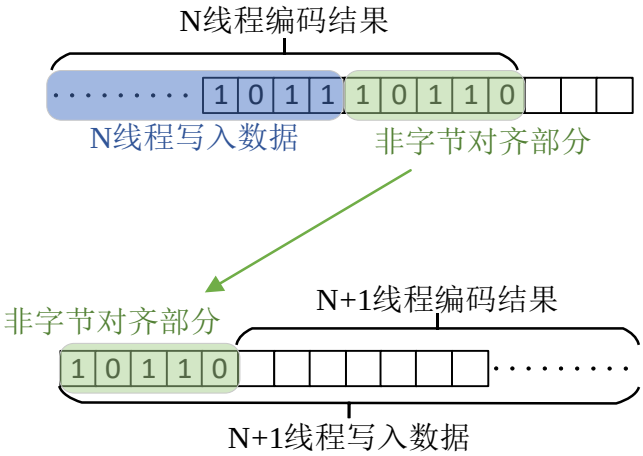


图 3.6 非字节对齐数据转移

综上所述，要解决好多线程环境下的非字节对齐数据的写入与对齐问题，就必须首先让每个线程了解写入的起始字节位置，在写入过程中首先要写入上一个子图线程编码结果中的非字节对齐部分。如上图 3.6 所示，为了实现当前线程的编码结果与上一子图的编码结果能实现无缝衔接，上一子图的非字节对齐部分必须要通过字节对齐的方式写入到当前子图的数据流头部。

3.1.4 哈夫曼切换词的设计与实现

综合 3.1.2 与 3.1.3 的分析多线程并行化的实现必须使每个线程在编码前获取到一些必要的数据，因此本文参考了 Dropbox 公司提出的“哈夫曼切换词”[49](Huffman Handover Words，HHW)的概念，设计了一个称为哈夫曼切换词的数据结构。本方案的大致思路是在 brn 文件的编码阶段将编码过程中得到的有用信息通过哈夫曼切换词存储到 brn 文件中，在 brn 解码过程中每个线程可以通过哈夫曼切换词获取必要的编码信息，实现多线程并行编码

哈夫曼切换词的具体结构，如下图 3.7 所示哈夫曼切换词主要包含两个部分：首先是用于定位编码位置的 MUC 编码信息，主要包含了 MCU 起始位置的行号以及结束位置的行号，以及该子图实际需要编码的字节数信息；然后哈夫曼切换词还包含了用于保存编码结果的写入信息，用于保存编码结果的起始位置以及与当前子图相连的上一个子图的非字节对齐量。

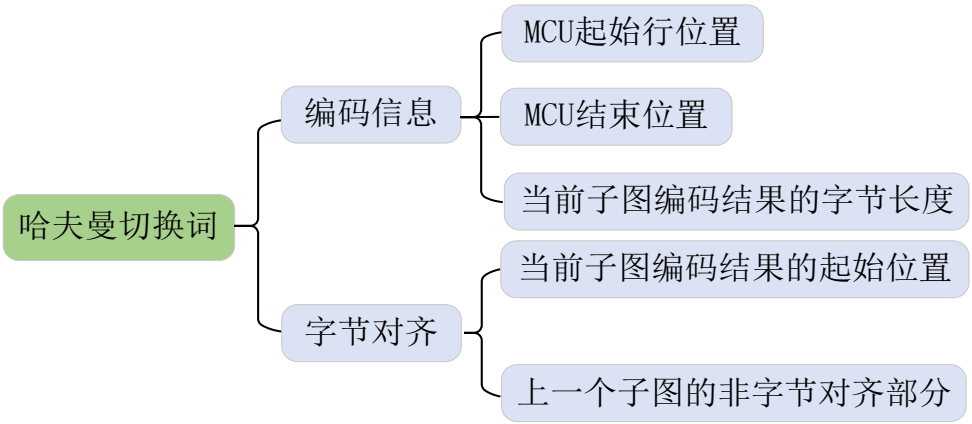


图 3.7 哈夫曼切换词结构

在多线程方案的编码过程中系统会为每个子线程写入一个哈夫曼切换词信息，在解码过程中通过哈夫曼切换词的使用，每个线程可以快速了解到当前编码的子图位置以及编码完成后数据的保存位置，从而彻底解决了上述所说多线程编码问题。

3.2 多线程并行化方案设计

3.2.1 多线程并行化总体方案

通过对多线程并行化解码的难点分析可知，多线程并行编码的主要难点集中在两个方面，第一个方面是编码前如何对子图进行均衡分割，第二个方面是编码后如何使多个非字节对齐的编码结果实现写入结果对齐。为了能够解决这两个方面的问题，本文设计并引入了“哈夫曼切换词”这一自定义的数据结构，通过在图片编码过程中记录并保存解码过程中所需要使用的参数信息并在解码过程中使用，实现了在多线程环境下的并行解码。综上所述，对于多线程并行化解码的方案改造涉及二次压缩过程中编码与解码两个部分，需要同时对编码与解码过程进行改造。

为了获取必要的数据生成哈夫曼切换词本文对二次压缩的编码过程进行了改造，如下图 3.8 所示，在编码过程中编码器首先会从 JPEG 图片的 SOF 数据段中解析出图片的尺寸信息，根据图片的尺寸信息可以计算出图片 MCU 的总行数为 Lines，并根据尺寸信息决定并行解码需要使用到的线程数量 N，然后可以求出每个线程负责 MCU 行的具体区间。在得到每个子图的 MCU 行区间后，通过 JPEG 二次解码过程中熵解码可以统计出完成当前子图 MCU 对应的解码数据长度，该数据长度即为并行编码还原到 JPEG 图片时编码结果的长度。通过将以上的信息写入到对应的哈夫曼切换词中，可以有效的解决 brn 并行解码过程遇到的问题。

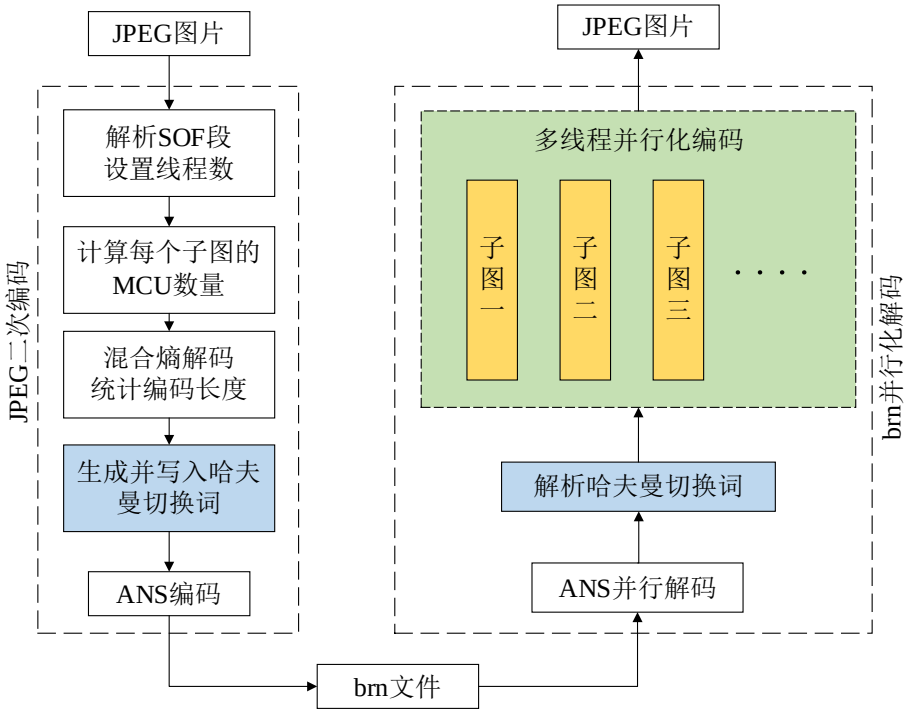


图 3.8 多线程并行化编解码方案

brn 文件的并行解码过程如上图 3.8 所示，首先会对 brn 文件进行 ANS 解码还原出 MCU 行数据，然后根据哈夫曼关键字的数量解析出需要的线程数，每个线程通过分析对应的哈夫曼切换字可以找到对应的子图区域并得到编码结果存储的起始位置以及上一子图的非字节对齐分量。

3.2.2 编码模式分析

在 JPEG 编码标准中包含两种编码模式，Baseline 模式和 Progressive 模式[50]，两种模式均能解码得到相同的图片。两种模式的解码过程如下图 3.9 所示，Baseline 模式下解码采取从上到下逐行进行解码的方式而 Progressive 模式下采取的是多轮解码的方式，第一轮解码先还原出图像的大致轮廓，之后的解码过程不断的补充图片的细节信息。为了能够使多线程并行化能够适配所有的模式，本方案为两种编码模式分别设计了不同的多线程并行写入方案。

(a)Baseline模式解码　　(b)Progressive模式解码

图 3.9 编码模式对比

Baseline 模式是 JPEG 标准中最常见的编码模式，也是目前绝大多数的 JPEG 图片采用的写入模式，它的写入采取顺序结构，对不同的颜色分量系数(Y/Cr/Cb)的编码结果通过一次写

入保存到图片文件中。Baseline 模式的优点是写入效率高且写入流程相对简单仅包含一次写入过程。

如下图 3.10 所示，在 Baseline 模式下编码 MCU 数据行时通过对各颜色分量的 MCU 数据进行合并为一个编码序列，在编码过程中 Baseline 的编码顺序由编码序列内的颜色分量顺序决定，最后按照序列中的颜色分量顺序保存编码数据。

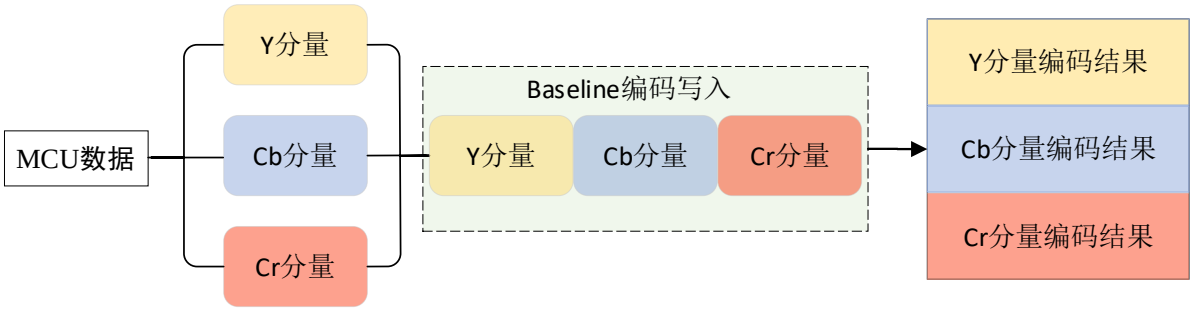


图 3.10 Baseline 模式写入流程

Progressive 模式适用于网络传输较大的 JPEG 图片的场景，它与 Baseline 模式最大的区别在于 Progressive 采用的是多次扫描写入。Progressive 模式下的数据编码写入过程如下图 3.11 所示，Progressive 包含十轮编码过程，每一轮编码过程都会选择不同颜色分量的不同系数的不同数据位。前几轮的编码过程主要负责将图像的大致轮廓信息进行编码，之后负责编码图像中的细节信息，Progressive 写入顺序与编码时的顺序一致。多轮编码的好处是即使在加载图片过程中即使遇到网络较慢的情况，也可以先看清图像的大致轮廓，一些网站加载较大的图片时往往会采取 Progressive 模式。

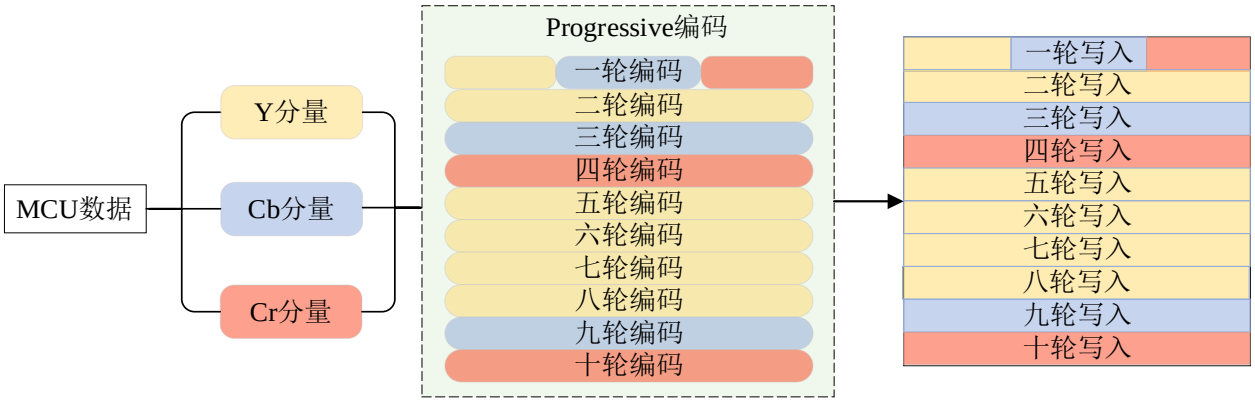


图 3.11 Progressive 模式写入流程

Progressive 对与不同的颜色分量(Y/Cr/Cb)分别采取多次写入，对于每一个 Block 数据块每次写入不同的系数的不同位数从而实现图像由模糊变得清晰的效果。在写入过程中通过五个系数来确定当前的写入数据范围，其中 Component 确定当前的颜色分量，Ss 和 Se 确定当前写入系数在 Block 中的序号，Al 和 Ah 系数确定写入系数的具体数据位。Progressive 模式下分十次写入过程如下表 3.1 所示，Al 表示编码的最低起始位而 Ah 表示编码的最高位，如 Ah=0 且 Al=1 时，表示当前系数的编码位数为从最高位到最低位除了 0 位以外的位数即从右

至左的一到十五位(系数位长为 16)。

表 3.1 Progressive 模式写入范围

编码轮数	颜色分量	Ss	Se	Ah	Al
第一轮	Y,Cr,Cb	0	0	0	1
第二轮	Y	1	5	0	2
第三轮	Cb	1	63	0	1
第四轮	Cr	1	63	0	1
第五轮	Y	6	63	0	2
第六轮	Y	1	63	2	1
第七轮	Y	0	0	1	0
第八轮	Y	1	63	1	0
第九轮	Cb	1	63	1	0
第十轮	Cr	1	63	1	0

