计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(6): 1562−1580, 2025

BeeZip2: 高性能无损数据压缩领域专用加速器

高睿昊　史舜晨　李雪琦　谭光明
（中国科学院计算技术研究所　北京　100190）
（中国科学院大学　北京　100049）
（gaoruihao20s@ict.ac.cn）

BeeZip2: A Domain-Specific Accelerator for High Performance Lossless Data Compression

Gao Ruihao, Shi Shunchen, Li Xueqi, and Tan Guangming
（*Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190）
（*University of Chinese Academy of Sciences*, *Beijing* 100049）

Abstract　High-performance and intelligent computing applications require massive data. The transfer and storage of data pose challenges for computer systems. Data compression algorithms reduce storage and transmission costs, making themselves crucial for improving system efficiency. Domain-specific hardware design is an effective way to accelerate data compression algorithms. The emerging data compression software, Zstandard, significantly enhances throughput and compression ratio. Zstandard is based on LZ77 compression algorithm, but it has a larger sliding window that increases on-chip storage overhead. Also, it has complex data dependencies and control flow. These features limit the effect of hardware acceleration. To improve the throughput while achieving a similar compression ratio for data compression accelerator in the context of large sliding windows, we propose a cross-layer optimization approach based on algorithm-architecture co-design to develop a novel data compression acceleration architecture, BeeZip2. First, we introduce the MetaHistory match method into the design of the large sliding window parallel Hash table, offering regular parallelism and addressing control flow data dependency. Then, we propose the shared match PE architecture, distributing the large sliding window across multiple processing units to share on-chip memory and reduce overhead. In addition, the Lazy match strategy and corresponding architecture help to fully leverage the resources for a higher compression ratio. Experimental results show BeeZip2 achieves 13.13 GB/s throughput while maintaining the software compression ratio. Compared with single-core and 36-core CPU software implementations, throughput increases by 29.2 times and 3.35 times, respectively. Compared with the baseline accelerator BeeZip, BeeZip2 achieves a 1.26 times throughput improvement and a 2.02 times throughput-per-area enhancement under the constraint of maintaining a higher compression ratio than its software counterpart.

Key words　domain-specific architecture; data compression; accelerator; high-performance computing; computer architecture

摘　要　领域专用加速器设计有望进一步提升数据压缩算法的性能，以适应更大规模的数据处理. 新兴的 Zstandard 压缩软件基于 LZ77 压缩算法，具有性能优势，但其“控制流数据依赖”与“滑动窗口扩大”的特征限制了加速器的性能发挥. 新型数据压缩加速器 BeeZip2 实践“算法-架构”跨层优化方法，首先，将“元历史匹配”与并行哈希表设计融合，应对控制流数据依赖问题. 然后，BeeZip2 采用“共享匹配处理单元”架

收稿日期：2025-01-10；**修回日期**：2025-04-09
基金项目：国家自然科学基金项目（T2125013, 62032023）
This work was supported by the National Natural Science Foundation of China (T2125013, 62032023).
通信作者：谭光明（tgm@ict.ac.cn）

构及组织方式，减少大滑动窗口的开销. 此外，BeeZip2 还包含“简易惰性匹配”策略及架构设计，提高“元历史匹配”和“共享处理单元”的利用效率. 实验结果表明，BeeZip2 在达到软件相同压缩比的同时，可实现最高 13.13 GB/s 的吞吐率，相较于单核和 36 核 CPU 软件吞吐率分别提升了 29.2 倍和 3.35 倍. 与基线加速器 BeeZip 相比，BeeZip2 在压缩比高于软件的约束下，吞吐率提升 1.26 倍，单位面积吞吐率提升 2.02 倍.

关键词 领域专用架构；数据压缩；加速器；高性能计算；计算机体系结构

中图法分类号 TP302.2

DOI: 10.7544/issn1000-1239.202550017 **CSTR:** 32373.14.issn1000-1239.202550017

现如今，生成式人工智能[1-2]、推荐系统[3-5]以及人工智能科学计算[6-8]等技术应用引起了广泛关注，产生了巨大的经济价值. “数据驱动”的应用需要依托大量数据进行开发和构建，在此背景下，海量数据不断涌入计算机系统，由此带来的数据存储传输的性能与成本问题日益突出[5]. 数据压缩算法可以在保留原始信息的基础上有效缩减数据规模，从而减少存储与传输的数据量[9]，是提高计算机系统效率的重要手段[10]. 随着数据规模和需求的持续扩大，发展高性能的数据压缩方法具有重要现实意义.

领域定制硬件设计是加速数据压缩算法的有效途径. 一方面，得益于数据压缩算法的标准化[11-13]，领域定制硬件加速数据压缩的成本可由其广泛应用场景和良好兼容性分摊. 另一方面，将固定的数据压缩算法从通用架构上卸载，可将算力受让于其他灵活多变的负载以发挥更高价值[10-14]. 现有研究[10,14-18]采用领域定制硬件方法加速基于字典的 LZ77 无损压缩算法[19]，取得了显著性能提升. 这些研究从微架构层面探索加速方法，形成了基于历史移位寄存器和基于并行哈希表的 2 类微架构分化[16]，也从系统集成角度提出了数据压缩加速器与现有通用架构集成的方案.

随着数据压缩工具 Zstandard[20] 的出现，现有数据压缩加速器设计面临新的挑战. Zstandard 凭借其优异的吞吐率、压缩比表现获得广泛应用. 根据相关研究[14]中的数据测算，Zstandard 现已成为 Google 数据中心中消耗 CPU 周期最多的数据压缩工具. 尽管 Zstandard 仍基于 LZ77 算法，但其中关键数据结构“滑动窗口”的规模显著扩大，相较于现有加速器普遍支持的 gzip/zlib[21] 扩大了 16~32 倍乃至更多. 大滑动窗口导致片上存储开销大幅增加，挑战现有架构设计的可行性，成为数据压缩加速设计的首要挑战. 此外，Zstandard 存在数据依赖的复杂控制流，与领域定制硬件追求的规整并行性相矛盾，限制了加速器性能的发挥[16].

针对大滑动窗口 LZ77 算法加速的 2 个挑战，本文采用“算法-架构”跨层优化方法，设计了新型数据压缩加速架构 BeeZip2. 首先，在算法层面，本文将 MetaZip[15] 中提出的元历史匹配机制引入大滑动窗口场景，与 BeeZip[16] 中的 BeeHash 算法整合，构成 Meta-BeeHash 算法；在 BeeZip[16] 提出的 MatchJob 并行方法基础上，增加提高压缩比且受控制流复杂影响小的简易惰性匹配策略，构成 LazyHiveMatch 算法；同时，提出 MatchJobSerializer 算法，用于消除 MatchJob 并行重叠，最大化利用匹配结果.

随后，在架构层面，本文设计了支持上述 2 个算法功能的加速引擎，添加功能模块以满足算法改进需求. 特别地，在 LazyHiveMatch 加速引擎中，本文提出了降低大滑动窗口开销的共享匹配处理单元架构及调度方法. 通过上述创新改进，本文实现了与 Zstandard 软件“压缩比对标，吞吐率提高”的性能目标，主要贡献有 4 点：

1）提出将元历史匹配方法加入大滑动窗口并行哈希表的设计方案，提供规整并行度，解决控制流数据依赖问题；同时，元历史匹配和归并操作可减少后续流程需要的访存次数，有利于大滑动窗口开销的缩减.

2）提出共享匹配处理单元架构及组织方式，将大滑动窗口拆分到多个分布式的处理单元中，实现片上存储资源分片共享，有效减少大滑动窗口的开销；配合本地匹配处理单元提供的低延迟、小副本，吞吐率表现优于现有加速器基线.

3）提出简易惰性匹配策略及架构实现，旨在受不规则控制流影响小的前提下，尽可能充分利用元历史匹配和共享匹配处理单元资源，实现更高的压缩比.

4）实验结果显示，本文提出的 BeeZip2 数据压缩加速器在压缩比达到软件标准的同时，相较于单核和 36 核软件实现，吞吐率分别提升了 29.2 倍和 3.35 倍. 与加速器基线 BeeZip 相比，最高吞吐率提高 1.26 倍，单位面积吞吐率提升 2.02 倍.

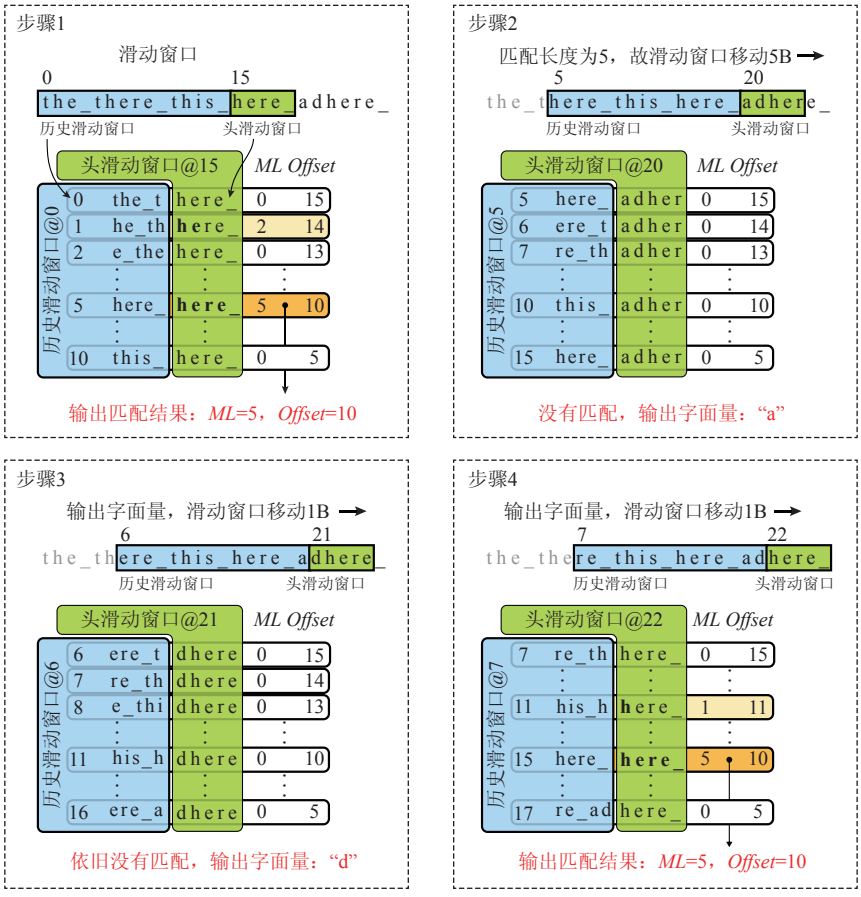
1 研究背景

1.1 数据压缩算法简介

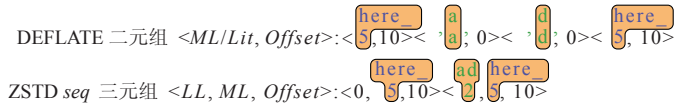
LZ77 是一种基于字典的无损压缩算法，由 Ziv 等人[19] 于 1977 年提出，在过去的 50 年中得到了广泛的应用. 众多知名的数据压缩工具，例如 gzip/zlib, 7zip/xz (LZMA) 和 Zstandard , 均基于 LZ77 算法或其变体.

LZ77 算法实现压缩数据的方式是查找和消除重复冗余. 图 1(a)展示了 LZ77 算法的执行流程. 输入的原始数据存储在滑动窗口数据结构中，滑动窗口可进一步分为历史滑动窗口和头滑动窗口 2 部分，二者具有固定的长度，在图 1(a)中，历史滑动窗口长度为 15 B 字节，头滑动窗口长度为 5 B 字节. LZ77 查找头滑动窗口中的子串在历史滑动窗口中最长的前缀匹配. 在步骤 1 中，头滑动窗口可在历史滑动窗口中的第 5 B 字节处找到最长的前缀匹配，故算法输出匹配长度 *ML*=5，匹配距离 *Offset*=10，然后转向步骤 2. 在步骤 2 开始前，由于上一步找到了长度为 5 的匹配，需要首先将滑动窗口向后移动 5 B. 步骤 2 遇到了边界情况：在历史滑动窗口中无法找到任何头滑动窗口的前缀匹配. 在无匹配的情况下，LZ77 输出字面量(Lit)，在此处即为头滑动窗口中第 1 个字“a”. 输出字面量保证了 LZ77 压缩后的数据可以被完整恢复，故称为无损压缩算法. 在步骤 2 之后，由于只输出 1 B 的字面量，故滑动窗口向后移动 1 个字节，转到后续步骤 3, 4, 重复运行直到输入数据被完全处理.

为了提高 LZ77 算法的执行效率，在 gzip, Zstandard 等软件实现中，使用哈希表降低滑动窗口查找的复杂度. 算法根据头滑动窗口的头部数个字节计算哈



(a) LZ77算法的执行流程



(b) DEFLATE和ZSTD Seq实现中采用的方式

Fig. 1 Illustration of execution workflow and output data format of LZ77 algorithm

图 1 LZ77 算法的执行工作流程以及输出数据格式示意图

希值，索引哈希表得到有限数量的潜在匹配位置，所需匹配查找操作的数量相对滑动窗口长度由$O(n)$降至$O(1)$.

吞吐率和压缩比是衡量压缩算法性能的2个指标. 吞吐率是指压缩算法在单位时间内能够处理的输入字节数，压缩比是指原始输入文件的大小除以输出文件的大小，二者数值均应尽可能提高.

滑动窗口是LZ77中的一个重要概念. 设置滑动窗口的最初目的是限制搜索范围，提高算法的吞吐率. 然而，现代压缩算法为了追求更高的压缩比，滑动窗口的大小被扩大到KB甚至MB级别. 更大的滑动窗口意味着更多的访存次数、更大的访存范围以及更高的计算比较频率，如何处理大滑动窗口中的匹配对压缩算法的性能至关重要.

不同压缩工具实现中，LZ77输出的数据格式略有不同，图1(b)对比了DEFLATE和Zstandard实现中采用的不同方式. 本文采用Zstandard中使用的*seq*三元组〈*LL*, *ML*, *Offset*〉，其中*LL*表示字面量长度，*ML*表示匹配长度，*Offset*表示匹配距离. 数据中的重复由匹配长度和匹配距离标识，字面量单独保存，由*seq*中的*LL*按顺序索引. LZ77算法输出的*seq*三元组需要进一步熵编码以实现更高的压缩比. 考虑到LZ77算法是当前压缩工具的性能瓶颈，本文将重点关注LZ77部分.

1.2 数据压缩加速器相关工作

文献[10, 14−18, 22-24]中提出了多种数据压缩加速器设计方案. 上述相关工作可分为关注加速器微结构设计和关注系统集成方式两大类，结合本文研究内容，本节重点讨论关注微结构设计的相关工作，更多细节将在第8节介绍. 关注微结构设计的相关工作可进一步分为基于历史移位寄存器(history shift register, HSR)和基于并行哈希表(parallel Hash table, PHT)两大类.

基于HSR微结构的加速器[10,18]泛化的微结构设计如图2所示. 此类加速器将滑动窗口保存在移位寄存器中，移位寄存器的长度与滑动窗口大小相同，使用比较器阵列并行地探查移位寄存器中的所有位置，寻找潜在匹配和选择最优结果. 由于比较器阵列涉及大量的硬件资源，为了最大限度地减少时序和面积负担，基于HSR结构的加速器支持的滑动窗口大小相对较小. 文献[10]工作仅支持512 B滑动窗口；文献[18]虽然采用了64 KB的移位寄存器，但与其关联的比较器阵列数量只有5 000左右，意味着移位寄存器中的滑动窗口不能被充分探查.

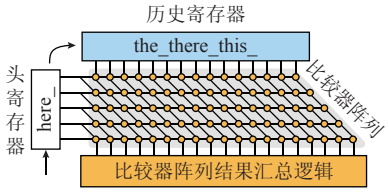


Fig. 2 Generalized historical shift register accelerator microstructure

图2 泛化的历史移位寄存器式加速器微结构

基于PHT微结构的加速器[15−17, 22−24]参考了软件实现中的哈希方法，泛化的结构设计如图3所示. 相关工作[17,24]将哈希表拆分存储在为多个SRAM单元中，实现了并行化的哈希查找和更新操作. 文献[15]在上述工作基础上提出“元历史匹配”方法，将长度为8 B的历史窗口片段放入哈希表中，而文献[22]中结构设计直接丢弃单独的历史缓冲区并仅使用融合哈希历史表. PHT微结构相对于HSR微结构具有更好的可扩展性，但是，并行哈希表操作之间存在结构相关冲突，即多个并行处理的哈希值可能同时映射到同一个并行哈希表分片上. 文献[15−16]的实验结果表明该冲突现象会降低压缩比.

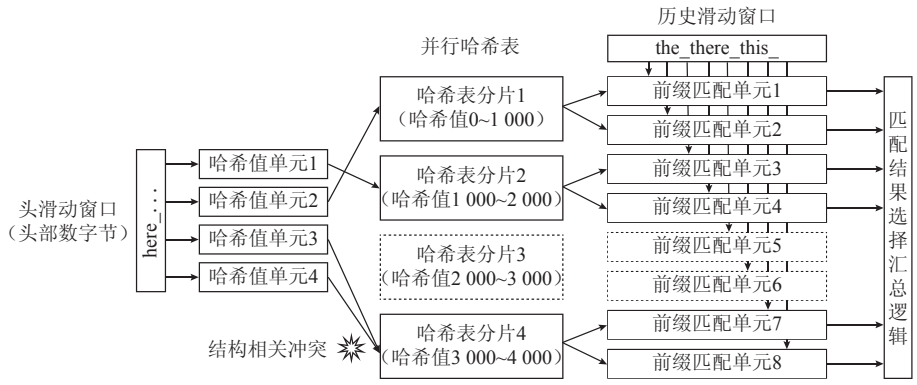


Fig. 3 Generalized parallel Hash table accelerator microstructure

图3 泛化的并行哈希表式加速器微结构

BeeZip[16]数据压缩加速架构在分类上属于PHT类型，支持Zstandard算法的加速，采用动态调度、多级并行算法-架构协同、异构规模处理单元等设计方法，在大滑动窗口LZ77算法的加速上取得了富有竞争力的结果，是本文的重要基线和对比目标. 本文在BeeZip的基础上进一步探索提升性能和减少面积开销的方法.

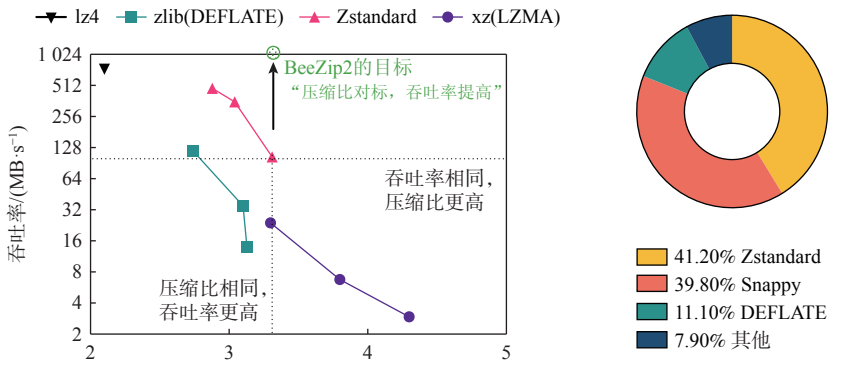
2 本文的动机、目标和挑战

本文选择Zstandard中的LZ77算法作为主要加速目标，动机理由如下：首先，Zstandard作为一种新兴的压缩工具，与其他压缩工具相比具有性能优势. 如图4(a)所示，相比于其他常用压缩工具，Zstandard在吞吐率和压缩比2个关键指标上具有均衡的提升. 得益于上述性能优势，Zstandard被操作系统内核[25]、数据库系统[26]、科学计算[27-29]等关键场景所接受. 最后，文献[14]中提供的分析结果显示(参照图4(b))，Zstandard已成为Google数据中心中消耗CPU周期最多的压缩工具. 同时，文献[16]中的测试结果显示，LZ77算法占据Zstandard工具的主要执行时间.

本文的目标如图4(a)所示，采用算法-架构协同设计方法，实现压缩比与软件实现对标以及吞吐率更高的数据压缩专用加速架构，提升数据压缩技术处理大规模数据的能力. 然而，上述动机与目标面临两大挑战：

1)更大的滑动窗口. Zstandard中的LZ77算法具有更大的滑动窗口. 相关工作[10,15,17,22,24]所支持的DEFLATE算法受到格式约束，滑动窗口最大不超过32 KB. 其中，文献[10]提出的关键结构nearCAM仅能支持512 B的滑动窗口；文献[17, 24]并未讨论滑动窗口数据保存的细节；文献[15]发现了滑动窗口资源开销的问题，但由于滑动窗口较小，故仍采用了滑动窗口副本的方法. 相比之下，Zstandard所配置的滑动窗口大小从512 KB起步，大滑动窗口意味着更好的压缩比[16]，也意味着更大的硬件开销. 尽管采用了多种优化手段，支持大滑动窗口的BeeZip[16]加速架构仍然需要13 MB左右片上存储空间用于保存滑动窗口.

2)数据依赖导致的不规则控制流. LZ77算法中包含的数据依赖不规则控制流是领域特定加速器设计的挑战[30-31]. 如1.1节所述，LZ77算法的执行流程包括匹配长度计算、滑动窗口移动等操作，上述操作的推进与输入数据直接相关，控制流复杂且不可预测，难以并行加速. 文献[16]中在通用处理器上的微结构测试结果显示，Zstandard中的LZ77算法执行时，有20%~30%的流水线槽出现分支预测错误.



(a) 各压缩工具的性能对比　(b) Google数据中心多种数据压缩工具的多种情况

Fig. 4 Performance advantages and industry adoption statistics of Zstandard[14,16,32]

图4 Zstandard的性能优势与行业采用数据统计[14,16,32]

3 BeeZip2并行压缩算法设计

面向大滑动窗口和不规则控制流带来的挑战，BeeZip2采用算法-架构跨层设计优化方法，首先从算法层面入手，提出新的LZ77算法并行化方法，包括MetaBeeHash算法、LazyHiveMatch算法以及Match-JobSerializer算法. 本节将分别介绍上述算法的流程以及优势.