3.2.3 子图编码方案介绍

3.2.3.1 Baseline 模式子图编码方案

Baseline 模式下每个线程的编码过程如下图 3.12 所示，编码前首先要根据当前线程的哈夫曼关键字确定编码 MCU 的行范围，然后根据当前的采样模式与颜色分量确定 MCU 内的采样间隔，最后根据行范围与采样间隔可以得到实际编码的 Block 范围，循环编码 Block 数据块直至编码完成结束当前线程。

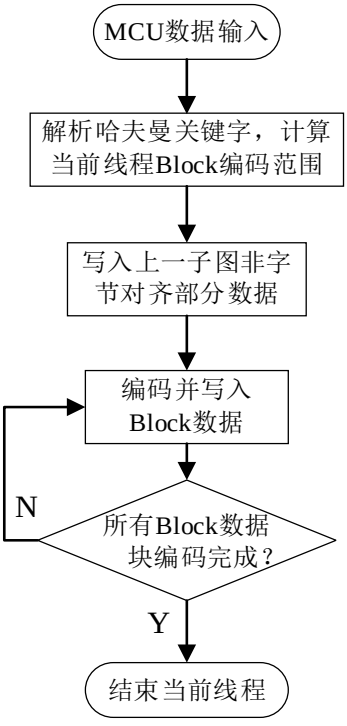


图 3.12 Baseline 模式编码流程图

3.2.3.2 Progressive 模式子图编码方案

Progressive 模式下单个线程的编码流程如下图 3.13 所示，在 Baseline 编码的基础上加入了对 Block 编码范围的解析。每个线程在编码前同样是通过解析哈夫曼关键字确定当前的 Block 编码范围，然后写入非字节对齐的数据，通过解析 Ss,Se,Al,Ah 系数可以确定编码的颜色分量与具体的系数范围。最后通过循环编码 Blcok 矩阵中的指定系数，直至所有 Block 矩阵编码完成结束当前线程。

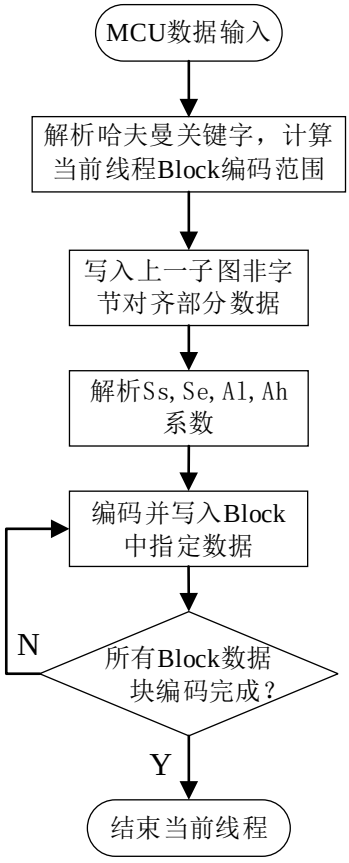


图 3.13 Progressive 模式编码流程

3.3 多线程并行化实验

3.3.1 实验环境与实验数据集

3.3.1.1 实验环境

多线程并行化的二次压缩 JPEG 实验的环境与实验所针对的 JPEG 模式如下表 3.2 所示：

表 3.2 实验环境与编码模式

处理器	AMD Ryzen 7 4800U
操作系统	Windows11
IDE	Visual Studio2019
编程语言	C++
JPEG 模式	Baseline/Progresive

3.3.1.2 实验数据集介绍

为了测试并行化二次压缩算法对于不同类型 JPEG 图片的二次压缩效率，本文从图片的分辨率、采样格式以及编码模式三个角度设计并制作了相应的测试数据集。为了测试并行化二次压缩算法对于不同分辨率 JPEG 图片的压缩效果，本文使用四类不同分辨率的手机分别拍摄了 1000 张图片作为实验的测试数据集。为了检验不同采样格式以及编码模式的 JPEG 图片在并行化二次压缩算法下的压缩效率，本文通过编写相应的脚本生成了相应的对照实验组。由脚本生成的 JPEG 图片仅在编码模式或采样格式上有相应的差异，其他 JPEG 属性均保证相同，形成可控的对照实验条件。实验过程中实际使用的图片类型于数量如下表 3.3 所示：

表 3.3 实验数据

分类标准	具体类型	图片数量
分辨率	2976*3968	1000
	3024*4032	1000
	3000*4000	1000
	3072*4096	1000
采样格式	YCrCb4:4:4	4000
	YCrCb4:2:0	4000
编码模式	Baseline	4000
	Progressive	4000

3.3.2 多线程并行化实验与实验结果分析

3.3.2.1 多线程二次压缩压缩率对比实验

本项实验主要是为了测试 JPEG 图片经过本文的二次压缩后的平均压缩率(Average Compression Ratio，ACR)的变化，来评价二次压缩相较于原始 JPEG 图片的压缩率提升幅度。压缩率的熟悉表达式如下式（3.2）所示，其中，S_j 表示 JPEG 原图所占用的存储空间大小，SS 表示经过二次压缩得到的 brn 文件所占用的存储空间，平均压缩率的数值越小说明二次压缩相较于原有 JPEG 的压缩效果越好。

$$ACR = \frac{S_S}{S_j} \times 100\% \tag{3.2}$$

本实验设定在 Baseline 模式下采样格式为 YCrCb4:2:0 的条件下，选择了四类主流手机分辨率图片各 1000 张作为测试图片，通过对比不同分辨率图片在单线程、双线程、三线程以及四线程编码算法编码后的平均压缩率来说明二次压缩算法的实际压缩效果，压缩率测试的结果入下图 3.14 所示：

通过对比压缩率可以发现单线程与多线程压缩后的 brn 文件大小相较于原始的 JPEG 图片至少减少了 21%的内存空间大小，对于特定分辨率如 3024*403233 的 JPEG 图片，二次压缩的压缩率可以达到 55%左右。由于多线程编码过程中需要写入哈夫曼关键字，因此多线程

编码压缩率相较于单线程编码压缩率要普遍高 1%左右，符合本文的理论预期值。

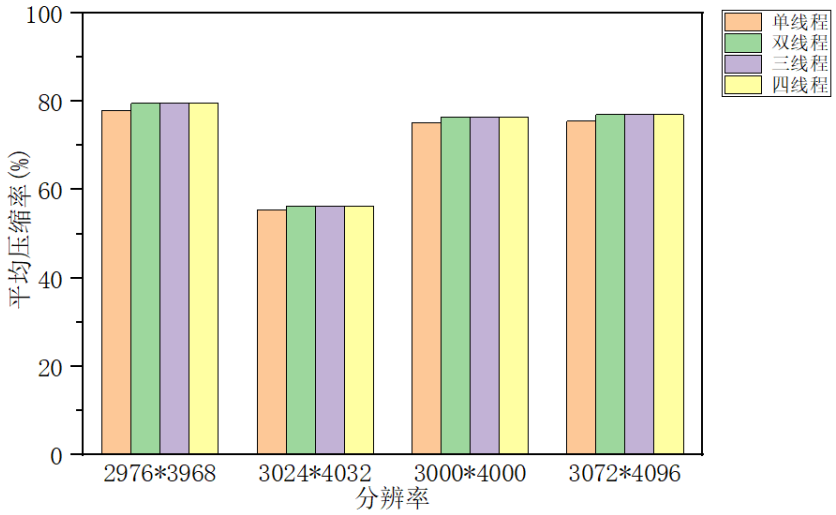


图 3.14 多线程并行压缩平均压缩率对比

3.3.2.2 多线程并行解码性能测试

对于图片编解码算法而言，实际解码的速度会影响对于整个编解码算法的日常使用体验。为此本实验主要研究对多线程并行化对于 brn 文件解码速度的影响，本实验设定在 Baseline 模式下采样格式为 YCrCb4:2:0 的条件下，选择了四类主流手机分辨率图片各 100 张作为测试图片，通过测试在不同线程数下的平均并行度(Average Parallelism Degree，APD)的方式来评价并行编码的优化程度，平均并行度的数学表达式如下式（3.3）所示。其中，T1 表示在单线程下 100 张测试图片的平均编码时间，Tp 表示指定线程数下 100 张图片的平均解码时间，平均并行度的数值越小说明相对解码时间越短，算法的并行度越高。

$$APD = \frac{T_p}{T_1} \times 100\% \tag{3.3}$$

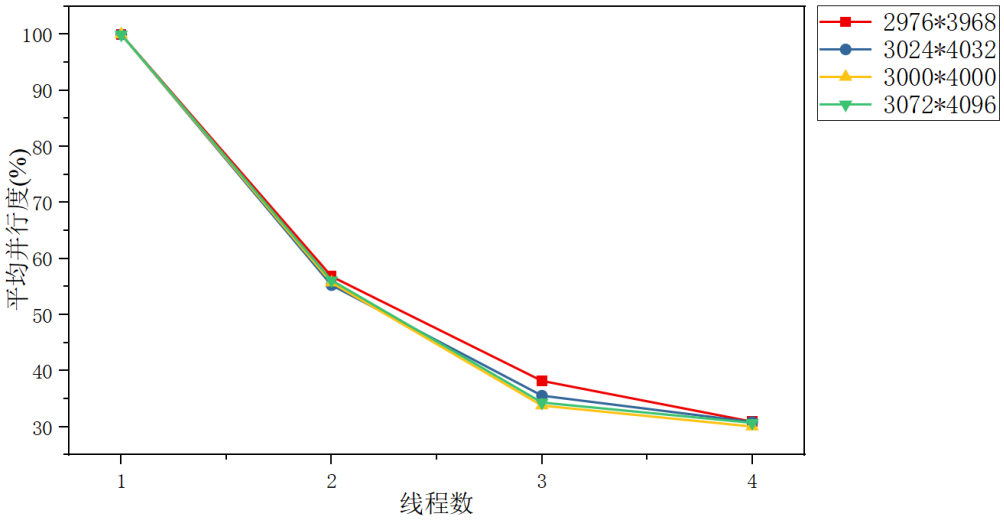


图 3.15 不同线程平均并行度对比

如图 3.15 所示，通过对比不同线程数的平均并行度可以发现，从线程数从 1 到 2 的过程中平均并行度的降幅最大，多线程并行化提升的效果最为明显，而当线程数从 3 提升到 4 的过程中并行度的降幅较小，多线程并行化提升效果微弱。线程数为 4 时的实际平均并行度远高于理论平均并行度值 25%，这是由于在实际编码过程中由于 CPU 的物理核心处于多线程竞争关系，因此伴随线程数的增加，线程调度与同步会造成 CPU 额外的性能开销，最终导致整体解码时间变长。

3.3.2.3 采样格式对比测试

二次压缩过程根据 JPEG 图片的降采样格式确定单个 MCU 内部的 Block 数量，因此采样格式的不同可能会对二次压缩并行度造成影响，为了测试二次压缩对于 JPEG 图片在不同采样格式与不同采样模式下的平均并行度，本实验在双线程模式下，分别对目前主流的两类采样格式 YCrCb4:4:4 与 YCrCb4:2:0 进行了平均并行度对比实验。

为了分析采样格式对并行度的影响，本实验选择了四类主流手机分辨率图片各 100 张作为测试图片，通过脚本软件分别生成了 YCrCb4:4:4 与 YCrCb4:2:0 采样格式的对照图片，然后对比两种采样格式图片在双线程下的平均并行度，检验不同采样格式对于二次压缩平均并行度是否存在影响。实验结果如下表 3.4 所示：

表 3.4 不同采样模式平均并行度对比

分辨率	采样模式	数量	单线程平均编码时间(ms)	双线程平均编码时间(ms)	平均平行度(%)
2976*3968	4:2:0	1000	59.45	34.57	58.14
	4:4:4	1000	94.82	55.23	58.25
3024*4032	4:2:0	1000	72.35	44.38	61.34
	4:4:4	1000	93.85	58.09	61.9
3000*4000	4:2:0	1000	70.95	42.56	59.98
	4:4:4	1000	98.26	60.79	61.86
3072*4096	4:2:0	1000	62.1	38.42	61.86
	4:4:4	1000	84.44	53.85	63.77

通过对表 3.4 的数据分析可得，在双线程模式下，YCrCb4:4:4 与 YCrCb4:2:0 采样格式的 JPEG 图片的平均编码时间都有显著的降低。从平均并行度的角度看，两种编码模式的平均并行度均在 60%左右，双线程的平均编码时间相较于单线程减少了 40%，说明了不同的编码模式对二次压缩的平均并行度不会造成重大影响。

3.3.2.4 编码模式对比测试

为了分析 Progressive 编码模式下的平均并行度，本实验选择了四类主流手机分辨率图片各 100 张作为测试图片，通过脚本软件分别生成了 Progressive 模式与 Baseline 模式的对比图片，然后对比两种编码模式图片在双线程下的平均并行度，检验不同编码模式对于二次压缩平均并行度是否存在影响。实验结果如下表 3.5 所示：

表 3.5 不同编码模式下平均并行度比较

分辨率	编码模式	数量	单线程平均编码时间(ms)	双线程平均编码时间(ms)	平均平行度(%)
2976*3968	Baseline	1000	115.5	53.79	56.86
	Progressvie	1000	655.25	364.35	55.6
3024*4032	Baseline	1000	114.11	50.44	55.29
	Progressive	1000	714.15	398.9	55.85
3000*4000	Baseline	1000	125.05	69.71	55.74
	Progressive	1000	6660.6	390.11	59.05
3072*4096	Baseline	1000	112.07	62.95	56.17
	Progerssive	1000	671.35	385.35	57.39

通过表 3.5 的对比试验可以发现双线程下 Progressive 模式与 Baseline 模式的图片的平均并行度均在 55%左右，符合理论的预期值，说明了在两种模式下本文给出的并行化优化方案均取得不错的并行化效果。

3.3.2.5 多线程二次压缩前后图像数据对比

为了验证 JPEG 图片在多线程二次压缩产生的 brn 文件可以无损的还原回到原始的 JPEG 图片，原图与二次压缩后还原图片的对比如下图 3.16 所示，通过对比可以发现二次压缩前后图片没有明显的图像质量差比。