3.1 MetaBeeHash算法

MetaBeeHash算法将文献[15]中提出的元历史匹配方法MetaHistory与文献[16]中的BeeHash算法整合，数据结构及执行流程如图5所示，实现并行化融合的哈希、匹配探查操作. MetaBeeHash并行接受

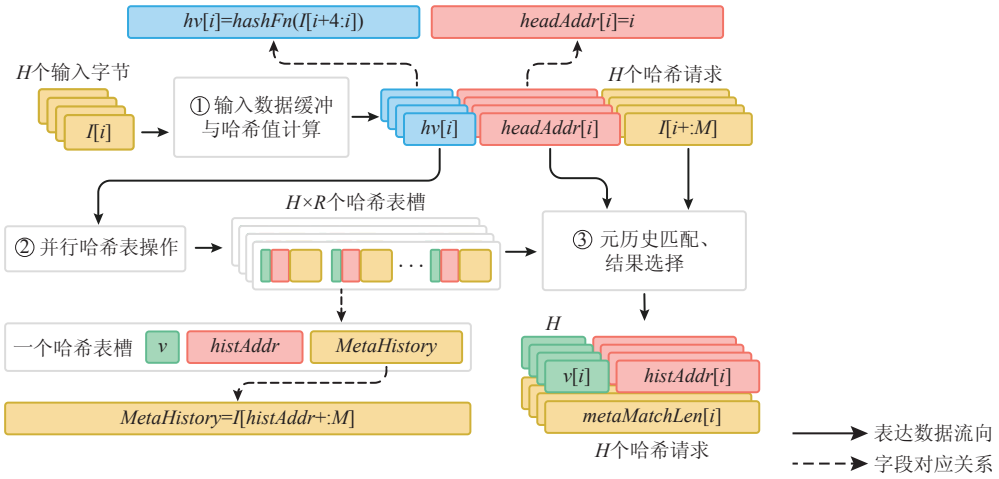


Fig. 5 Execution flow and data structure of MetaBeeHash algorithm

图 5 MetaBeeHash 算法的执行流程以及数据结构

H 个输入数据字节，首先，经由数据缓冲和哈希值计算流程产生 H 个哈希请求，与 H 个输入字节按序对应. 每个哈希请求包含哈希值(*hv*)、头地址(*headAddr*)和输入片段(I)共 3 个字段. 其中输入片段 I 的长度为 M 字节. MetaBeeHash 算法约束 $M<H$，在此限制下，输入数据缓冲仅需保存 2 组 H 字节输入即可实现完整的并行哈希请求生成.

然后，哈希请求并行处理，每个哈希值均可读出 R 个哈希表槽(Hash table slot)，因此，H 个哈希请求共产生 $H\times R$ 个哈希表槽. 每个哈希表槽包含有效位(v)、历史地址(*histAddr*)以及长度同为 M 字节的 *MetaHistory* 字段. 根据串行 LZ77 算法的特点，H 个哈希请求在完成读取后还需插入到哈希表的对应行中，插入过程中，*headAddr* 作为 *histAddr*，输入片段(I)成为 *MetaHistory*.

接下来，每个哈希请求与对应的 R 个哈希表槽进行元历史匹配和结果选择操作. 该操作先将哈希请求的输入片段(I)与每个哈希表槽中的 *MetaHistory* 进行前缀匹配，得到匹配长度(*metaMatchLen*)，再根据 *histAddr* 与 *headAddr* 的差判断该匹配距离是否在滑动窗口内，此外还判断匹配长度是否达到 Zstandard 要求的最小匹配长度(4 B)，若匹配距离、匹配长度均满足要求，则成为一个有效的哈希结果. 当一个哈希请求存在多个有效的哈希结果时，结果选择操作会选择其中匹配长度最长、匹配距离最小的一项. 最终 MetaBeeHash 算法输出 H 个包含 v、*histAddr* 和 *metaMatchLen* 的哈希结果，供给后续的 LazyHiveMatch 算法使用.

从控制流复杂性的角度分析，哈希查找和 *MetaHistory* 匹配操作的融合并行展开具有规整的控制流，不依赖于输入数据，有利于领域定制硬件架构的实现.

3.2 LazyHiveMatch 算法

LazyHiveMatch 算法在 BeeZip 中 HiveMatch 算法基础上增加了简易惰性匹配策略，执行流程如图 6 所示. LazyHiveMatch 采用匹配任务(MatchJob)并行策略，$J\times H$ 个哈希结果一组，按输出顺序组成一个 MatchJob, LazyHiveMatch 算法并行处理 N 个 MatchJob.

单个 MatchJob 的处理包含了简易惰性匹配策略，即：发现一个有效哈希结果时(步骤①)，并不立刻扩展、输出，而是将从该哈希结果位置开始的 L 个哈希结果同时纳入考虑(步骤②). 如果一个哈希结果中的 metaMatchLen 达到了 M，则意味着该结果需要进一步扩展(步骤③)，匹配扩展操作需要访问滑动窗口. 若参与惰性匹配的 L 个请求中有多个需要进一步扩展，则这些操作可以并行执行. 在完成扩展操作后，需要对所有 L 个哈希结果(包括扩展后的哈希结果)计算收益 g(步骤④)，规则如下：

1)若哈希结果无效，$g=0$，

2)若哈希结果有效，$g=4ML-4LL-bits(Offset)$，

其中 ML 表示匹配长度，LL 表示字面量长度，*Offset* 表示匹配，*bits*( ) 函数求解保存 *Offset* 数值所需的最少二进制比特数. 在 L 个哈希结果中选择匹配收益 g 最大的一项作为输出(步骤⑤)，然后根据匹配长度向后移动指针(步骤⑥). 最终，重复步骤①~⑥直到当前 MatchJob 处理完成.

从不规则控制流角度分析，LazyHiveMatch 算法中 MatchJob 和简易惰性匹配策略提供规整、不依赖于输入数据的并行性. 从减少大滑动窗口开销角度分析，单个 MatchJob 处理时的串行处理跳过冗余的

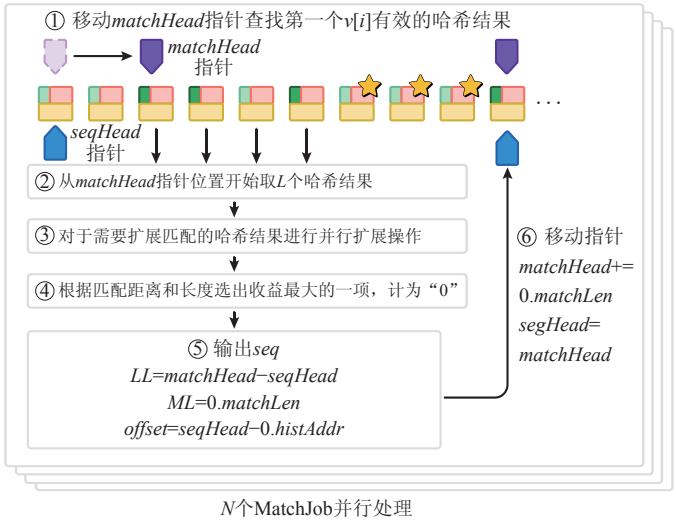


Fig. 6 Execution flow of LazyHiveMatch algorithm

图 6 LazyHiveMatch 算法的执行流程

匹配扩展操作(图 6 中星标表示)，可减少匹配扩展操作的数量，为访存资源的缩减提供机会. 简易惰性匹配策略能够充分利用既有的哈希结果，有利于压缩比的提高.

3.3 MatchJobSerializer 算法

LazyHiveMatch 算法按照 MatchJob 并行的输出结果，可能存在空隙(gap)或重叠(overlap)问题，本文提出 MatchJobSerializer 算法予以消除这些问题，算法面临的情况以及处理方法如图 7 所示. 空隙和重叠只可能出现在一个 MatchJob 的尾部，当 MatchJob 尾部无有效匹配结果时，出现 *LL*>0 且 *ML*=0 的空隙 *seq*，如图 7(a)所示，这种 *seq* 在 Zstandard 中被视为数据块结尾，不应出现在数据块内部，故需要合并到下一个 *seq* 的字面量长度中. 当 MatchJob 尾部的最后一个 *seq* 的匹配长度越过 MatchJob 边界，则出现重叠. 图 7(b)~(d)展示了 3 种简单的重叠情况及处理方法. 在图 7(c)中，由于重叠修剪后的 *seq*[*i*+1]. *ML* 仍然大于等于 Zstandard 算法支持的最小匹配长度，可以正常输出. 若修剪后剩余的 *ML* 大于 0 但小于最小匹配长度，则需进一步按照图 7(e)情况处理. 最复杂的重叠情况如图 7(f)所示，*seq*[*i*] 完全覆盖了 *seq*[*i*+1]，MatchJobSerializer 算法将这种情况视作 MatchJob 边界向后移动，之后即可转换到图 7(b)~(e)的情况进一步处理.

MatchJobSerializer 算法设计遵循 2 个原则：1)尽可能保留匹配结果；2)保持流式处理方式. 第 1 个原则旨在提高压缩比，在 MatchJobSerializer 算法中，只有图 7(e)涉及匹配长度向字面量长度的转换，且转换长度不超过最小匹配长度. 第 2 个原则遵从数据压缩算法的流式特性，具体表现为 *seq*[*i*] 总是保持原状输出，并将修改结果合并到后续的 *seq*[*i*+1] 中，该特性可递归推知已输出的 *seq* 不需要再进行修改.

4 BeeZip2 加速器整体架构设计

BeeZip2 加速器的整体架构如图 8 所示，硬件部分主要由 MetaBeeHash 和 LazyHiveMatch 这 2 个加速引擎构成，在领域定制硬件架构层面支持大滑动窗口 LZ77 算法. BeeZip2 硬件加速引擎辅以软件实现的 MatchJobSerializer、熵编码来构成完整的数据压缩加速系统，将原始输入数据处理成兼容 Zstandard 格式的压缩后数据.

在模块功能层面，MetaBeeHash 加速引擎和 LazyHiveMatch 加速引擎分别对应于 MetaBeeHash 和 LazyHiveMatch 算法. 在数据流层面，BeeZip2 接收字节流形式的原始数据，于 MetaBeeHash 加速引擎中处理成包含元历史匹配的哈希结果，再由 LazyHiveMatch 加速引擎产生完整匹配结果并输出 *seq* 三元组. 现阶段，MatchJobSerializer 算法和熵编码由软件实现，出于以下 2 点理由，本文假设软件实现部分不影响吞吐率. 1)LZ77 算法在 Zstandard 压缩流程中占据主要执行时间；2)软件部分的串行化操作、熵编码操作可通过多线程并行或者进一步的硬件设计[10,14-17]加速，在吞吐率方面与 LZ77 部分对齐(但具体的加速结构设计超出了本文的讨论范围). BeeZip2 关注的重点在于 LZ77 算法的加速架构实现，本文第 5~6 节将分别介绍 MetaBeeHash 和 LazyHiveMatch 加速引擎的架构设计.

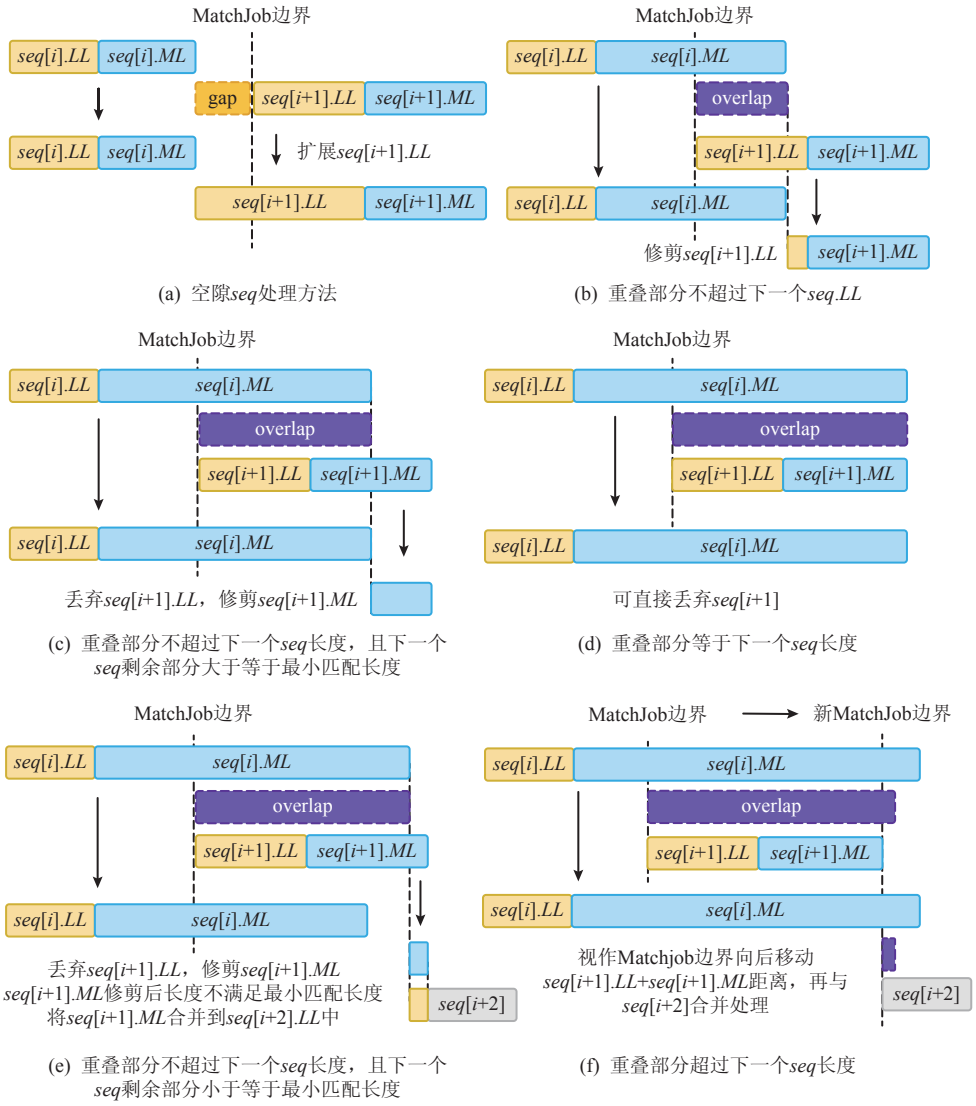


Fig. 7 Challenges faced by MatchJobSerializer algorithm and solutions

图 7 MatchJobSerializer 算法面临的挑战及处理方法

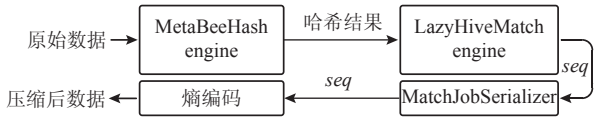


Fig. 8 Overall architecture of BeeZip2 accelerator

图 8 BeeZip2 加速器整体架构

5 MetaBeeHash 加速引擎架构设计

在功能层面，MetaBeeHash 加速引擎实现 MetaBeeHash 算法要求的并行化哈希操作以及元历史匹配操作. MetaBeeHash 加速引擎同步接收 H 个字节输入数据，输出 H 个哈希结果. 在结构层面，如图 9 所示，MetaBeeHash 加速引擎由 H 个执行计算访存操作的哈希函数模块、H' 个哈希处理单元和实现协调调度的哈希请求调度器、哈希结果同步器组成. 面向大滑动窗口开销问题，哈希处理单元提供了元历史匹配支持；面向不规则控制流问题，哈希请求调度器和哈希结果同步器提供了运行时可配置的动态调度能力.

5.1 哈希函数模块与哈希处理单元

MetaBeeHash 加速引擎中的哈希函数模块选用与 Zstandard 软件实现相同的 5 B 整数哈希方法，相较于文献 [15, 17] 以及 gzip/zlib 软件采用的异或哈希方法，碰撞情况更少. 哈希处理单元的结构如图 10 所示，主体由 R 个 SRAM Bank 和与之对应的长度计算、距离计算单元构成. 哈希处理单元具有非阻塞流水线结构，每周期可接收 1 个哈希请求，使用哈希请求中的哈希值作为读地址访问（SRAM）. R 个 SRAM Bank 输出 R 个哈希槽，哈希槽中的历史地址、元历史片段分别送往距离计算单元和长度计算单元，并与哈希请求中的对应字段进行比较. 长度计算单元由按字

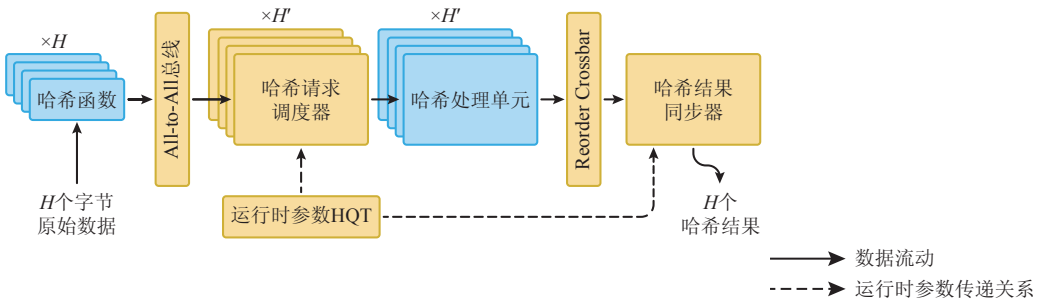


Fig. 9 Overall structural design of MetaBeeHash accelerated engine

图 9 MetaBeeHash 加速引擎整体结构设计

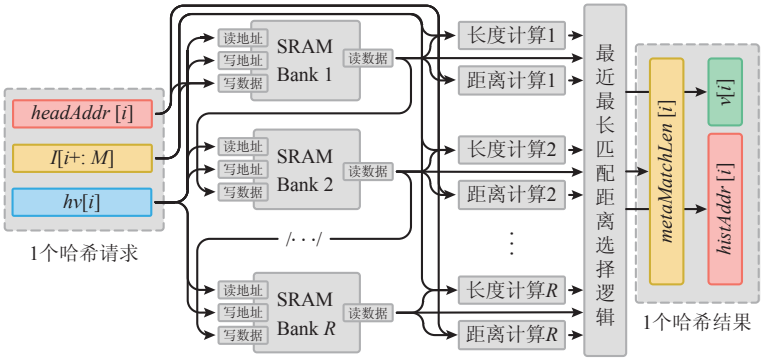


Fig. 10 Hash PE architecture design

图 10 哈希处理单元结构设计

节比较的异或逻辑以及产生匹配长度的优先编码逻辑构成，产生哈希请求中输入数据片段与哈希槽中元历史片段的前缀匹配长度. 最长匹配选择逻辑按二叉树状结构进行归并，在归并过程中根据算法要求选择有效且匹配长度最优者，若有多个相同匹配长度结果出现，则选择偏移最小者；若所有输入元历史匹配结果均无效，则该逻辑输出的哈希结果标记为无效. 在架构实现中，由于 SRAM 读取操作存在至少 1 周期延迟，因此需要在写地址输入、读数据到写数据路径上加入寄存器，但哈希处理单元架构仍然具有流水线形式，不影响吞吐率.

与 BeeZip[16] 中的 BeeHash Engine 对比，MetaBeeHash 加速引擎的哈希处理单元增加了元历史匹配能力，且能够在元历史匹配结果产生后立即进行最优结果的归并选择，减少了后续流程需要处理的哈希结果数量，有助于吞吐率的提升和大滑动窗口开销的消减. 与已采用元历史匹配机制的 MetaZip[15] 对比，一方面，MetaBeeHash 加速引擎的归并方法在单个哈希请求层面进行，相比于 MetaZip 的全局归并方法更合理；另一方面，通过增加合理的流水线寄存器，该哈希处理单元可使用更通用的单读单写双端口 SRAM 原语搭建，适用于 ASIC/FPGA 流程，不再依赖 MetaZip 中特殊的 FPGA LUTRAM 资源.

从大滑动窗口开销的角度分析，本文研究测试了 Zstandard 软件在压缩等级 level 1~3 上产生的匹配长度分布. 如图 11 所示，当元历史片段长度 M=3 时，54.9% 的匹配操作在 MetaBeeHash 阶段即可完成，后续访存操作的减少意味着片上存储资源的优化. 另一方面，尽管在哈希表中增加历史片段的设计在直觉上违背“减少片上存储”的原则，但本文研究发现，在 Zstandard 和 BeeZip 实现中，为了减少由哈希值碰撞引发的额外无效访存操作，哈希表包含 1×10^6 乃至更多个哈希表槽；而 MetaBeeHash 算法对小匹配长度

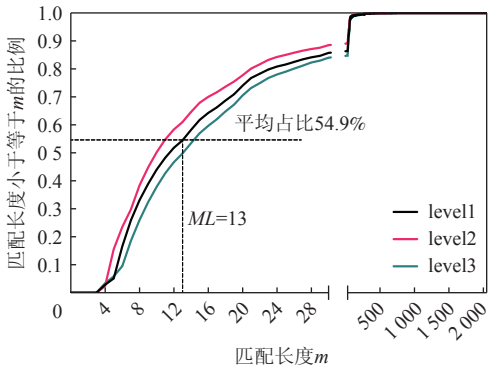


Fig. 11 Match length distribution in Zstandard software implementation of three compression levels

图 11 Zstandard 软件实现中 3 个压缩等级上的匹配长度分布

的过滤可以消除哈希值碰撞导致的影响，故哈希槽的数量在保证相同压缩比表现的前提下，可以减少至 256×10^3 项（减少至原先的 1/4），即使增加的元历史片段使得单个哈希槽的大小扩大了 4 倍，保存哈希表所需的总 SRAM 容量大小仍保持不变.