原始图片

二次压缩后还原图片

图 3.16 二次压缩前后图像质量对比

为了进一步说明本文的二次压缩方案为无损压缩，本文使用了 beyond compare 软件对原始

JPEG 图片与二次压缩解码后的 JPEG 图片进行了十六进制的码流对比，图片码流的对比结果如图 3.17 所示，通过十六进制码流数据对比显示二次压缩前后的图像质量完全相同，达到了本文方案所说的无损还原的指标。

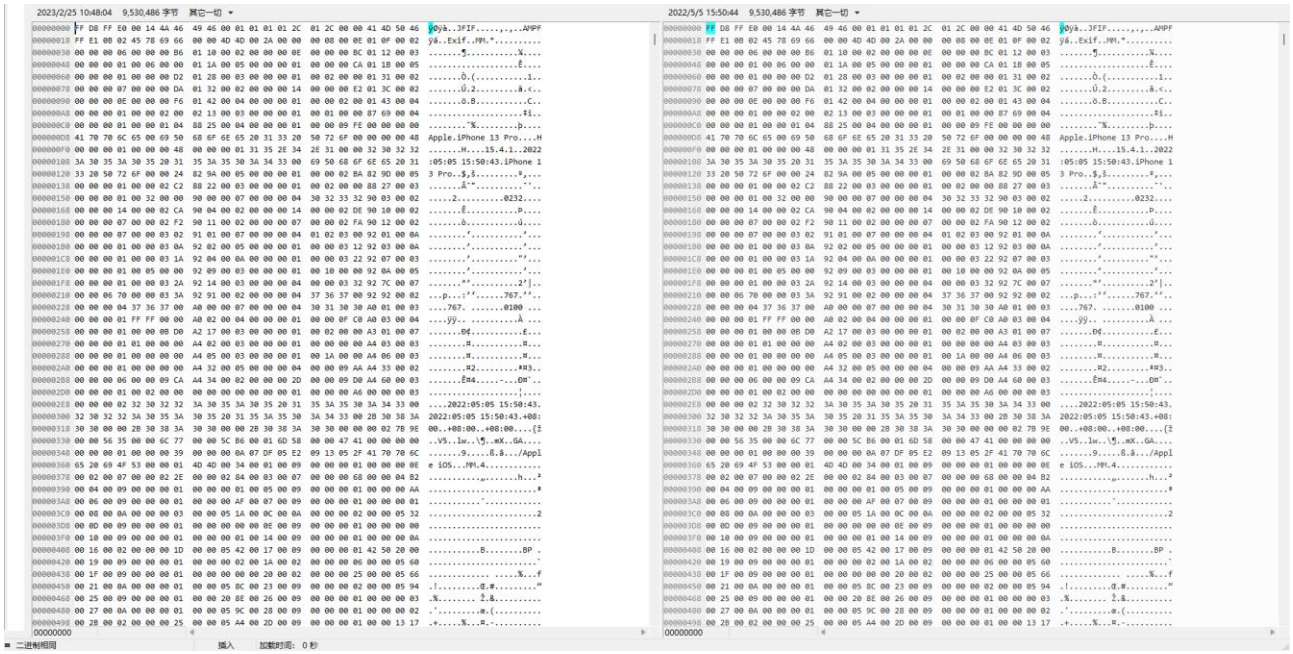


图 3.16 二次压缩前后图像码流对比

3.3.3 实验总结

本章测试了多线程并行化二次压缩方案在不同线程数下的平均压缩率与平均并行度。结果显示，多线程方案相较于单线程在平均压缩率提升了 1%左右，但在平均并行度上多线程方案在双线程、三线程、四线程下分别降低了 45%、60%、70%左右，有效的降低了上编码的编码时间。为了测试二次压缩的普适性，本章的实验分别对不同分辨率、不同采样格式、不同编码模式的图片进行了对比实验，实验的结果说明了本文提出的多线程并行化二次压缩方案面对不同类型的 JPEG 图片均取得了相同幅度的编码速度提升。

3.4 本章小结

本章首先分析了多线程编码方案主要的实现难点，得到了实现子图分割式多线程并行编码方案的两个主要难点：如何实现对子图的均衡分割以及如何实现对非字节对齐的码流的写入。为了解决这两个难点，本方案引入了哈夫曼切换词，通过哈夫曼切换词的设计与使用，实现了在编码环节提取出解码过程所需的相应参数，最终顺利完成了对解码过程中的熵编码环节进行并行化优化。然后，针对 JPEG 图片在 Baseline 模式与 Progressive 模式下的文件结构特点，本章分别设计并实现了两种模式下的多线程并行化方案。通过实验验证了两种方案对应各自模式均取得了理想的提速效果。最后，本章选取了四种主流手机分辨率的 JPEG 图片各 100 张作为实验数据集，对这些图片分别进行了不同线程下的二次压缩实验，实验的结果说明多线程并行化编码可以有效提升熵编码环节的编码速度。

第 4 章 SIMD 数据并行化优化

为了实现对 Block 数据块的数据并行化编码，本章设计了一种基于 SIMD 指令的数据并行化熵编码方案。该方案通过加入每个 Block 数据块的 Bitmap 信息与掩码信息，解决了数据并行化编码过程中的零系数统计与不同类型系数的统一编码问题。最后，本章通过实验说明了数据并行化编码方案对于熵编码速度的提升幅度。

4.1 SIMD 并行化的原理介绍

4.1.1 SIMD 数据并行化原理介绍

SIMD 技术即单指令多数据并行处理技术，当需要对多个标量数据执行相同操作时，通过 SIMD 数据并行化优化可以极大的提升操作的效率。单个标量数据操作与 SIMD 并行化操作的对比如下图 4.1 所示，在处理标量数据过程中每一个标量数据(A_1、A_2…)都需要单独进行乘法操作，而在使用 SIMD 的情况下，通过特定的硬件寄存器与并行处理单元的支持可以实现将若干标量数据合并为一个向量数据($A_1A_2A_3A_4$)，通过对该矢量数据进行统一的乘法操作，最终实现了数据并行操作，减少了重复标量操作的时间成本。

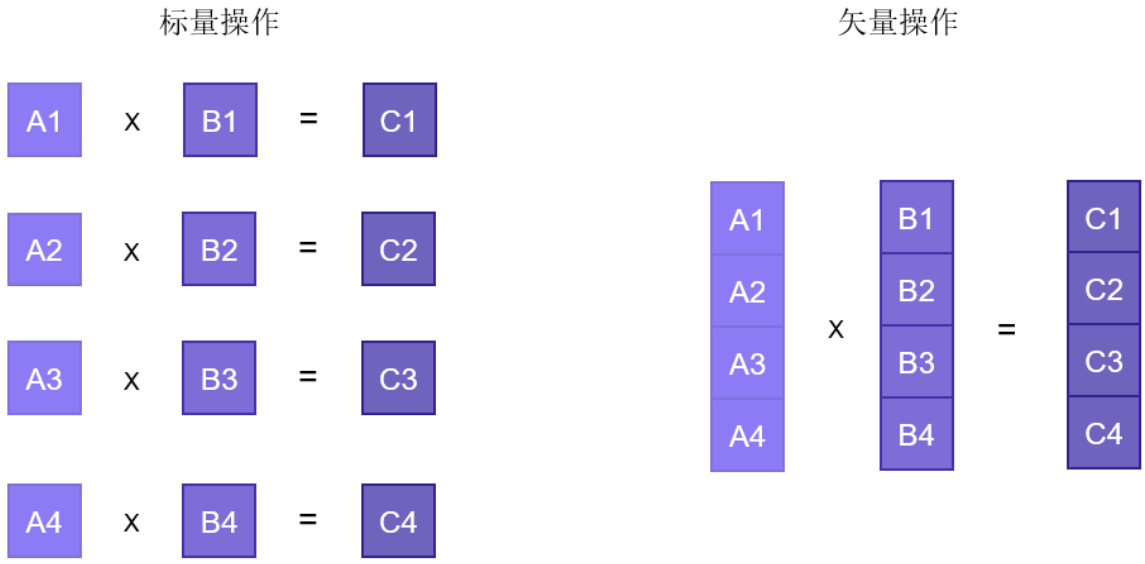


图 4.1 SIMD 并行化

由前文中对二次编解码流程的分析可知，编解码过程的实质是将图像数据分解为若干个 Block 数据块，然后对 Block 数据块中的数据逐一进行相应的二进制操作，最终得到相应的编码结果。在 Block 内部数据的编码过程中充斥着大量简单而重复的运算，且单个系数的位宽仅为 16 位可以方便的进行合并后，通过 SIMD 技术可以实现对一组数据的并行处理，编解码过程中的相关数据块的串行操作也可以通过 SIMD 技术改进为并行操作，可以有效的提升编解码的效率[51,52]。

4.1.2 X86 架构下的 SIMD 介绍

1997 年由英特尔公司首先推出了基于 X86 架构的第一代多媒体扩展指令集(Multi Media e Xtension,MMX)，该指令集仅包含 57 条多媒体处理指令，标志着 SIMD 技术第一次大规模应用到了消费级计算机设备上。

由于 MMX 并没有为英特尔公司带来预期的 3D 图形计算能力的大幅提升，于是 1999 年英特尔公司在其奔腾 3 处理器架构中推出了全面覆盖多媒体扩展指令集的流式 SIMD 扩展(Streaming SIMD Extensions,SSE)指令集[53,54]，该指令集有效的提升了 CPU 对于 3D 图形、视频和音频的处理能力。之后英特尔进一步完善了 SSE 指令集，陆续推出了一系列的 SSE 系列指令集，包含 SSE、SSE2、SSE3 以及 SSE4，每一代的 SSE 指令集相较于上一代都做了数据操作的部分扩展，但其操作的实质核心都是将数据放入一个 128 位长的寄存器并通过相应的指令实现对该寄存器内数据的并行化操作。

在处理 Block 数据过程中每个数据的长度为 16 位，在原有的串行数据处理过程中每个数据都要经过类似的数据操作即 64 次数据操作，而在 SSE 指令的支持下单次操作的数据量提升到了 128 位，单指令处理的数据量是不使用 SSE 指令的 8 倍即在不考虑其他因素影响的情况下可以实现 8 倍的编码效率提升。考虑到尽可能提升 SIMD 算法的通用性以及部分必要的指令操作的支持，本方案最终选择使用的 SSE 指令集版本为 SSE2。

4.1.3 ARM 架构下的 SIMD 介绍

ARM 架构下同样设计了类似的 ARM NEON 指令集[55,56]，ARM NEON 指令集由 ARMV7 引入并在 ARMV8 上对其指令集功能进行了部分扩展，使其能够完成 X86 下 SSE 可以实现的所有基本操作。

NEON 同样是使用 32 个 128 位长的寄存器来实行数据的并行处理，与 SSE 不同的是 NEON 考虑到了利用汇编指令直接进行优化的难度较大因此设计了 ARM NEON intrinsic 指令集，该指令是对原有汇编指令的封装，通过 intrinsic 指令可以直接通过 gcc 进行编译不需要考虑实际底层的寄存器分配情况，大大的降低了指令优化的难度。

4.2 SIMD 熵编码方案介绍

4.2.1 标量数据的熵编码过程分析

为了分析对熵编码进行并行化优化的主要难点，本小节简要分析了标量数据的熵编码流程。单个 Block 数据块中的熵编码流程如下图 4.2 所示，每一个标量系数都需要经过 Zig-Zag 序列化、RLE 编码、VLI 编码以及哈夫曼编码四个编码环节，最终生成了压缩文件中的二级制码流。

通过 Zig-Zag 序列化后，Block 数据块中的每一个标量数据都会按照特定的顺序排列，排列后的第一个系数被称为 DC 系数。由于 DC 保存了图像主要的直流信息，因此 DC 系数数值普遍较大，为了方便对其进行编码当前 Block 中的 DC 系数会与上一个 Block 中的 DC 系数进行差分运算在进行后续编码。序列中从第 2 个到第 64 个系数为对应频率的 AC 系数，差

分后的 DC 系数与 AC 系数会逐个进行 RLE 编码，VLI 编码以及哈夫曼编码。

RLE 编码会分析当前系数是否为零，对于非零的系数，RLE 编码会结合已统计的零系数的个数进行一次编码，而对于零系数，RLE 会对连续的零系数个数进行计数操作，直到出现非零系数或当前计数值大于 15 时，编码会进行一次写入操作。

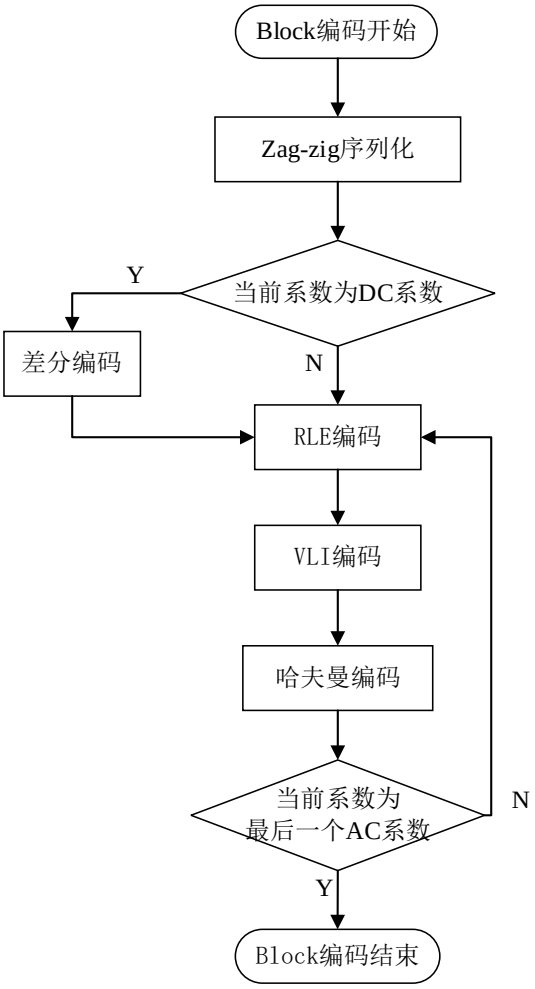


图 4.2 Block 数据熵编码流程图

VLI 编码会将 RLE 编码完成后的非零系数进一步压缩，VLI 编码首先会根据系数是否大于零进行分类讨论。对于正数(大于零)的情况 VLI 编码对标量去除前导的无效零位并记录编码后的实际编码长度，将编码长度与 RLE 统计的零系数个数合并为后一个字节(高四位存储前面零系数个数，第四位存储实际的编码长度)；对于负数(小于零)的情况，由于负数在计算机中通过补码进行存储，因此 VLI 编码的处理流程稍加复杂，在正数处理流程的基础上需要先对负数取绝对值，并将绝对值的反码作为 VLI 编码的最终结果写入。