5.2 MetaBeeHash 加速引擎中的协调调度模块

MetaBeeHash 加速引擎中的协调调度模块包括哈希请求调度器和哈希结果同步器. 哈希请求调度器和哈希结果同步器可实现高效的动态调度，减少控制流数据依赖带来的负面影响. 在 MetaBeeHash 加速引擎中，由于完整的哈希地址空间被均分到 H' 个哈希处理单元中，每个哈希处理单元负责不同的地址空间，存在同一周期、多个哈希请求映射到单个哈希处理单元的情况. 考虑到哈希处理单元每周期仅能接受 1 个输入请求，故在每个哈希处理单元前方设置一个哈希请求调度器，结构如图 12 所示. 哈希请求调度器围绕哈希请求调度表设计. 哈希请求调度器的工作流程为：在 H 个输入哈希请求中依照哈希值筛选出当前哈希处理单元能够处理的请求，若有多个哈希请求，则按顺序依次发出，最多发出数量由运行时参数 *HQT* 决定.

哈希结果同步器的结构如图 13 所示. H' 个哈希处理单元并行工作，每周期最多输出 H' 个哈希结果，经过 Reorder Crossbar 排序后（见图 9），在哈希结果同步器中汇总. Reorder Crossbar 的排序依据是哈希结果对应的输入地址低位，由于 H' 个哈希结果一定来自于 H 个哈希请求，所以排序不会出现冲突. 哈希结果同步器在接收到 H 个结果或者达到固定握手次数后，

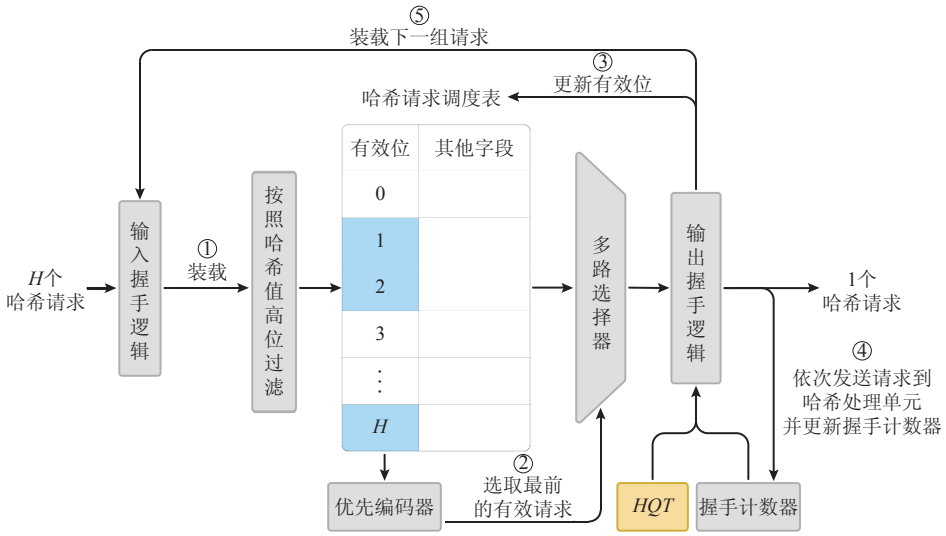


Fig. 12 Architecture and function of Hash request scheduler

图 12 哈希请求调度器结构与功能

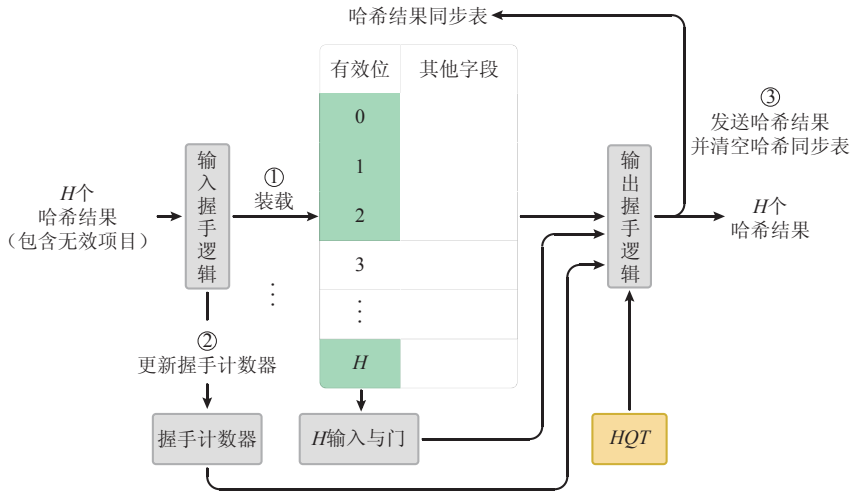


Fig. 13 Architecture and function of Hash result synchronizer

图 13 哈希结果同步器结构与功能

将最多 H 个哈希结果同步输出.

哈希请求调度器和哈希结果同步单元均受到参数 *HQT* 控制，*HQT* 限制了哈希请求调度器的最大调度数量. 例如，当 *HQT* 设置为 2 时，在收到一组 H 个请求后，所有哈希调度器最多调度 2 个请求，剩余的请求被丢弃. 哈希结果同步器根据 *HQT* 确定每一组哈希结果的最长等待时间. *HQT* 为运行时可配置参数，保存在寄存器中. 修改 *HQT* 可实现 BeeZip2 架构的吞吐率-压缩比权衡调整. 当 *HQT* 值较小时，哈希处理单元的排队请求数量少，吞吐率提高，但同时丢弃的排队请求增多，减少了哈希处理单元发现匹配的机会，所以压缩比亦会下降. 在实际运行中，哈希请求调度器和哈希结果同步器既提供合理的排队支持，尽量减少哈希插入的丢失，有益于压缩比的改善，又通过运行时可配置参数限制排队长度，限制对吞吐率的负面影响.

6 LazyHiveMatch 加速引擎架构设计

为了更有力地应对大滑动窗口开销问题，本文重新设计了 BeeZip 架构中的 HiveMatch 加速引擎设计，提出了效率更高的 LazyHiveMatch 加速引擎架构. 在功能层面，该加速引擎是 LazyHiveMatch 算法的领域定制硬件架构实现，接受来自 MetaBeeHash 加速引擎输出的哈希结果，按需实施惰性匹配扩展，并选取收益高的匹配结果输出成 *seq* 三元组. 结构层面，如图 14 所示，LazyHiveMatch 加速引擎由一组任务匹配处理单元集群(job match PE cluster, JMPC)和共享匹配处理单元(shared match PE, SMPE)组成. JMPC 和 SMPE 之间通过总线和 Mesh 网络互连.

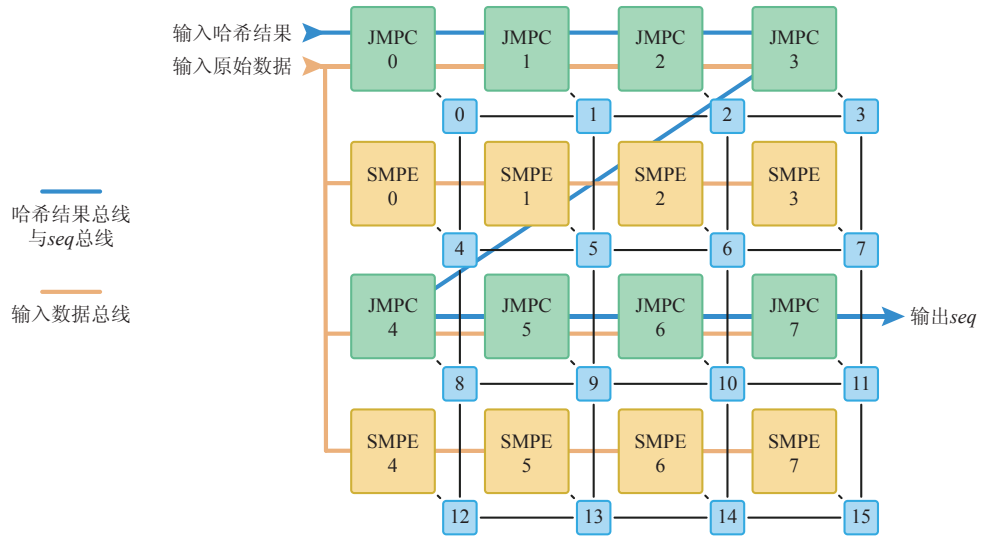


Fig. 14 Overall architecture design of LazyHiveMatch accelerated engine

图 14 LazyHiveMatch 加速引擎整体结构设计

JMPC 由匹配任务处理单元(job PE, JPE)、匹配请求调度器、多种规模的本地匹配处理单元(local match PE, LMPE)及匹配响应同步器组成，结构如图 15 所示. JPE 实现 LazyHiveMatch 算法中除步骤③(见图(6))匹配扩展之外的操作，匹配扩展操作在匹配处理单元(match PE, MPE)中进行，MPE 进一步分化为 SMPE 和 LMPE 两种类型. 匹配请求调度器与匹配响应同步器实现 JPE 与匹配处理单元 SMPE, LMPE 之间的协同.

MetaBeeHash 引擎的输出经过哈希结果总线进入 JMPC，JMPC 输出的 *seq* 经过 *seq* 总线按序输出，上述 2 条总线的结构如图 16 所示，由哈希总线节点(Hash bus node, HBN)与 *seq* 总线节点(Seq bus node, SBN)串联组成. 在任一时刻，输入的哈希结果仅进入特定 1 个 JMPC，也仅有 1 个特定的 JMPC 输出 *seq*. MetaBeeHash 引擎的输出包含哈希结果的头地址信

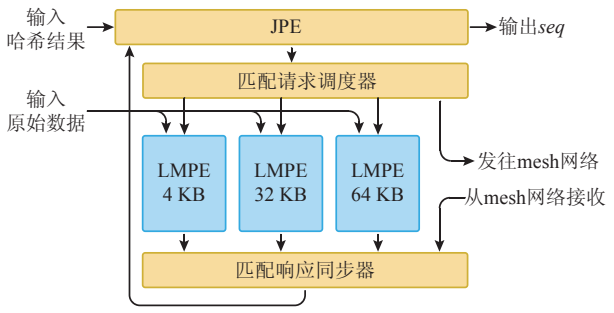


Fig. 15 Architecture of JMPC

图 15 JMPC 结构

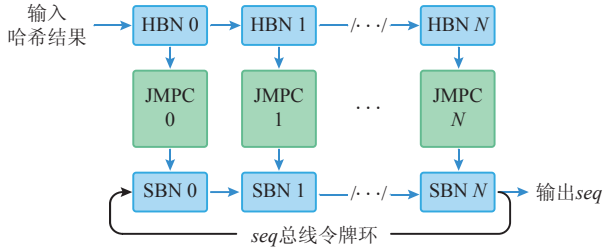


Fig. 16 Bus structures of JMPC Hash result input and *seq* output

图 16 JMPC 哈希结果输入和 *seq* 输出的总线结构

息，而每个 MatchJob 中包含的哈希结果数量固定，因此 HBN 可以根据哈希结果的头地址确定数据接收目标. 但 1 个 JMPC 处理单个 MatchJob 产生的 *seq* 数量不固定，为此 SBN 附带令牌环结构，在一个 JMPC 所有 *seq* 输出完成后将令牌传递给相邻的 JMPC，实现 *seq* 结果的顺序输出.