哈夫曼编码是熵编码的最后一步，负责对 VLI 编码得到的实际编码结果部分进行进一步的压缩。最终的二进制码流中包含了 VLI 编码结果中的一个字节前导信息以及实际编码结果经过哈夫曼编码再压缩得到的结果。

4.2.2 SIMD 并行化难点分析

在寄存器的位长为 128 位与单个标量系数的位长为 16 的情况下，采用 SIMD 进行熵编码

一次并行处理 8 个标量数据。由上一节的标量系数处理流程可知，标量系数在编码过程前首先要根据 Zag-Zig 规则进行重新排列，在 RLE 编码过程中需要区分当前系数是否为零，若当前数据为零则需要统计累计的零系数个数；在 VLI 编码过程中需要根据系数是正整数还是负整数的情况进行分类处理。

SIMD 指令每次操作的都是整个 128 位的寄存器，因此根据具体情况进行分类操作的方法在 SIMD 下是难以实现的。要使用 SIMD 实现编解码就意味着需要用一个统一的操作方式处理 Block 中的数据，同时也要保证 SIMD 编码后的结果与单独的标量系数编码结果一致。如何设计并行化处理的流程以及使用指令集来实现这一流程就成为了实现数据并行化编码的难点。

SIMD 的统一的编码过程除了要要保证对不同类型系数均能正确编码外，还必须解决 RLE 编码过程中的零系数个数统计问题。在单个标量数据的处理流程中，连续零系数的统计采取逐个统计的方式，而在多个数据并行编码过程中，每个指令处理并行处理多个标量数据无法采取逐个统计的方式完成零系数个数的统计。因此在设计 SIMD 并行编码方案之外，额外加入一种可以实现快速统计零系数的机制，用于后续的编码写入过程使用。

4.2.3 SIMD 熵编码方案介绍

针对上一小节提到的 SIMD 并行化无法在编码过程中完成零系数的统计问题，本方案加入了 Bitmap 的概念。通过 Bitmap 可以将原有 64 * 16 位的 Block 数据块中的数据根据对应位置是否为零映射到一个 64 位无符号整型变量 Bitmap 上。通过移位分析该变量的数值即可实现对当前 Block 中零系数的快速定位与数量统计。

为了并行化生成 Bitmap，首先要将 Zag-zig 序列中标量系数按照八个标量系数一组分别放入 8 个 128 位的寄存器中，然后对 8 个寄存器中的非零系数信息进行提取与合并，最终实现 bitmap 的生成。具体流程如下图 4.3 所示，每一个 128 位 Row 寄存器保存有 8 个标量系数，通过指令首先将每个寄存器中非零系数位置用 0xFFFF 进行标记，然后将两个寄存器中的位置信息进行合并，实现用一个寄存器表示 16 个标量系数，同理类推最终实现了将原有的 64 个 16 位的标量数据映射到一个 64 位的无符号整形数据 Bitmap 中。

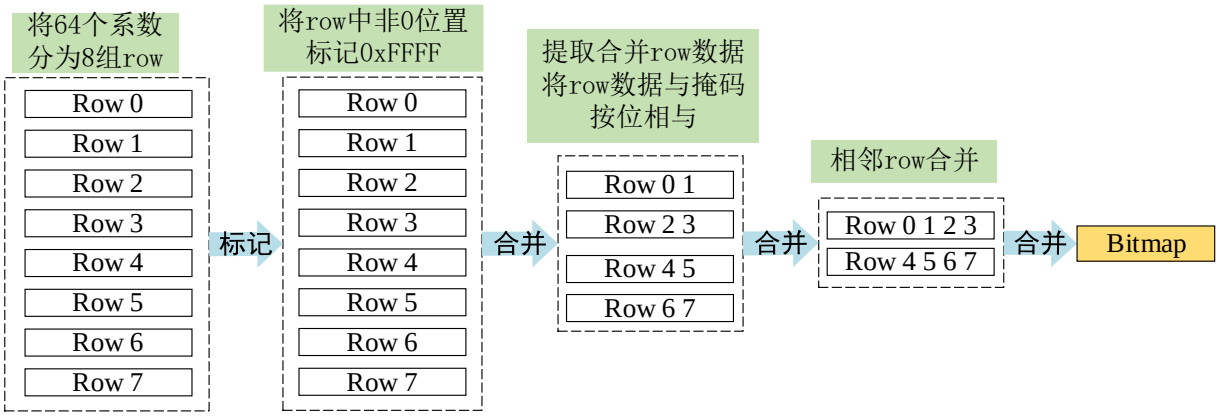


图 4.3 Bitmap 生成

为了能够在 VLI 编码过程中实现对负数与正数系数的统一操作，本方案对每一个负数系数生成了一个对应的掩码(负数有效数据长度内均为 1 的二进制值)与绝对值，通过掩码与绝对值的异或操作使得负数系数变换为对应的绝对值的反码，处理后的正数系数不发生变化这样做就可以实现通过统一流程同时处理正数与负数系数，同时保证处理后的结果符合预期编码结果。

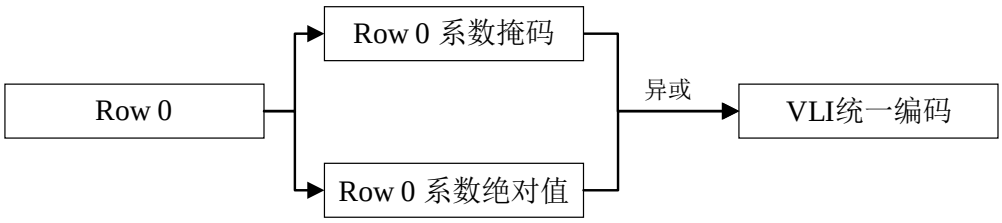


图 4.4 VLI 编码

如上图 4.4 所示，对于并行操作的寄存器 Row0 首先要生成一个正数位置为 0 负数位置为对应掩码的掩码矩阵，然后在对 Row 系数分别取绝对值生成一个绝对值矩阵，最后通过掩码矩阵与绝对值矩阵的异或操作实现对正、负系数的统一编码工作。绝对值异或掩码的结果即为绝对值的反码也就是负数的编码结果，而对于正数而言无论是取绝对值的操作还是异或上 0 都不会影响编码的结果。

综上所述为了实现 SIMD 下的并行编码，在原有的编码任务外要额外添加 Bitmap 的计算以及对正负数的统一处理。SIMD 的并行编码总的流程图如下图 4.5 所示，编码过程中会首先根据 Block 中的数据生成对应的 Bitmap,然后通过数据并行化对所有系数执行统一的编码操作(在 4.3 节将具体介绍)，编码完成后具体的编码结果将报错到 diff 数组中。最后通过分析 Bitmap 可以实现对零系数的快速统计与非零系数的快速定位，通过将 diff 数组中对应的非零系数编码结果写入完成整个并行编码的流程。

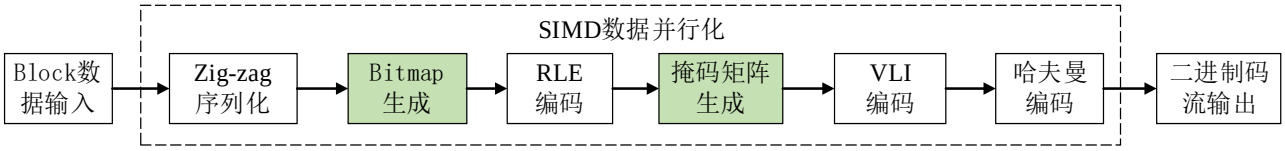


图 4.5 SIMD 并行编码

4.3 SIMD 并行化对比实验

4.3.1 X86 平台并行化加速实验

X86 下的 SIMD 并行化基于 SSE2 汇编指令实现，为了实现对汇编指令的顺利编译以及编译后与二次压缩的 C++代码部分顺利链接，本文使用了 NASM 对 SIMD 的汇编文件进行了单独的编译生成相应的.o 文件并在最终动态库的链接过程中对.o 文件进行相应的链接，实现了 C++与汇编代码的联合编译生成动态库。X86 平台 SIMD 动态库的编译过程如图 4.6 所示，为了实现 X86 架构下的 SIMD 指令调用，本文分别将核心的 C++编码算法与使用 SSE2 指令

集编写的汇编代码使用对应的编译器进行编译，然后通过链接操作将两部分的二进制代码进行链接并生成相应的动态库，X86 架构下的程序可以通过该动态库方便的对核心编解码算法进行调用。

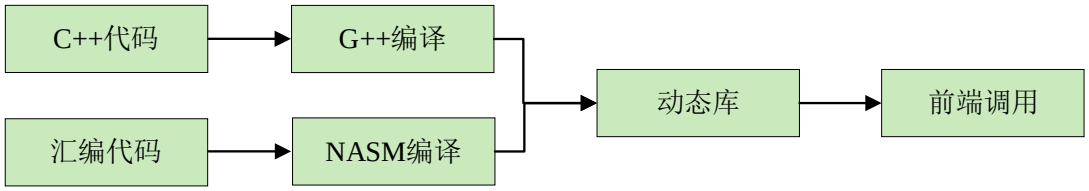


图 4.6 X86 平台 SIMD 库的生成

为了测试 SIMD 并行化熵编码在 X86 架构下的提速效果，本实验选择了四类主流分辨率图片各 1000 张作为测试图片集，通过比较 Brunsli 项目中的熵编码时间与 SIMD 并行化后的熵编码的编码时间,验证 SIMD 并行化对于熵编码的速度提升幅度。实验结果如下表 4.1 所示:

表 4.1 X86 架构 SIMD 平均并行度

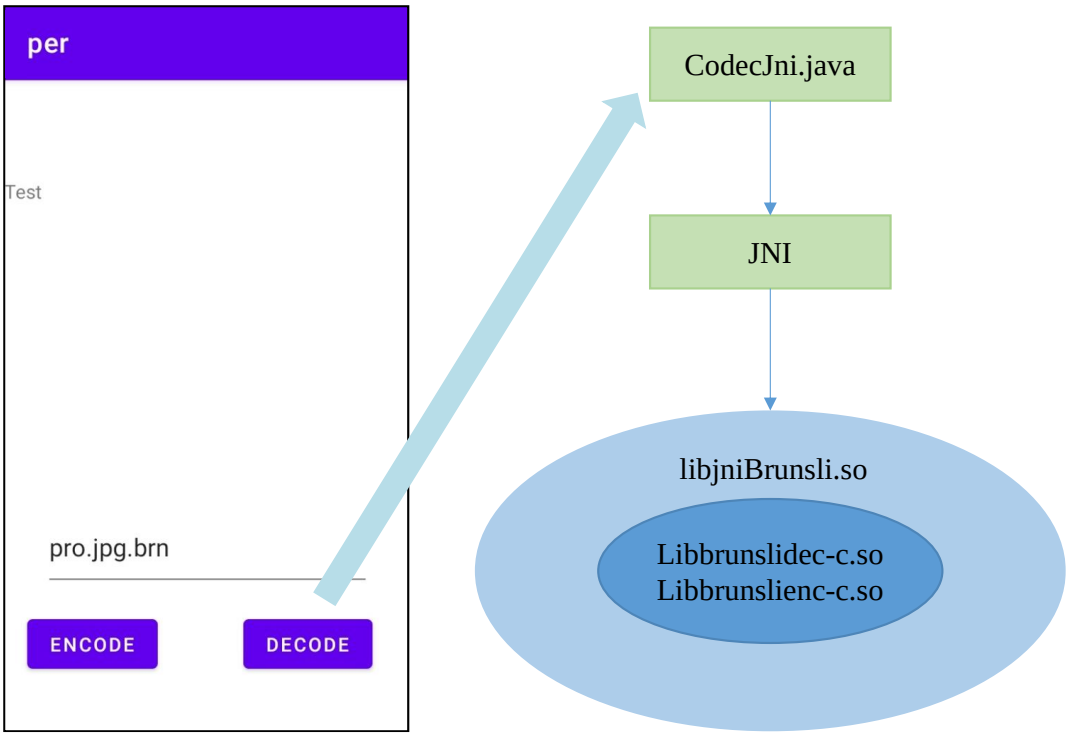
分辨率	数量	Brunsli 平均编码时间(ms)	SIMD 并行化平均编码时间(ms)	平均平行度(%)
2976*3968	1000	115.5	53.79	45.7
3024*4032	1000	114.11	50.44	44.2
3000*4000	1000	125.05	56.27	44.99
3072*4096	1000	112.07	49.61	44.26

通过对上表 4.1 中的实验结果分析可得，X86 平台下的 SIMD 平均并行度为 45%左右，即经过 SIMD 并行化优化后可以降低 55%的熵编码时间。经过测试对于颜色丰富的场景图片解码时间在 140ms 左右，而在经过 SIMD 数据并行化优化后的编码时间在 60ms 左右，SIMD 数据并行化方案有效的降低了 brn 文件解码的时间成本。

4.3.2 ARM 平台并行化加速实验

ARM 架构是一种基于 RISC（精简指令集计算机）设计的处理器架构，凭借其其低功耗、高性能、高集成度等特点使得 ARM 架构成为移动设备处理器的首选。本文对 ARM 架构的 CPU 同样进行了算法的数据并行化优化。本试验选择了麒麟 960 作为测试的 CPU 平台，该 CPU 内部包含两个 ARM Cortex-A73 大核以及两个 ARM Cortex-A53 小核均支持 NEON 指令集进行并行化优化。