JMPC 与 SMPE 之间需要交换匹配请求和匹配响应，故建立 Mesh-2D 片上网络. 当 JMPC 中的 JPE 遇到需要扩展的哈希请求时，发往 LMPE 或 SMPE 处理. 对于单个 JMPC 而言，LMPE 是独享的，通过匹配请求调度器直接可达；SMPE 则被所有的 JMPC 共享，SMPE 处理的匹配请求和响应经 Mesh-2D 片上网络交换. JMPC 和 SMPE 在 Mesh-2D 网络中统一编址，JMPC 通过请求的历史地址确定目的 SMPE，SMPE 则通过匹配请求中的 tag 字段判断匹配响应的目的地.

本节的剩余部分将介绍 LazyHiveMatch 加速引擎中的关键细节.

6.1 MPE 架构设计及组织方式

当哈希结果中的元历史匹配长度达到元历史长度，则该哈希结果的实际匹配长度可能更长，LazyHiveMatch 引擎中的 LMPE 和 SMPE 负责上述情况的匹配扩展操作，二者统称为 MPE，具有统一的结构，如图 17 所示. 匹配扩展操作的执行部件包括保存滑动窗口的头/历史缓冲区、由异或逻辑组成的按字节比较器以及由优先编码器实现的长度编码模块，为了更好的时序表现，上述三者按照流水线形式组织.

LZ77 算法的数据依赖控制流特性与匹配操作的流水线设计矛盾，若等待上一个匹配操作完成再确定是否发射下一个操作，则流水线中会存在“空泡”. 为解决该矛盾，MPE 设计首先沿用了 BeeZip[16] 的猝发请求方法，即每个请求触发 4 次连续流水线输入，完成 4×*H* 字节长度的匹配，由猝发请求逻辑控制. 然后，增加了匹配请求表结构，允许多个匹配请求在流水线上交错执行，进一步提升利用率. 匹配请求表包括状态位（占用、等待、完成）、标签（*tag*），以及匹配请求状态信息（*headAddr*, *histAddr*, *ML*）. MPE 根据匹配请求表的状态位接收请求，发射到流水线以及发送响应. 匹配请求表的 tag 作为标识维护匹配响应与请求的对应关系，确定请求来源. 匹配请求状态信息根据匹配操作的进行而更新. 需要特别注意的是，若单次猝发操作仍不能满足一个匹配请求的扩展需求，该请求可在匹配请求表中驻留，等待发起下一组匹配. BeeZip2 的 MPE 单次猝发可完成 64 B 以内长度的匹配扩展操作，能够支持最长 1 024 B 的匹配.

LazyHiveMatch 引擎中匹配处理单元的组织方式是消减大滑动窗口开销的有力举措. 大滑动窗口 LZ77 的 MB 级历史滑动窗口是片上存储开销的主要来源. LazyHiveMatch 引擎将 1 MB 历史滑动窗口均分为 32

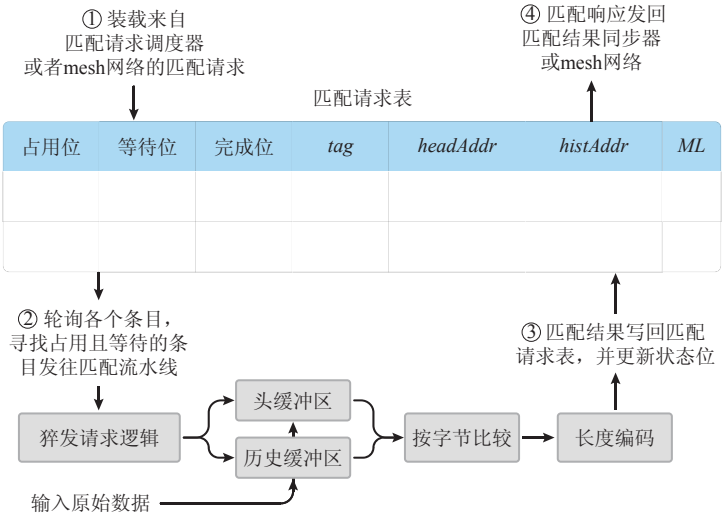
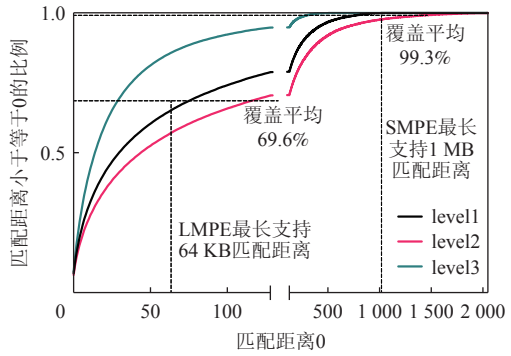


Fig. 17 Architecture and function of MPE

图 17 MPE 结构与功能

个片段保存在 SMPE 中，每个 SMPE 负责滑动窗口上 32 KB 大小区段的匹配扩展操作，各个 SMPE 之间保存的区段不重叠，输入数据独立、轮流地写入各个 SMPE. 来自多个 JMPC 的匹配请求和响应通过片上网络传递，充分挖掘有限存储资源上的并行性.

经由片上网络传递匹配请求和响应的方式会引入更高的延迟，对吞吐率表现有不利影响，为此，LazyHiveMatch 加速引擎进一步采用层次化的匹配处理单元组织方式，在 JMPC 中增加 LMPE，如图 15 所示. LMPE 处理数量占优的滑动窗口的近端匹配操作，本文研究过程中对 Zstandard 软件的匹配距离进行了分析，如图 18 所示，平均 69.6% 的匹配距离小于 64 KB，因此将 LMPE 的最大规模设置为 64 KB，剩余部分请求再交由 SMPE 处理. JMPC 中的 LMPE 采用异构规模[16]设计方案，除去最大 LMPE(64 KB)，还包含 2 个分别为 4 KB 和 32 KB 的 LMPE. 同一个 JMPC 中的 LMPE 之间构成副本关系，大规模的 LMPE 可以分摊小规模的 LMPE 负载. 例如，匹配距离为 16×10^3 的请求，可由 32 KB 或 64 KB 的 LMPE 处理. 该设计方式意味着 LazyHiveMatch 对于滑动窗口提供不同的并行度支持，匹配距离越小，可获取的并行度越高.



注：LMPE 支持的最长匹配距离为 64 KB，SMPE 支持的最长匹配距离为 1 MB

Fig. 18 Distribution of match offsets in the software implementation of Zstandard for three compression levels

图 18 Zstandard 软件实现中 3 个压缩等级上的匹配距离分布

6.2 JPE 及匹配请求调度器结构设计

LazyHiveMatch 算法中除匹配扩展外的其他操作在 JPE 中完成，结构设计如图 19 所示. JPE 主体包含 2 个表结构：Job 表与 Lazy 表. Job 表可装载一个完整的 MatchJob，即，$J\times H$ 个哈希结果. seqHead 以及 matchHead 指针保存在寄存器中，寻找匹配头逻辑移动上述指针指向有效哈希结果，支持并行查找，每次可探查 8 个条目. 当 matchHead 指向有效哈希结果时，从 matchHead 起 L 个哈希结果装入 Lazy 表. 倘若 Lazy 表中装入了需要进一步扩展的结果，则匹配扩展逻辑将请求发向匹配请求调度器，然后等待匹配结果响

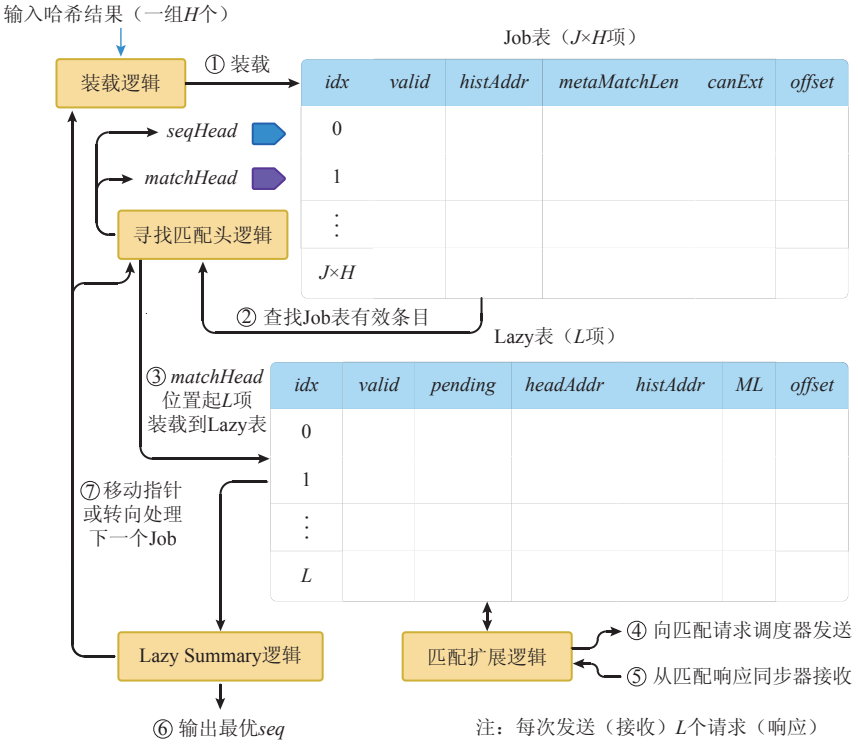


Fig. 19 Architecture and function of JPE

图 19 JPE 结构与功能

应同步器提供的结果(结合图 15). 当 Lazy 表中不包含需要扩展的结果或者已经收到所有需要的匹配响应时, Lazy Summary 逻辑按照算法规则选取最优结果并输出.

JPE 发出的匹配请求由匹配请求调度器发送到 LMPE 或 SMPE. 匹配请求调度器的架构如图 20 所示. JPE 发送请求到匹配请求调度器时根据 *offset* 计算请求的“路由位图”, 标识每个请求可以发送的请求通道, 请求通道直接连接 LMPE 或通过 Mesh-2D 网络连通 SMPE. 来自 JPE 的请求装入匹配请求调度器的匹配请求调度表, 匹配请求调度器根据路由位图和请求通道占用情况(由请求通道的就绪信号判断)实施固定优先级调度, 单周期内最多支持 4 个匹配请求的调度发出. 匹配请求调度器包含了对吞吐率的性能考虑, 并行调度减少延迟且与 JPE 协同, 优先调度请求到 LMPE 以减少 SMPE 的负载.

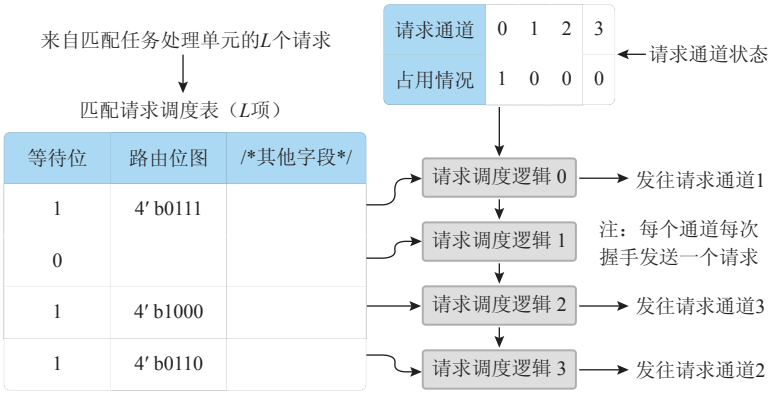


Fig. 20 Architecture design of match request scheduler

图 20 匹配请求调度器架构设计

7 实验评估与分析

本节首先介绍 BeeZip2 实验评估方法, 然后介绍 BeeZip2 性能、面积、功耗结果以及与基线设计的对比, 最终通过多个对比实验分析 BeeZip2 设计的有效性.