如图 4.7 所示，为了在安卓平台下进行图片编解码测试，本文设计了一个安卓端测试软件进行二次压缩的测试。为了为核心编解码算法的移植提供便利，本文选择了通过将核心的编解码算法打包分别生成一个编码库与解码库，在安卓端仅需通过 JNI 就可以实现实现 JAVA 代码对于动态码中的编解码函数的调用。如下图 4.7 所示，Libbrunslidec-c.so 与 Libbrunslienc-c.so 分别为编解码的 C++核心代码库，为了通过 JNI 实现 JAVA 代码对于 C++库函数的调用需要严格按照 JNI 的调用标准对 C++库进行二次封装打包生成 libjniBrunsli.so 库，该库函数可以通过 JNI 实现安卓软件对于编码码算法的快速调用。



（a）安卓端测试界面　　（b）后端编解码库调用流程

图 4.7 安卓端测试软件与后端库的调用

为了测试在 ARM 架构下 SIMD 数据并行化的优化结果，本试验选择了四类主流手机分辨率图片各 1000 张作为测试图片集，通过比较 Brunsli 项目中的熵编码时间与 SIMD 并行化后的熵编码的编码时间,验证数据并行化在 ARM 架构下对于熵编码的性能提升幅度。实验的结果如下表 4.2 所示：

表 4.2 ARM 架构下 SIMD 并行化

分辨率	数量	Brunsli 平均编码时间(ms)	SIMD 并行化平均编码时间(ms)	平均平行度(%)
2976*3968	1000	287	180	62.7
3024*4032	1000	357	217	60.7
3000*4000	1000	453	270	59.6
3072*4096	1000	358	218	60.89

通过上表中的实验结果分析可得，ARM 架构下的 SIMD 平均并行化度为 60%左右即通过 NEON intrinsic 指令集进行并行化优化后可以降低 40%左右的熵编码时间。相较于 X86 下 45%左右的平均并行度，本方案基于 NEON intrinsic 指令集优化的提升幅度较低，主要的原因是 NEON intrinsic 并不是直接通过汇编指令进行编写，在转化成汇编指令的过程中存在 10%左右的性能损失，因此实验的结果符合对并行化速度提升的理论预期。

4.3.3 多线程 SIMD 联合并行化实验

SIMD 并行化加速是通过对 Block 数据块进行数据并行处理实现对熵编码的并行提速，

上一章中的多线程并行化主要是通过将图片分割为多个子图，然后通过多核处理器对子图进行并行编码的方式实现。因此两种并行化方案不存在优化冲突，可以进行联合并行化。

为了能够测试多线程并行化与 SIMD 并行化的联合提速效果，本实验选择了四类主流手机分辨率图片各 1000 张作为测试图片集，在双线程下使用 SIMD 并行化进行联合并行化优化，通过对比 Brunsli 项目的平均编码时间，验证双线程 SIMD 并行化对于熵编码的速度提升幅度。实验结果如表 4.3 所示：

表 4.3 多线程 SIMD 联合并行化

分辨率	数量	Brunsli 平均编码时间(ms)	双线程 SIMD 并行化平均编码时间(ms)	平均平行度(%)
2976*3968	1000	115.5	35.34	30.59
3024*4032	1000	114.11	35.51	31.11
3000*4000	1000	125.05	37.65	30.1
3072*4096	1000	112.07	34.9	31.14

通过上表 4.3 中的实验结果分析可得，双线程 SIMD 模式下的平均并行化度为 30%左右即通过 NEON intrinsic 指令集进行并行化优化后可以降低 70%左右的熵编码时间。结合前面对于双线程并行化优化的平均并行度为 60%左右，X86 架构下 SIMD 的平均并行度为 45%左右，因此本实验的联合优化结果符合对并行化速度提升的理论预期。

4.3.4 SIMD 二次压缩前后图像数据对比

为了验证 JPEG 图片在多线程二次压缩产生的 brn 文件可以无损的还原回到原始的 JPEG 图片，原图与二次压缩后还原图片的对比如下图 4.8 所示，通过对比可以发现二次压缩前后图片没有明显的图像质量差比。

原始图片

二次压缩后还原图片

图 4.8 二次压缩前后图像质量对比

为了进一步说明本文的二次压缩方案为无损压缩，本文使用了 beyond compare 软件对原始 JPEG 图片与二次压缩解码后的 JPEG 图片进行了十六进制的码流对比，图片码流的对比结果如图 4.9 所示，通过十六进制码流数据对比显示二次压缩前后的图像质量完全相同，达到了本文方案所说的无损还原的指标。

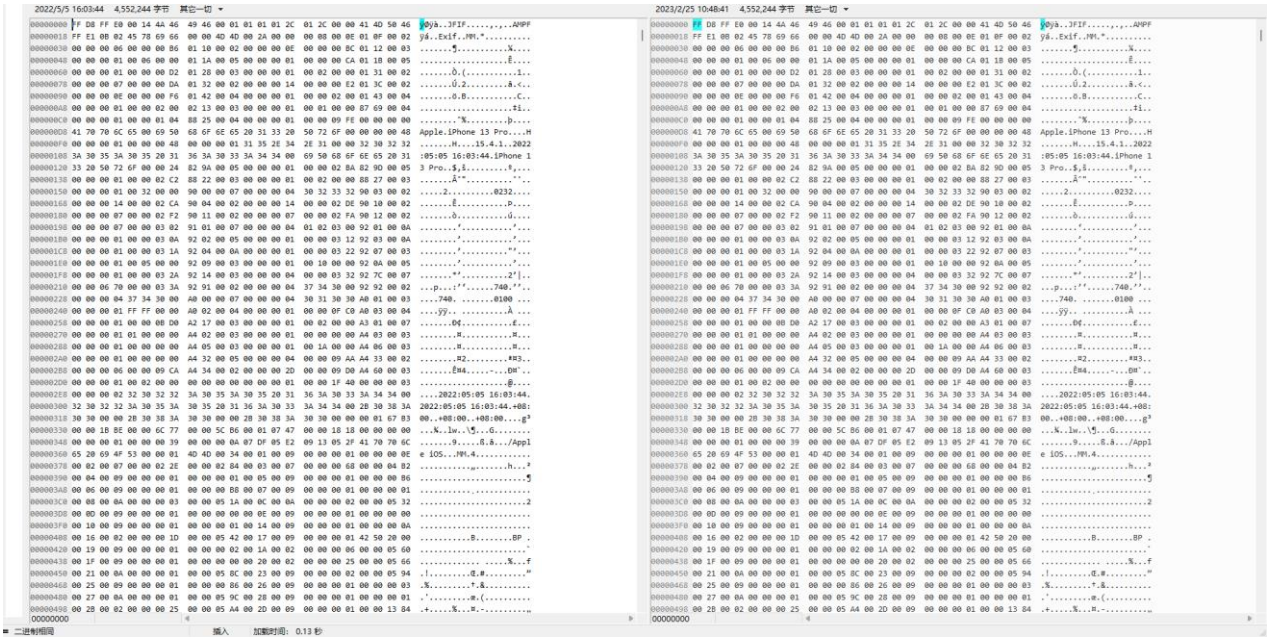


图 4.9 二次压缩前后图像码流对比

4.4 本章小结

本章首先介绍了 SIMD 数据并行化的实现原理，然后介绍了 Block 数据的串行编码流程，并通过对数据并行化流程的分析得出了实现该方案的主要难点。为了解决这些难点，本章的方案加入了 Bitmap 与系数掩码矩阵，实现了并行编码过程中的零系数统计与不同类型数据的统一编码处理。最后，本章在 ARM 架构与 X86 架构下，分别对数据并行化方案进行了实验，实验的结果说明不同架构下熵编码的时间分别减少 55%与 40%，数据并行化方案有效的提升了熵编码的效率。此外在双线程与 SIMD 联合并行化实验中，并行化方案的熵编码时间相较于原始方案减少 70%左右，进一步提升的熵编码的速度。

第 5 章 基于超标量架构的指令并行化优化

超标量架构是现代 CPU 为实现指令并行化而广泛采用的微体系结构，为了实现对熵编码的指令并行化加速，本章首先介绍了超标量架构下指令并行化的实现原理，分析原有熵编码算法难以实现指令并行化的原因。然后，基于上述原因，提出了通过软件优化实现指令并行化熵编码的方案。最后，通过实验结果说明了该指令并行化方案可以有效提升熵编码的效率。

5.1 超标量架构下指令并行化原理概述

超标量架构[57,59]能够实现在 CPU 主频不变的情况下，通过多条指令并行处理实现更高的 CPU 吞吐率。本章节简要的分析了超标量架构的实现原理，然后对当前指令并行化存在的问题进行了分析，最后介绍了如何通过针对超标量架构的程序优化提升指令并行化的概率。

5.1.1 超标量硬件架构概述

在传统的五级流水线处理器中，主要包含取址、译码、执行、访存、写回五个单元，在标量流水线中每个时钟周期内都会有一条指令从存储单元中取出进入流水线，理论上每隔一个时钟周期就会有一条指令被处理完毕，但在实际指令运行过程中由于指令跳转、冲突等问题，实际的指令平均执行速度要小于一个周期一条指令。为了提高单周期内的指令执行效率引入了超标量架构，如图 5.1 所示，超标量流水线相较于标量流水线加入了发射环节，并且包含了两个以上的发射队列与执行单元，使得多条指令可 以在同一个时钟周期内被不同的执行单元进行处理。

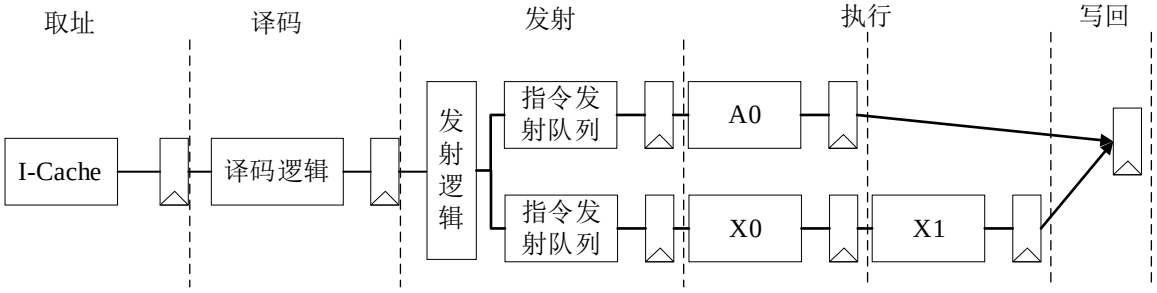


图 5.1 超标量流水线架构

超标量又被称为动态多发射[60,61]，指的是在多级流水线的基础上加入了若干个平行的指令发射队列，发射队列中的指令经过发射逻辑处理后可以将原有的串行指令重新排布将不存在冲突的指令放入同一个指令发射队列，最终实现了在一个周期内并行执行指令的目的。一个简单的 2-Way 超标量乱序处理器的流水线模型如上图 5.1 所示，该流水线内部共分为五段，分别是取指、解码、发射、执行。接下来简单分析下该流水线的工作流程：

首先指令会从指令缓冲区 I-Cache 中被取出交由分支预测单元进行判断当前指令是否会涉及分支，若涉及分支则需要按条件进行地址跳转操作，若当前指令不是分支指令则将当前指令交由解码单元进行解码操作，指令所需要操作的寄存器会在这个环节被记录，之后进入

重命名环节。在重命名环节中会将当前指令需要用到的寄存器进行重新映射，以避免寄存器使用存在冲突，重新映射后的指令会进入发射单元。在发射单元中会进行乱序操作，根据乱序发射的逻辑选择发射队列中最适合当前发射的指令发射到执行单元，若发射队列中存在两条不相关的指令可以并行执行，则此时发射单元会进行多发射（超标量）操作，两个执行单元会并行的执行两条指令，实现指令并行的目的。如下图 5.2 所示，在理想情况下通过两个执行单元可以实现两个相邻指令的并行执行。

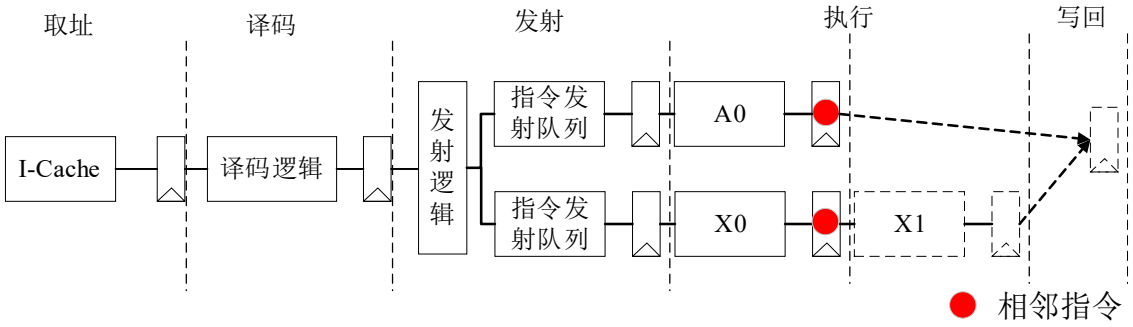


图 5.2 理想情况下的指令并行

综上所述，多发射架构下阻碍指令并行化的主要问题是串行指令间存在着指令冲突的问题，因此在多发射架构下又通过了乱序执行单元的引入改变了原有的串行指令的排布顺序，从而减少了指令冲突发生的概率，接下来的小节本文将着重分析指令冲突发生的具体情况以及乱序执行单元起到的关键作用。

5.1.2 指令冒险与乱序执行

现代 CPU 均采用多级流水线[62]结构，即每条指令的取址、译码、执行等操作分别交由单独的单元执行，这样设计的优点是提升了硬件的综合利用率，使得 CPU 整体运行速度得到明显提升。在单个时钟内完成一条指令的流水线称为标量流水线，标量流水线在理想情况下可以实现各个指令可以实现没零等待的重叠操作，达到每个时钟流出一条指令的效率。然而在实际执行过程中，如下图 5.3 所示，由于指令二操作的寄存器结果必须等待指令一处理完毕写回后才能进入执行单元，因此指令必须等待指令完成写回操作。除了下图中的情况外标量流水线还会因为多种原因出现频繁出现空指令(NOP)等待的情况，这种通过空指令浪费 CPU 计算资源的情况被称为指令冒险。