7.1 实验评估方法

1) 测试数据集

本文选择 Silesia 压缩语料库[33] 作为基准数据集, 该测试集在相关工作中使用广泛[10,14-17,20,24,30,34], 可支持公平的性能对比. 此外, Silesia 压缩语料库的大小(212 MB)比其他 2 个常用数据集 Calgary 和 Canterbury 语料库[35] 更大, 更适合于评估具有大滑动窗口的压缩算法.

2) 软件基线

本文采用 Zstandard v1.5.5 版本作为软件基线, 使用 GCC 10.2.1 编译, 软件基线的所有实验结果均在配置 2 颗 Intel Xeon Gold 6354(3.00 GHz, 每颗 CPU 具有 18 个物理核心)以及 256 GB 内存的服务器上测得, 使用 Zstandard 的内置基准测试功能(通过-b 参数调用)报告压缩比和吞吐率.

3) 加速器基线

本文尽最大努力将多种加速器基线纳入比较, 力求全面、公平的性能对比, 选择了采用不同设计模式的多种加速基线, 包括基于 HSR 和 PHT 的不同架构分类, 覆盖 ASIC, FPGA 不同实现方式, 以及来自开源社区和商业公司产品的架构设计. 此外, 本文也将在 NVIDIA A100 GPU 上进行数据压缩的 nvCOMP[36] 加速库纳入对比. 对于所有加速器基线, 本文优先采用相关工作文献或技术报告的结果. 相关工作[14,18,22] 未见公开报告的 Silesia 数据测试结果, 但提供了开源实现, 本文在其开源代码上进行了测试.

4) BeeZip2 的评估方法

本文研究过程中, 使用 VerilogHDL 语言对 BeeZip2 中的 MetaBeeHash 以及 LazyHiveMatch 引擎进行了 RTL 建模, 依托 Verilator 进行功能仿真, 验证了正确性(可对完整 Silesia 数据集进行压缩, 联合软件 Match-JobSerializer 和熵编码实现产生正确、可解压的 zst 格式数据). 使用商业逻辑综合工具、国产 28 nm 工艺库对计算和控制逻辑的时序、面积及功耗进行了评估; 联合使用 CACTI 7.0[37] 评估 SRAM 组件的相应指标. 以 1 GHz 时序为目标, 综合优化片上存储用量以及整体性能的考虑, 经过参数空间探索后, 本文最终确定的 BeeZip2 配置参数于表 1 中列出, 实验结果均在此参数组合下测得. BeeZip2 采用的吞吐率和压缩比评估方法与 BeeZip 一致, 将硬件 RTL 模型与 Zstandard

Table 1 Configuration Parameters of BeeZip2 Modeling

表 1 BeeZip2 建模配置参数

参数	文中符号	取值
并行输入字节数	H	16
哈希处理单元数量	H'	32
哈希表每行哈希槽数	R	8
哈希表总行数		32×10^3
滑动窗口大小/MB		1
元历史片段长度	M	13
并行 MatchJob 数量、JMPC 数量、SMPE 数量	N	32
MatchJob 长度	$J\times H$	4×16
惰性匹配长度	L	4
每个 JMPC 中的 3 个 LMPE 规格	4 KB，32 KB，64 KB	
单个 SMPE 规格/KB	32	
Fast，Balance，Better 模式下的 3 个 *HQT*	1，2，4	

软件的 externalProducer API 接口对接，软件部分提供按 8 MB 大小分片(滑动窗口大小的 8 倍)的 Silesia 数据集. 吞吐率根据 RTL 仿真周期数、逻辑综合时序结果计算；压缩比根据最终输出 zst 文件与原始输入文件大小计算. 为了验证可行性，本文研究过程中建立了 BeeZip2 的 FPGA 原型，并进行了软硬件联合测试. 图 21 展示了 BeeZip2 在 Xilinx Alveo U55c FPGA 上的布局布线情况. 需要说明，BeeZip2 面向 ASIC 工艺设计，FPGA 原型受限于主频，难以充分发挥性能优势，当前对 BeeZip2 性能的分析采用 ASIC 评估结果.

7.2 整体性能评估结果

BeeZip2 与测试基线的吞吐率和压缩比的测试结果和对比效果于表 2 和图 22 中呈现. 其中 BeeZip 和 BeeZip2 的 Fast、Balanced 和 Better 模式分别对应 *HQT* 参数设置为情况 1、2、4，该参数为运行时配置，可在相同的硬件结构上实现. 与 CPU 软件基线对比，

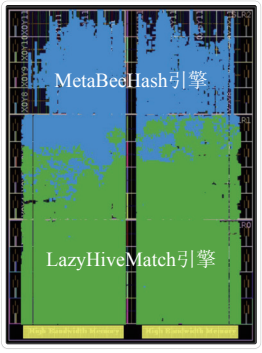


Fig. 21 Place and routing result of FPGA prototype for BeeZip2

图 21 BeeZip2 的 FPGA 原型布局布线结果

在具有相似压缩比(2.89 对比 2.88)时，BeeZip2 的吞吐率为单线程软件实现(zstd-T1-1)的 29.2 倍以及 36 线程软件实现(zstd-T36-1)的 3.35 倍. BeeZip2-Balanced 模式在压缩比相较 zstd-T36-2 高 2 个百分点的前提下，可实现 1.8 倍的吞吐率提升. 与软件对比的结果表明，BeeZip2 实现了“压缩比对标，吞吐率提高”的既定目标.

BeeZip2 的前代设计 BeeZip 是重要的加速器对比基线. 在 Fast 模式下 BeeZip2 的最高吞吐率可达到 13.13 GB/s，在该吞吐率下的压缩比为 2.89. 对比而言，BeeZip 的最高吞吐率为 10.42 GB/s，对应压缩比为 2.96. 尽管在 Fast 模式下 BeeZip2 的压缩比相较于 BeeZip 有 2.36% 的下降，但该压缩比仍然大于软件 zstd-1 的 2.88，符合 BeeZip2 的性能目标，可视为最高吞吐率相比 BeeZip 提升 1.26 倍. BeeZip2 和 BeeZip 的最高压缩比均为 3.14，达成该压缩比时 BeeZip2 的吞吐率略低 0.5 个百分点. 整体而言，BeeZip2 与 BeeZip 具有相似的吞吐率-压缩比权衡趋势，BeeZip2 进一步拓宽了支持的吞吐率范围. 此外，还需注意到 BeeZip2 的性能提升是建立在面积缩减基础上的，关于面积效率的

Table 2 Experimental Results of Throughput and Compression Ratio for BeeZip2 Compared with the Baselines

表 2 BeeZip2 与对比基线的吞吐率和压缩比实验结果

加速器/软件实现	吞吐率/(GB·s^{-1})	压缩比	加速器/软件实现	吞吐率/(GB·s^{-1})	压缩比	加速器/软件实现	吞吐率/(GB·s^{-1})	压缩比
BeeZip2-Fast	13.13	2.89	zstd-T4-1	1.66	2.88	IBM NXU	6.35	2.38
BeeZip2-Balanced	7.69	3.11	zstd-T4-2	1.35	3.05	VLDB23[22]	4.48	2.06
BeeZip2-Better	5.92	3.14	zstd-T4-3	0.97	3.18	MetaZip(*P*=8)	1.90	2.49
BeeZip-Fast	10.42	2.96	zstd-T36-1	3.92	2.88	MetaZip(*P*=16)	3.90	2.34
BeeZip-Balanced	7.73	3.11	zstd-T36-2	4.26	3.05	MetaZip(*P*=32)	7.80	2.06
BeeZip-Better	5.95	3.14	zstd-T36-3	2.65	3.18	MetaZip(*P*=64)	15.60	1.68
zstd-T1-1	0.45	2.88	nvCOMP	5.13	2.87	Project-Zipline	3.12	2.40
zstd-T1-2	0.38	3.05	QZSTD	11.15	2.76			
zstd-T1-3	0.27	3.19	CDPU	3.50	2.58			

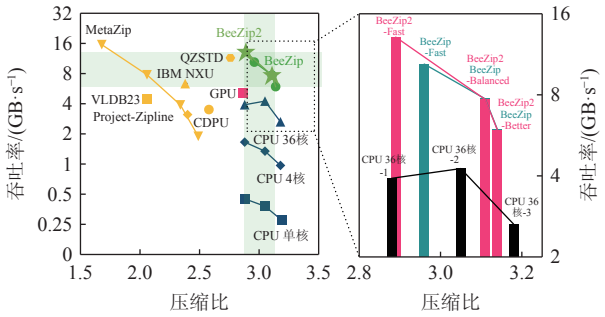


Fig. 22 Performance results comparison of BeeZip2 and the baselines

图 22 BeeZip2 与基线的性能结果对比

对比将于 7.3 节详细讨论.

对比其他加速器基线, GPU 基线具有与 BeeZip2-Fast 接近的压缩比, 但吞吐率仅为 BeeZip2-Fast 的 39.1%. BeeZip2-Better 的吞吐率略高于 GPU 基线, 同时具有 9 个百分点的压缩比优势. 综合考虑压缩比和吞吐率表现, QZSTD 是 BeeZip2 除 BeeZip 外的强劲竞争对手, 但 BeeZip2-Fast 相较 QZSTD 仍然具有 4.7 个百分点的压缩比优势. MetaZip 在所有加速器基线中具有最高的吞吐率表现, 但 BeeZip2 相较 MetaZip 具有明显的压缩比优势.

7.3 面积功耗评估结果

由于 BeeZip 是本文的重要对比基线, 且其他相关工作未提供可直接对比的面积功耗结果, 本节重点介绍与 BeeZip 的对比结果. 为了实现公平的面积功耗评估, 本文研究采用相同的逻辑综合方法, 对 BeeZip 开源实现进行了评估, 表 3 列出了 BeeZip 与 BeeZip2 的面积功耗结果对比.

Table 3 Area and Power Estimation Resuls of BeeZip2

表 3 BeeZip2 面积和功耗评估结果

加速器/软件实现	分类	面积/mm^2	功耗/W		
			静态	动态	总计
BeeZip	逻辑	1.31	0.04	1.10	1.14
	存储	27.7	1.74	6.19	7.94
	总计	29.0	1.78	7.29	9.08
BeeZip2	逻辑	2.50	0.08	2.12	2.20
	存储	15.6	1.10	6.80	7.90
	总计	18.1	1.18	8.92	10.1

相比于 BeeZip, BeeZip2 的总面积减少了 37.6%. 若按二者 Fast 级别的吞吐率计算单位面积吞吐率 (GB/s·mm^{-2})则 BeeZip2 相比 BeeZip 提升 2.02 倍. BeeZip2 的面积收益来自于共享匹配单元设计对于滑动窗口的消除. 从 SRAM 用量角度分析, BeeZip2 与 BeeZip 用于保存哈希表的 SRAM 用量均为 4 MB, 保存滑动窗口所需 SRAM 用量分别为 4.375 MB 和 9.02 MB. 共享匹配单元设计使得 LazyHiveMatch 引擎相比 HiveMatch 引擎 SRAM 用量减少了 51.5%, 促成了面积效率优势.