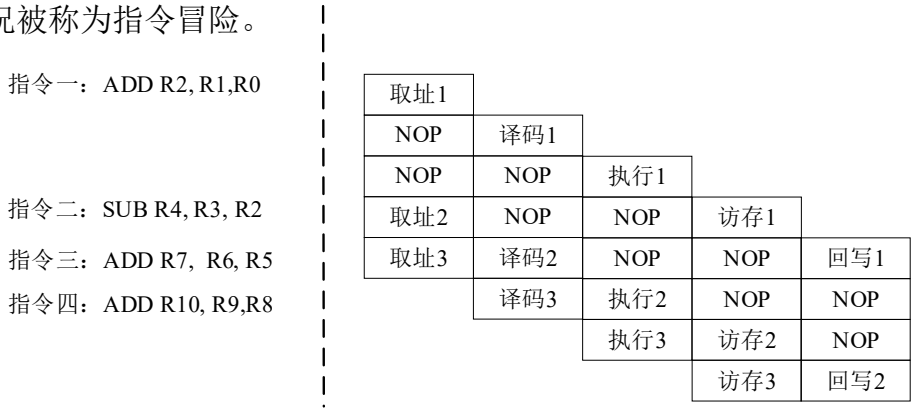


图 5.3 指令冒险

造成指令冒险的核心原因是由于两条连续的指令在顺序执行时存在一定的“冲突”或“依赖”关系，导致指令 A 在多级流水线中运行完成前指令 B 无法进行取址、译码等预处理操作，根据造成冒险的原因又可以将指令冒险细分为结构冒险、数据冒险与控制冒险三种情况，如下表 5.1 中伪代码所示：

表 5.1 指令冒险伪代码

结构冒险	数据冒险	控制冒险
		1 BEQ R1，R2, Func1
1 ADD R2, R1, R0	1 ADD R2, R1, R0	2 ADD R2, R1, R0
2 SUB R1, R4, R3	2 SUB R4, R2, R3	3 ….
		4 Func1:
		5 SUB R2, R1, R0

在结构冒险中，连续的两条指令都要处理相同硬件资源。如上表 5.1 结构冒险伪代码中的 R1 寄存器，本质上这两条指令间不存在“依赖”关系，而仅仅是由于使用的硬件资源发生了“冲突”导致流水线产生停顿。为了解决这类冲突，可以通过对产生冲突的寄存器进行寄存器重命名操作来实现。目前寄存器重命名可以通过寄存器静态编译实现也可以有 CPU 内部的寄存器重命名单元动态实现。

在数据冒险中，连续的两条指令间在运算结构上存在着一定的依赖关系。如上表 5.1 中数据冒险伪代码所示，后一条 SUB 指令操作的 R2 寄存器必须由前一条 ADD 指令执行完毕写回操作完成后才可以进行操作，因此会产生如图 5.4 所示的流水线停顿暂缓 SUB 指令的执行。

在控制冒险中，当前一条指令执行条件判断指令时，由于无法判断后续指令执行的指令顺序,因此无法实现对后续指令直接进行取址与译码操作。如上表中控制冒险伪代码所示，由于无法确定 R1 与 R0 寄存器内容是否相等因此无法确定下一条指令是否会发生跳转，此时 CPU 会进行如数据冒险一样的流水线停顿，等待跳转位置确定后才会对后续指令进行操作。

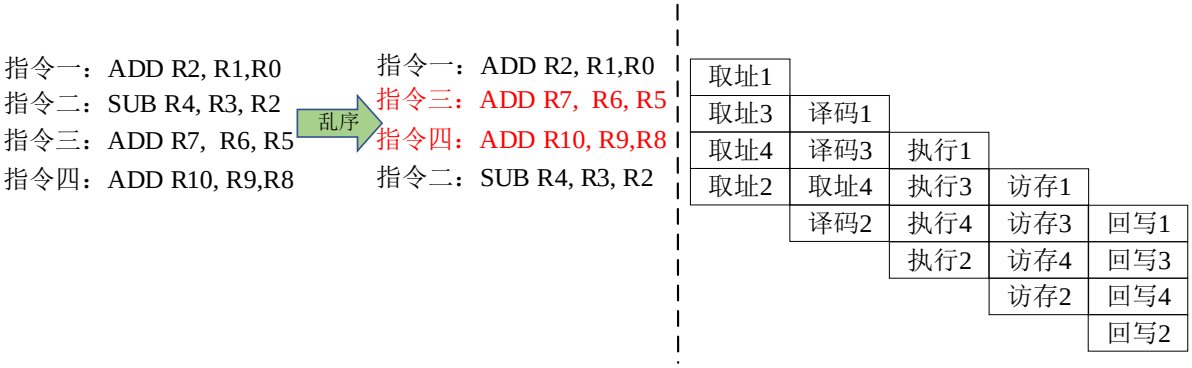


图 5.4 指令乱序操作

为了尽可能地减少指令冒险情况的发生，现代 CPU 架构引入了乱序执行（Out-of-Order Exection，OOOE）单元[63-65]。乱序执行单元在 CPU 在处理指令流前会对原有的顺序指令流进行分析，然后通过改变指令流中部分指令的运行顺序，有效降低指令冒险出现的概率。如图 5.3 所示，在顺序执行过程中由于指令一与指令二均需要操作寄存器 R2，因而指令二必须等

待指令一执行完毕将 R2 寄存器资源释放才可以进行取址、译码等操作。在乱序执行单元引入后，指令执行的顺序被重新排布，在指令一与指令二之间插入了指令三与指令四，有效的消除了指令一与指令二之间因为指令冒险而产生的空指令（NOP），提升了系统整体的硬件资源利用率。

5.1.3 超标量架构下的程序优化

尽管目前大多数的 CPU 内部均采用了超标量流水线的架构，但实现所有指令的并行化依旧困难重重，主要的原因在于目前大多数的高级编程语言在设计之初就是围绕面向过程进行设计，这样做的好处是代码逻辑清晰可读性强，更符合人类的传统思维模式。但这也带来了一些缺点，由于侧重于对单个对象的过程设计使得经过编译得到的汇编指令往往前后关联性很强，相邻指令往往操作的都是相关的硬件资源，这就会极大的增加指令冒险的概率使得指令并行化无法进行。因此尽管硬件上支持了指令并行化没有软件程序上的配合就难以实现高性能的指令并行化操作。

如何优化代码实现串行指令的并行化是程序优化的重点。尽管在乱序执行过程中会从指令队列中尽可能的选取可以实现多发射的并行指令，但乱序执行队列的深度相对有限往往只能缓存几句高级语言对应的汇编指令，因此如果高级语言间前后的关联度过高依旧无法通过乱序执行来调整实现指令并行化。下面通过一个代码举例说明如何通过优化代码结构后提升指令并行化的几率。

表 5.2 指令并行化软件优化

未进行优化		指令并行化优化	
1	double func1() {	1	double func2() {
2	double tmp1=0;	2	double tmp[4]={0};
3	for (int j = 0; j < 10000000; j+=4) {	3	for (int j = 0; j < 10000000; j+=4) {
4	tmp1 += j * j;	4	tmp[0] += j * j;
5	tmp1 += (j+1) * (j+1);	5	tmp1 += (j+1) * (j+1);
6	tmp1 += (j+2) * (j+2);	6	tmp[2] += (j+2) * (j+2);
7	tmp1 += (j+3) * (j+3);	7	tmp[3] += (j+3) * (j+3);
8	}	8	}
9	return tmp1;	9	return tmp[0]+tmp[1]+tmp[2]+tmp[3];
10	}	10	}

如上述伪代码所示，func1 与 func2 均是计算 0 到 10000000 之间整数的平方和，从时间复杂度的角度分析，两个函数的时间复杂度均为 O(n)，所以理论上两个函数的运行时间应该也是基本相同的。但在实际的实验过程中，通过测量 func1 与 func2 执行 100 次所需的时间，我们发现 func2 的运行时间仅为 func1 运行时间的 40%左右。从高级语言的角度难以解释造成这一现象的原因，本文选择通过对汇编指令的分析进一步的解释造成这一实验结果的原

因,Func1 与 Func2 对应的汇编伪代码如下表 5.3 所示：

通过对比 func1 与 func2 汇编代码，发现 func2 相较于 func1 最大的区别在于 func1 的循环体内部语句在完成平方操作的乘法运算后，将计算的结果始终加入了同一个地址-8(%rbp),而在 func2 中每一步的计算都是通过不同的地址进行存储，计算结果分别存储在-48(%rbp)、-40(%rbp)、-32(%rbp)、-24(%rbp)对应的内存区域。由于 func1 中始终操作同一内存地址因此指令集间存在着指令冒险无法实现并行执行，而在 func2 中由于对计算的结果进行了单独的存储，因此不会形成指令冒险指令并行化的概率有明显的提升。

表 5.3 指令并行化汇编代码

Func1 循环体内部汇编代码			Func2 循环体内部汇编代码		
1	movsd	-8(%rbp), %xmm1	1	movsd	-48(%rbp), %xmm1
2	addsd	%xmm1, %xmm0	2	addsd	%xmm1, %xmm0
3	movsd	%xmm0, -8(%rbp)	3	movsd	%xmm0, -8(%rbp)
4	…		4	…	
5	movsd	-8(%rbp), %xmm1	5	movsd	-40(%rbp), %xmm1
6	addsd	%xmm1, %xmm0	6	addsd	%xmm1, %xmm0
7	movsd	%xmm0, -8(%rbp)	7	movsd	%xmm0, -8(%rbp)
8	…		8	…	

5.2 指令并行化编码难点分析

5.2.1 JPEG 编解码指令冒险情况分析

指令并行化在软件层面优化的主要方向是通过减少指令冒险来增加指令并行化发生的概率，因此本文对 JPEG 原有算法的指令冒险情况进行了必要的分析。在 JPEG 的熵编码过程中，编码的总体流程是：

（1） 将图像数据分割为若干的 Block 数据块，逐个操作 Block 数据块实现整体的编码

（2） 对于 Block 数据块内部的数据采取逐个循环操作的方式，每个数据都需要经过 Zig-Zag 序列化、RLE 编码、VLI 编码以及哈夫曼编码操作。

从上面的总体编码流程可以看出编码的核心过程在于对 Block 内部数据的编码，因此分析 Blcok 内部编码的指令冒险情况是非常有必要的。在传统的 JPEG 编码过程中对于单个 Block 采取的是循环编码方式，即按照 Zig-Zag 调整后的数据顺序将每个系数进行逐个编码，当一个系数被编码完成后才会对下一个系数进行操作。由于每次编码过程仅针对一个系数，因此编码过程中产生的指令必然操作着相同的内存区域，最终造成编码指令间的紧耦合，语句间的关联度极高，导致底层汇编指令在实际运行过程中无法实现指令并行化如图 5.5 所示，由于两条指令操作同一对象的地址，因此即便在超标量流水线架构中包含量套执行单元也无法实现指令并行化。

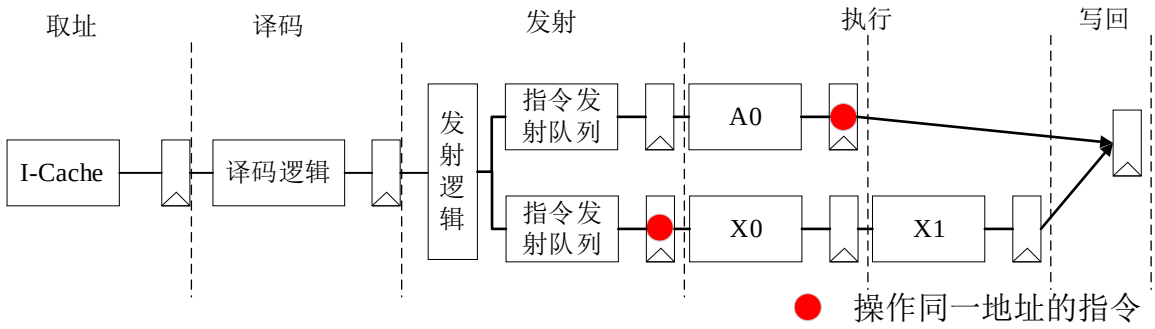


图 5.5 数据冒险情况下的指令运行

除了上面提到的由于操作对象单一的原因外，造成指令冒险的另一个原因是由于分支语句的频繁出现，在编码过程中对于零系数与非零系数中的正整数与负整数的操作流程均不相同，因此每个系数都需要进行若干次的分支语句才能够得到正确的编码结果，这也就造成指令跳转的频繁发生。

综上所述，在熵编码过程中存在频繁的指令冒险现象。因此如果能实现熵编码的编码流程的指令并行化优化，理论上可以极大的提升编码的效率。

5.2.2 JPEG 指令并行化难点分析

指令并行化借助的是现代 CPU 内部的多发射架构，软件层面对于指令并行化[66]的优化点在于减少相邻语句间的相关度以减少指令冒险产生的可能。在传统面向过程的编程的过程中，处理流程往往针对的是某一特定的内存区域，因此语句间关联性极强也造成了底层指令的指令冒险现象频繁发生。

由上一小节的分析，可知在 JPEG 中的熵编码过程中采取的是对 Block 中的系数进行逐个编码的方式，单个系数的编码过程中存在着指令冒险的问题，因此改进编码的逻辑顺序，使得编码过程中可以实现多系数并行编码是实现指令并行化的一个难点。除了处理流程中的数据过于单一问题外，实现指令并行化的另一个问题是指令跳转，在一旦语句发生跳转则意味着并行指令预先取值译码的操作均失效，因此实现指令并行化的另一个问题是如何尽可能的减少分支的产生。

在原始的 JPEG 编码中对于系数的编码与写入操作都是顺序执行的，连续操作使得生成的汇编指令间存在着严格的数据依赖关系，因此在原始 JPEG 中即便硬件支持多发射也难以实现指令间的并行化。如何减少指令间存在的数据冒险成为实现指令并行化的核心问题。除了数据冒险外在原有的 JPEG 编码中存在大量的分支判断，这些分支判断指令一旦发生跳转就会造成多级流水线下的预操作指令全部失效，因此尽可能地减少分支语句的产生也是实现指令并行化所要解决的问题之一。

为了消除并行化过程中的处理对象过于单一的现象，可以通过采取多系数并行操作的方式即每次操作多个 Block 中的系数。但这样的方式又会带来一连串的问题，由于不同的系数采取的编码流程可能会不同因此，不同系数的并行编码过程难以保证能实现全过程的并行，

其次在原有的编码过程中包含了对零系数的统计（RLE 编码的必须的数据），因此并行操作过程中如何保证对于单个系数的零系数正确统计又是另一个问题。

5.3 JPEG 指令并行化方案的设计与实现

5.3.1 指令并行编码方案设计

由于单 Block 编码过程中内部系数逐个编码会造成底层指令间的紧耦合关系，最终导致指令并行化的概率降低，因此本文尝试通过对多个 Block 进行来实现指令并行编码，消除单系数编码过程中的指令冲突问题。

在多个 Block 并行编码的方案设计过程中，同样存在着与多线程并行化方案相同的非字节对齐数据的写入以及 SIMD 并行编码过程中对于零系数个数的统计问题。因此对于数据并行化方案的设计借鉴了前面两章并行化方案的解决方案。具体的指令并行化熵编码方案框图如下图 5.6 所示：

在指令并行化方案中，多个 Block 的选择方式参考了多线程方案中的子图分割方式，即每次编码不同子图对应位置的 Block 块。因此与多线程方案一样，在编码前需要具体的编码起始位置信息与编码完成后写入地址信息。为了获取这些编码信息，本方案同样的设计了数据并行化的关键字，在二次压缩的编码过程中提取相关信息并写入到 brn 文件中，在 brn 解码过程中则可以根据这些信息实现解码数据写入位置的快速定位。

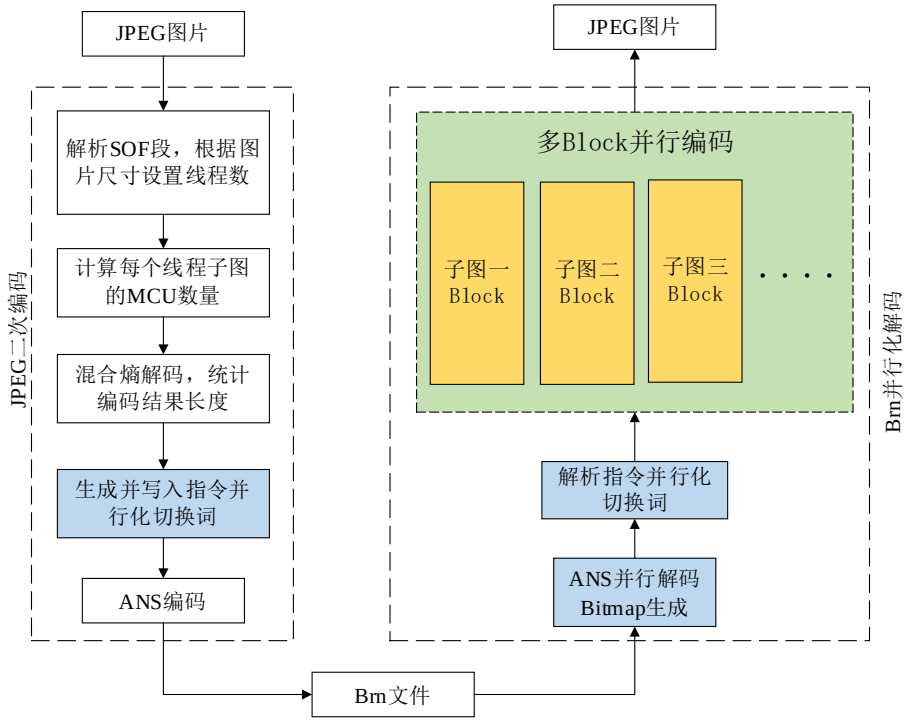


图 5.6 指令并行化编解码框图

在指令并行化的方案中除了要解决 Blcok 的选择外还要考虑单个 Block 内部如何实现系数间的并行化问题。在对单个 Block 编码过程中，RLE 编码会统计非零系数前零系数的个数，但是当多个 Block 并行编码时，每个 Block 内部非零系数分布情况不同，因此无法统一完成

零系数的统计与非零系数的编码问题。

5.3.2 Block 并行编码内部实现

将图片分割为若干子图后，将会对多个子图内的对应 Block 进行并行编码。 如下图 5.7 所示，并行编码过程中将会对两个子图对应位置的 Block 内部系数进行并行编码，通过对两个Block内部系数的交替处理使得相邻指令间的耦合度降低，增大了指令并行化发生的概率。

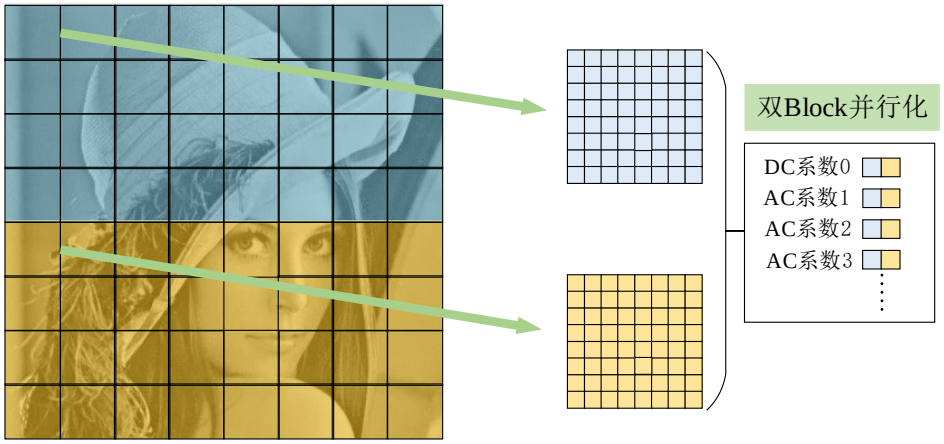


图 5.7 Block 内部数据并行化流程

通过多个 Block 数据块中对应系数位置系数的交替编码，编译后产生相邻汇编指令间实现了对于操作同一数据地址的解耦，解耦的指令在超标量流水线中的执行顺序如图 5.8 所示，通过解耦指令间的依赖可以有效提升指令并行化发生的概率。

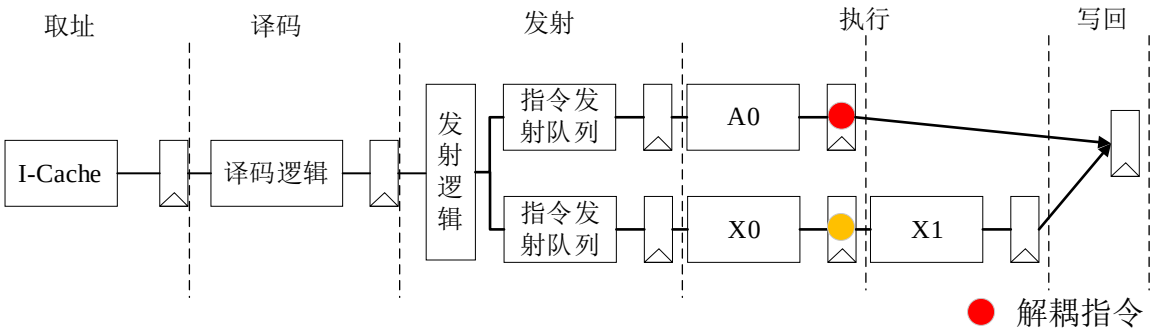


图 5.8 解耦后的指令并行化

多个 Block 并行熵编码过程中，对于不同类型系数的编码过程需要执行不同的操作，因此并行编码过程中可能会出现因为不同类型的系数进行并行而无法实行统一操作的问题。比如在 VLI 编码过程中会对正负系数编码流程不同，多个系数并行时难以保证并行系数间均为同类系数。为了统一正负系数编码的操作流程，本文引入了 SIMD 编码过程中的 VLI 统一编码方式，通过使用系数掩码与系数绝对值实现了正负系数的统一处理。

5.4 指令并行化实验对比与结果分析

5.4.1 指令并行化加速实验

为了能够测试不同数量的 Block 并行后对于指令并行化下的速度提升效果，本实验对同一组图片分别进行了不同数量的 Block 并行编码测试，通过测量解码时间的变化比

较不同 Block 数下指令并行化的提升效果。实验结果如下表 5.4 所示：

表 5.4 指令并行化对比实验

并行 Block 数	Brunsli 平均编码时间(ms)	指令并行化平均编码时间(ms)	平均平行度(%)
2	115.5	103.34	89.47
3	114.11	101.67	89.1
4	125.05	112.4	89.88
5	112.07	102.24	91.22

通过上表中的实验结果分析可得，在不同的 Block 数下指令并行化的平均并行度均为 90%左右即通过指令并行化可以实现 10%左右的编码速度提升。实验的结果说明在原有的 Block 顺序编码过程中确实存在着指令冒险的情况，导致了 CPU 无法以最大的吞吐量进行数据的处理。本试验通过指令顺序的调整有效的减少了指令冒险的发生，提升指令并行化的概率。

5.4.2 双线程指令并行联合优化实验

指令并行化与多线程并行化均是通过将图片分割为多个子图，然后对每个子图进行单独编码实现并行化。因此可将两种优化方案进行结合，在多线程的基础上，对每个线程负责的子图区域是由指令并行化进行二次优化。

表 5.5 双线程指令并行化对比实验

分辨率	数量	Brunsli 平均编码时间(ms)	双线程指令并行化平均编码时间(ms)	平均平行度(%)
2976*3968	1000	115.5	63.08	54.62
3024*4032	1000	114.11	64.2	56.2
3000*4000	1000	125.05	69.75	55.78
3072*4096	1000	112.07	60.19	53.7

实验结果如上表 5.5 所示，为了能够测试多线程并行化与指令并行化的联合提速效果，本实验选择了四类主流手机分辨率图片各 1000 张作为测试图片集，在双线程下使用指令并行化进行联合并行化优化，通过对比 Brunsli 项目的平均编码时间，验证双线程指令并行化对于熵编码的速度提升幅度。

通过熵表 5.5 中的实验结果分析可得，双线程指令并行化模式下的平均并行化度为 55%左右即通过联合进行并行化优化后可以降低 45%左右的熵编码时间。结合前面对于双线程并行化优化的平均并行度为 60%左右，指令的平均并行度为 10%左右，因此本实验的联合优化结果符合对并行化速度提升的理论预期。

5.5 本章小结

本章首先通过对超标量架构、乱序执行、指令冒险概念的介绍，说明了指令并行化实现

的硬件原理。然后本章对指令并行化熵编码方案实现难点进行了必要的分析，基于 Bitmap 与指令并行化切换词的设计，提出了本章的指令并行熵编码化方案。最后，本章指令并行化加速试验的结果显示指令并行化方案可以减少约 10%的熵编码时间，说明通过软件优化实现指令并行化的可行性。此外，在双线程与指令并行化联合优化实验中并行化提速的幅度在 45%左右，进一步提升了熵编码的效率。

第 6 章 总结与展望

6.1 论文工作总结

JPEG图片作为主流的图片编码格式受到目前主流的操作系统以通用浏览器的普遍支持，然而越来越高的图片分辨率以及移动互联网海量的传播需求对图片的压缩比提出了更高的要求，因此实现对JPEG图片的二次压缩的研究具有一定的实际意义。本文以改进型JPEG算法的并行化作为研究对象，首先对当前国内外的JPEG并行化技术进行了详细的调查与整理，最后选择了基于谷歌的开源项目Brunsli作为实验研究的主体，该项目对JPEG图片实现了二次无损压缩，压缩后生成的brn文件大小相较于JPEG减少了约22%。然而该项目由于在解码过程中所需的时间过长难以实现推广与应用。本文为了提升Brunsli的解码速度，提出了多线程并行化、数据并行化与指令并行化三个方向作为解码算法改进的突破点。本文详细介绍并分析了Brunsli的二次编解码流程与传统JPEG的编解码流程，并说明了通过并行化技术提升解码速度的必要性。然后引出了本文对并行化研究与实现的主要内容：

(1) 为了通过多核多线程技术实现对解码过程中熵编码速度的提升，本文详细分析了对串行编码流程进行并行化优化的设计难点，对改造过程中遇到的环节依赖的问题、如何保证不同线程任务量的均衡性以及并行化编码完成后的非字节对齐数据的写入问题进行了详细的说明。然后提出了本文的多线程并行化解决方案，通过在编码过程中对MCU行数的统计与横向分割的设计实现了子线程对图像的均衡分割，通过引入霍夫曼切换词的概念设计了新的数据结构用于在编码过程记录子线程的写入起始位置与非字节偏移量解决了解码写入时遇到的非字节写入问题。最后通过实验说明多线程并行改造后对于二次压缩的压缩率影响在1%左右，在双线程下的解码速度提升了40%左右。最后本文对不同采样格式与编码模式的JPEG图片均进行了测试，实验结果说明多线程并行二次编解码方案对于常见JPEG图片均可以实现有效的解码速度提升。

(2) 针对现代CPU架构支持SIMD技术的特点，本文在X86与ARM结构下对二次压缩项目，分别进行了SIMD数据并行化的改造。对于解码过程中的熵编码环节，使用SIMD指令集进行重新实现。对于单个Block数据块的并行化过程中，由于合并后的数据无法保证数据的类型一致使得无法通过统一的并行化步骤进行编码问题，本文加入了新的数据结构Bitmap用于快速定位当前Block数据块中的零系数的数量与具体位置，引入了正负系数统一处理的步骤使得正负系数经过处理后可以实现统一的编码流程。通过实验结果说明在x86架构下SIMD并行化解码的时间减少了约55%，在ARM架构下解码时间减少了约40%左右。最后实验测试了通过多线程与SIMD技术进行联合并行化优化，结果说明联合并行化的解码时间节约了70%左右。

(3) 针对现代CPU架构支持超标量架构的特点，本文提出了基于软件优化的超标量并行化

优化方案。通过对超标量架构实现原理的分析，说明了处理器硬件实现指令并行化所遇到的指令冒险问题，然后引出了通过软件优化来实现减少指令冒险的并行化方案。通过对熵编环节指令的冒险情况进行分析，提出了多个Block数据并行编码来减少指令冒险的方案。最后通过实验证明了通过指令并行化提升解码速度的可行性，实验结果说明指令并行化方案相较于原有方案在解码时间上减少了10%左右，在双线程与指令并行化联合优化下的解码时间减少了约45%左右。

本文针对二次压缩方案中解码时间过长问题，设计了三种并行化编码方案。针对不同编码环境可以使用不同的优化方案进行组合优化，如针对移动端的嵌入式设备对功耗较为敏感的特点，可以使用指令并行化与SIMD数据并行对解码速度进行优化，而对于云存储服务器端高并发的特点，可以选择通过多线程并行化与SIMD数据并行化联合方案进行优化。

6.2 未来工作展望

本文对二次 JPEG 编解码进行了并行化优化，通过多线程并行化、数据并行化以及指令并行化优化证明了对解码环节进行并行化优化的可行性。但是本研究目前所做的工作仍有很大的局限性，未来对于解码并行化的改进方向可以围绕以下几个方面进行展开：

(1) 通过 GPU 对解码过程进行并行化优化，目前智能手机均配备了 GPU 用于图像的处理，GPU 在并行处理速度上要远高于 CPU。通过 GPU 对图片编解码进行并行化优化已有相当多可以参考的案例，因此在后续的工作中可以通过尝试利用 GPU 的硬件并行处理能力对二次压缩的编解码过程进行优化。

(2) 增加 SIMD 并行化的解码环节，本文目前仅对解码过程中的熵编码环节进行了 SIMD 并行化，而对于同样耗时较长的 ANS 编解码、DCT 变换等环节尚未进行相应的并行化优化。在后续的研究中可以同样的并行化方式对耗时较长的环节进行并行化优化，进一步缩短整体的编解码时间。

(3) 设计针对二次 JPEG 压缩的 FPGA 实现方案或专用集成电路芯片。目前主流的编解码都有相应的专用集成电路芯片，通过专用芯片的使用可以有效的提升编解码的效率并且降低 CPU 的处理负担，促进二次 JPEG 编解码方案的应用与推广。

参 考 文 献

[1] Wallace G K. The JPEG still picture compression standard[J]. IEEE transactions on consumer electronics, 1992, 38(1): xviii-xxxiv.

[2] Taubman D S, Marcellin M W. JPEG2000: Standard for interactive imaging[J]. Proceedings of the IEEE, 2002, 90(8): 1336-1357.

[3] Dufaux F, Sullivan G J, Ebrahimi T. The JPEG XR image coding standard [Standards in a Nutshell][J]. IEEE Signal Processing Magazine, 2009, 26(6): 195-204.

[4] 张雅媛, 孔令罔. 一种基于改进量化表的 JPEG 图像压缩算法[J]. 包装工程, 2016, 37(13): 189-194.

[5] Guo L, Shi X, He D, et al. Practical Learned Lossless JPEG Recompression with Multi-Level Cross-Channel Entropy Model in the DCT Domain[C]//Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2022: 5862-5871.

[6] Duda J. Asymmetric numeral systems[J]. arXiv preprint arXiv:0902.0271, 2009.

[7] https://github.com/google/brunsli

[8] Flynn M J. Very high-speed computing systems[J]. Proceedings of the IEEE, 1966, 54(12): 1901-1909.

[9] Delgado J B, Castro M C. Construction and validation of a subjective scale of stigma and discrimination (SISD) for the gay men and transgender women population in Chile[J]. Sexuality Research and Social Policy, 2014, 11: 187-198.

[10] Aklilu N, Elliott D G, Wickman C A. A tightly coupled hybrid SIMD/SISD system[C]//Engineering Solutions for the Next Millennium. 1999 IEEE Canadian Conference on Electrical and Computer Engineering (Cat. No. 99TH8411). IEEE, 1999, 1: 446-449.

[11] Smart N P, Vercauteren F. Fully homomorphic SIMD operations[J]. Designs, codes and cryptography, 2014, 71: 57-81.

[12] Páll S, Hess B. A flexible algorithm for calculating pair interactions on SIMD architectures[J]. Computer Physics Communications, 2013, 184(12): 2641-2650.

[13] Popov A. An introduction to the MISD technology[C]//Proceedings of the 50th Hawaii International Conference on System Sciences. 2017.

[14] Halaas A, Svingen B, Nedland M, et al. A recursive MISD architecture for pattern matching[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2004, 12(7): 727-734.

[15] Gottlieb A, Grishman R, Kruskal C P, et al. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer[J]. IEEE Trans. Computers, 1983, 32(2): 175-189.

[16] Bozkus Z, Choudhary A, Fox G, et al. Compiling Fortran 90D/HPF for distributed memory MIMD computers[J]. Journal of parallel and Distributed Computing, 1994, 21(1): 15-26.

[17] 张敏华, 张剑贤, 裘雪红, 等. 基于 OpenCL 的 JPEG 压缩算法并行化设计与实现[J]. 计算机工程与科学, 2017, 39(05): 860.

[18] 廖醒宇, 余水来. 嵌入式 ARM MPcore 平台 JPEG 并行编码的研究[J]. 单片机与嵌入式系统应用, 2016, 16(4): 10-13.

[19] da Silveira T L T, Canterle D R, Coelho D F G, et al. A Class of Low-complexity DCT-like Transforms for Image and Video Coding[J]. IEEE Transactions on Circuits and Systems for Video Technology, 2021, 32(7): 4364-4375.

[20] Brahimi N, Bouden T, Brahimi T, et al. Lossy image compression based on efficient multiplier-less 8-points DCT[J]. Multimedia Systems, 2022, 28(1): 171-182.

[21] Shatnawi M K A, Shatnawi H A. A performance model of fast 2D-DCT parallel JPEG encoding using CUDA GPU and SMP-architecture[C]//2014 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2014: 1-6.

[22] Wang C, Shan R, Zhou X. APBT-JPEG image coding based on GPU[J]. KSII Transactions on Internet and Information Systems (TIIS), 2015, 9(4): 1457-1470.

[23] Zhu F, Yan H. An efficient parallel entropy coding method for JPEG compression based on GPU[J]. The Journal of Supercomputing, 2022: 1-28.

[24] Sodsong W, Jung M, Park J, et al. JParEnt: Parallel entropy decoding for JPEG decompression on heterogeneous multicore architectures[C]//Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores. 2016: 104-113.

[25] Prasanna Y L, Tarakaram Y, Mounika Y, et al. Comparison of different lossy image compression techniques[C]//2021 International Conference on Innovative Computing, Intelligent Communication and Smart Electrical Systems (ICSES). IEEE, 2021: 1-7.

[26] Jidin R, Andrews D, Peck W, et al. Evaluation of the hybrid multithreading programming model using image processing transforms[C]//19th IEEE International Parallel and Distributed Processing Symposium. IEEE, 2005: 8 pp.

[27] Samyan Q W, Sahar W, Talha W, et al. Real time digital image processing using point operations in multithreaded systems[C]//2015 Fourteenth Mexican International Conference on Artificial Intelligence (MICAI). IEEE, 2015: 52-57.

[28] Ragan-Kelley J, Barnes C, Adams A, et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines[J]. Acm Sigplan Notices, 2013, 48(6): 519-530.

[29] Coello Coello C A, Reyes Sierra M. A study of the parallelization of a coevolutionary multi-objective evolutionary algorithm[C]//MICAI 2004: Advances in Artificial Intelligence: Third

Mexican International Conference on Artificial Intelligence, Mexico City, Mexico, April 26-30, 2004. Proceedings 3. Springer Berlin Heidelberg, 2004: 688-697.

[30] Nugteren C, Corporaal H, Mesman B. Skeleton-based automatic parallelization of image processing algorithms for GPUs[C]//2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation. IEEE, 2011: 25-32.

[31] Welch E, Patru D, Saber E, et al. A study of the use of SIMD instructions for two image processing algorithms[C]//2012 Western New York Image Processing Workshop. IEEE, 2012: 21-24.

[32] Kim S, Chung W, Lee J. DISCRETE COSINE TRANSFORM (DCT)[J]. Journal of Seismic Exploration, 2021, 30: 365-380.

[33] Chen W H, Smith C H, Fralick S. A fast computational algorithm for the discrete cosine transform[J]. IEEE Transactions on communications, 1977, 25(9): 1004-1009.

[34] Ponomarenko N, Lukin V, Egiazarian K, et al. DCT based high quality image compression[C]//Image Analysis: 14th Scandinavian Conference, SCIA 2005, Joensuu, Finland, June 19-22, 2005. Proceedings 14. Springer Berlin Heidelberg, 2005: 1177-1185.

[35] Tu C, Tran T D. Context-based entropy coding of block transform coefficients for image compression[J]. IEEE Transactions on Image Processing, 2002, 11(11): 1271-1283.

[36] Fan P, Lang G, Yan B, et al. A method of segmenting apples based on gray-centered RGB color space[J]. Remote Sensing, 2021, 13(6): 1211.

[37] Lee D J. Color space conversion for linear color grading[C]//Intelligent Robots and Computer Vision XIX: Algorithms, Techniques, and Active Vision. SPIE, 2000, 4197: 358-366.

[38] Ma T, Liu C, Fan Y, et al. A fast 8× 8 IDCT algorithm for HEVC[C]//2013 IEEE 10th International Conference on ASIC. IEEE, 2013: 1-4.

[39] Patil R B, Kulat K D. Image and text compression using dynamic huffman and RLE coding[C]//Proceedings of the International Conference on Soft Computing for Problem Solving (SocProS 2011) December 20-22, 2011: Volume 2. Springer India, 2012: 701-708.

[40] Liang H, Zhang X, Cheng H. Huffman-code based retrieval for encrypted JPEG images[J]. Journal of Visual Communication and Image Representation, 2019, 61: 149-156.

[41] Singh P K, Gani A. Fuzzy concept lattice reduction using Shannon entropy and Huffman coding[J]. Journal of Applied Non-Classical Logics, 2015, 25(2): 101-119.

[42] Duda J. Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding[J]. arXiv preprint arXiv:1311.2540, 2013.

[43] Witten I H, Neal R M, Cleary J G. Arithmetic coding for data compression[J]. Communications of the ACM, 1987, 30(6): 520-540.

[44] Langdon G G. An introduction to arithmetic coding[J]. IBM Journal of Research and

Development, 1984, 28(2): 135-149.

[45] Langdon G, Rissanen J. Compression of black-white images with arithmetic coding[J]. IEEE Transactions on Communications, 1981, 29(6): 858-867.

[46] Hsieh P A, Wu J L. A review of the asymmetric numeral system and its applications to digital images[J]. Entropy, 2022, 24(3): 375.

[47] Duda J. Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding[J]. arXiv preprint arXiv:1311.2540, 2013.

[48] Gaitan V G, Gaitan N C, Ungurean I. CPU architecture based on a hardware scheduler and independent pipeline registers[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2014, 23(9): 1661-1674.

[49] Horn D R, Elkabany K, Lesniewski-Laas C, et al. The Design, Implementation, and Deployment of a System to Transparently Compress Hundreds of Petabytes of Image Files for a File-Storage Service[C]//NSDI. 2017: 1-15.

[50] In J, Shirani S, Kossentini F. JPEG compliant efficient progressive image coding[C]//Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181). IEEE, 1998, 5: 2633-2636.

[51] Patil R B, Kulat K D. Image and text compression using dynamic huffman and RLE coding[C]//Proceedings of the International Conference on Soft Computing for Problem Solving (SocProS 2011) December 20-22, 2011: Volume 2. Springer India, 2012: 701-708.

[52] Welch E, Patru D, Saber E, et al. A study of the use of SIMD instructions for two image processing algorithms[C]//2012 Western New York Image Processing Workshop. IEEE, 2012: 21-24.

[53] Arabnia H R, Oliver M A. Arbitrary rotation of raster images with SIMD machine architectures[C]//Computer Graphics Forum. Oxford, UK: Blackwell Publishing Ltd, 1987, 6(1): 3-11.

[54] Aberdeen D, Baxter J. General matrix-matrix multiplication using SIMD features of the PIII[C]//Euro-Par 2000 Parallel Processing: 6th International Euro-Par Conference Munich, Germany, August 29–September 1, 2000 Proceedings 6. Springer Berlin Heidelberg, 2000: 980-983.

[55] Jang M, Kim K, Kim K. The performance analysis of ARM NEON technology for mobile platforms[C]//Proceedings of the 2011 ACM Symposium on Research in Applied Computation. 2011: 104-106.

[56] Lindoso A, García-Valderas M, Entrena L, et al. Evaluation of the suitability of NEON SIMD microprocessor extensions under proton irradiation[J]. IEEE Transactions on Nuclear Science, 2018, 65(8): 1835-1842.

[57] Palacharla S, Jouppi N P, Smith J E. Complexity-effective superscalar processors[C]//Proceedings of the 24th annual international symposium on Computer architecture. 1997: 206-218.

[58] Smith J E, Sohi G S. The microarchitecture of superscalar processors[J]. Proceedings of the IEEE, 1995, 83(12): 1609-1624.

[59] Karkhanis T S, Smith J E. A first-order superscalar processor model[J]. ACM SIGARCH Computer Architecture News, 2004, 32(2): 338.

[60] 吴凌云. 面向高密度计算的 NoC 平台多发射技术研究[D]. 合肥工业大学, 2015.

[61] 孙凌宇, 冷明, 夏洁武, 等. 多发射处理器的指令调度算法研究[J]. 井冈山大学学报: 自然科学版, 2008 (6): 25-27.

[62] Olukotun K, Mudge T N, Brown R B. Multilevel optimization of pipelined caches[J]. IEEE transactions on computers, 1997, 46(10): 1093-1102.

[63] Jenista J C, Eom Y H, Demsky B C. OoOJava: Software out-of-order execution[C]//Proceedings of the 16th ACM symposium on Principles and practice of parallel programming. 2011: 57-68.

[64] Hwu W W, Patt Y N. Checkpoint repair for out-of-order execution machines[C]//Proceedings of the 14th annual international symposium on Computer architecture. 1987: 18-26.

[65] Farrell J A, Fischer T C. Issue logic for a 600-mhz out-of-order execution microprocessor[J]. IEEE Journal of Solid-State Circuits, 1998, 33(5): 707-712.

[66] Novack S, Nicolau A. A hierarchical approach to instruction-level parallelization[J]. International Journal of Parallel Programming, 1995, 23(1): 35-62.