在功耗方面, 由于 BeeZip2 的调度结构更加复杂, 且分片设计增加了 SRAM 单元数量, 因此相较于 BeeZip 功耗有所增加. 但从单位功耗吞吐率(GB/s·W^{-1})角度分析, BeeZip2 相较于 BeeZip 实现了 1.13 倍提升.

7.4 MetaBeeHash 加速引擎设计有效性实验

将元历史匹配方法引入大滑动窗口 LZ77 加速器设计是 MetaBeeHash 引擎的重要创新点. 本节介绍围绕元历史匹配设计的 2 个对照实验, 论证该设计的有效性. 第 1 个对照实验将元历史长度 M 由 13 B 缩短至 4 B; 第 2 个对照实验只使用元历史匹配结果, 后续不再进行扩展. 这 2 个对照实验的结果于图 23 中展示.

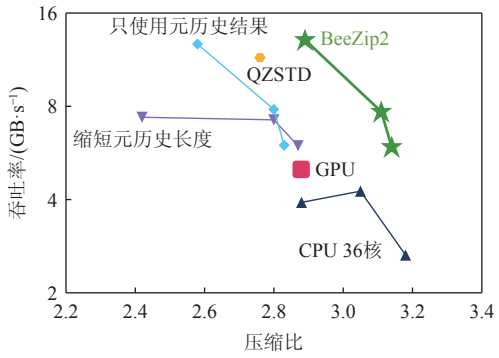


Fig. 23 Experimental results for effectiveness of MetaBeeHash accelerated engine design

图 23 MetaBeeHash 加速引擎设计有效性实验结果

缩短元历史匹配长度后, BeeZip2 的吞吐率和压缩比均出现明显下降. 当元历史长度由默认配置长度缩减至 4 B 后, 元历史匹配仅能过滤哈希值冲突, MetaBeeHash 引擎不能提供规整、有效的并行, LazyHiveMatch 引擎的负载增大, 控制流数据依赖现象开始对性能产生不利影响. 此外 MetaBeeHash 无法进行有效的哈希结果归并择优, 故压缩比也受到影响.

只使用元历史匹配实验结果验证了该设计的另一个极端情况, 即结果显示 BeeZip2 的压缩比优势被破坏. 该现象的成因是元历史匹配仅能提供有限的匹配长度, 论证了元历史机制与后续匹配扩展操作结合的必要性.

7.5 LazyHiveMatch 加速引擎设计有效性实验

LazyHiveMatch 引擎的主要创新点之一是 SMPE 的组织方式, 本节引入不采用共享匹配单元的架构

对比实验来分析该设计的有效性, 测得的吞吐率和压缩比结果如图 24 所示. 不采用 SMPE 时, 为了支持完整的大滑动窗口匹配扩展需求, 需要在每个 JMPC 中增加一个保存完整滑动窗口的 LMPE, 这将引入额外 32 MB 的片上存储. 理论上, LMPE 具有更低的延迟, 应当有明显的吞吐率提升, 但结果显示该提升不显著. 该实验可证明 SMPE 设计不会对吞吐率带来明显影响, 结合片上存储资源开销, 可论证 SMPE 设计是解决大滑动窗口开销问题的有效手段.

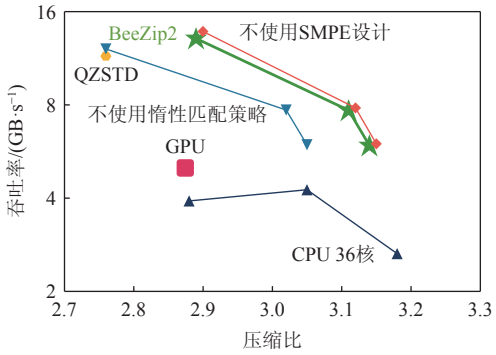


Fig. 24 Experimental results of effectiveness for LazyHiveMatch accelerated engine design

图 24 LazyHiveMatch 加速引擎设计有效性实验结果

惰性匹配策略是 LazyHiveMatch 引擎的另一个创新点, 将惰性匹配长度 L 设置为 1 可构造对该策略的消融实验, 结果于图 24 中展示. 该实验表明, 惰性匹配策略可在不明显影响吞吐率的前提下, 实现压缩比的有效提升, 是支撑 BeeZip2 性能优势的重要设计.

8 相关工作

8.1 Zstandard 压缩加速器

文献 [14, 16, 23, 34] 中提出了多种支持 Zstandard 的加速器设计. BeeZip[16] 已在前文有所介绍并参与了性能对比, 此处不再赘述. CDPU[14] 是支持 Zstandard 算法的压缩加速器架构, 与 RISC-V 处理器通过扩展指令紧耦合集成, 依托 FireSim 框架[38] 实现. CDPU 重点关注压缩加速器与现有通用处理器架构的集成方法, 以及解压缩部分的加速, 并未提及压缩加速器微架构的改进. CDPU 加速 Zstandard 的性能结果已纳入与 BeeZip2 的对比中, 由于 CDPU 未提供完整的 Silesia 测试结果, 本文使用的测试结果依据其开源代码在 Xilinx Alveo U280 FPGA 加速卡[39] 上测得. 文献 [23] 提出了一种针对高频交易数据优化和基于 FPGA 的数据压缩加速器来支持 Zstandard 算法, 在 Xilinx Alveo U200 FPGA 加速器卡上实现了 8.6 GB/s 的吞吐量. 由于缺少 Silesia 的测试结果以及开源实现, 该研究工作未直接参与和 BeeZip2 的对比, 但是, 文献 [23] 研究支持的滑动窗口大小仅为 64 KB, 且包含多个副本, 可合理推断其在支持更大滑动窗口时会产生较大的资源开销. Xilinx Vitis 数据压缩库[40] 可实现 1.17 GB/s 和 2.68 倍的 Zstandard 压缩, 但块大小限制为 32 KB, 支持的滑动窗口大小有限, 性能结果明显低于 BeeZip2. nvCOMP 是 NVIDIA GPU 上运行的高速数据压缩库, 支持 Zstandard 算法, 已加入与 BeeZip2 的对比. QZSTD 是基于 Intel QAT 技术实现的硬件加速 Zstandard 压缩工具. Intel QAT 是一系列包括数据压缩和加解密在内的加速器集合[41], 可作为 PCIe 板卡或主板芯片组形式与系统集成. 目前尚无 Intel QAT 详细的架构设计信息, QZSTD 公开的测试结果已加入与 BeeZip2 的对比.

8.2 其他数据压缩加速器

除 Zstandard 之外, DEFLATE 算法是数据压缩加速器的常见目标. IBM NXU[10] 是集成于 IBM POWER 9 和 z15 处理器中的数据压缩加速器, 支持 DEFLATE 算法. IBM NXU 的主要架构创新是 NearCAM, 因此属于 HSR 类加速器, 在支持大滑动窗口时可扩展性有限, 在 Silesia 上的测试结果已加入与 BeeZip2 的对比. Project-Zipline 是微软主导的开源数据压缩加速器实现, 支持 DEFLATE 和 XP10 算法, 结合对开源代码的分析, 本文研究认为其属于 HSR 类别, 本文研究测试时采用了其最大滑动窗口(64 KB)支持配置, 结果已加入与 BeeZip2 的对比.

文献 [15, 17, 22, 24] 提出了多种基于 PHT 架构的设计, 其中 MetaZip[15] 提出元历史匹配方法并实现 15.6 GB/s 的吞吐率. MetaZip 的实验结果在表 2 中已列出. 上述基于 PHT 架构的实现均未考虑数据依赖控制流问题的影响, 无法实现高压缩比.

WaveSZ[42] 提出了加速科学计算数据压缩算法 SZ 的软硬件协同设计方案, 其中包含了在 FPGA 上并行计算相邻数值相似性的结构. 与 LZ77 算法按字节查找冗余的方式不同, SZ 需要分析浮点数之间的关系, 是针对科学计算中矩阵、张量数据的专用压缩算法. 不过, SZ 的后处理流程中也包含无损压缩算法(如 Zstandard), 结合 WaveSZ 的吞吐率测试结果(3~4 GB/s)分析, BeeZip2 可以作为无损压缩组件与之整合.

9 总结与展望

本文提出了一种新型数据压缩加速架构 BeeZip2.

该架构采用“算法-架构”跨层优化方法，应对 Zstandard 工具中“大滑动窗口”LZ77 片上存储开销大和控制流数据依赖带来的挑战. 在算法层面，本文提出了包含元历史匹配的 MetaBeeHash 和 LazyHiveMatch 算法，引入了元历史匹配机制和简易惰性匹配策略；在架构层面，设计了支持上述算法改进的 MetaBeeHash 和 LazyHiveMatch 加速引擎，并提出了共享匹配处理单元架构，有效降低了片上存储开销. 实验结果表明，BeeZip2 在压缩比达到软件标准的同时，吞吐率显著提升，相较于单核和 36 核软件实现分别提升了 29.2 倍和 3.35 倍，与 BeeZip 基线加速器相比，最高吞吐率提升 1.26 倍，单位面积吞吐率提升 2.02 倍. 元历史匹配机制、共享匹配单元设计、简易惰性匹配策略的有效性也通过对比实验加以证明.

当下 BeeZip2 架构设计关注大滑动窗口 LZ77 算法的加速，构建端到端的数据压缩加速器还需将熵编码以及系统集成方式纳入考虑. 展望未来，将 BeeZip2 部署到 FPGA 平台是一种实现高效数据压缩系统的可行方案. 然而，由于 FPGA 平台的工作主频相较于 ASIC 实现更低，在部署过程中，需要优化流水线宽度来弥补性能差距. 具体优化举措包括增加哈希调度结构位宽、扩展 LazyHiveMatch 引擎总线位宽，以及扩展 MPE 的读写位宽等.

作者贡献声明：高睿昊提出方案、完成实验并撰写论文；史舜晨协助完成实验中的硬件综合评估部分、整理格式与排版；李雪琦指导论文思路与写作；谭光明提供研究思路并给予指导意见.

参 考 文 献

[1] Li Shuang, Puig X, Paxton C, et al. Pre-trained language models for interactive decision-making[J]. Advances in Neural Information Processing Systems, 2022, 35: 31199−31212

[2] Touvron H, Lavril T, Izacard G, et al. LLaMA: Open and efficient foundation language models[J]. arXiv preprint arXiv: 2302.13971, 2023

[3] Liu Haifeng, Zheng Long, Huang Yu, et al. Enabling efficient large recommendation model training with near CXL memory processing[C]//Proc of 2024 ACM/IEEE 51st Annual Int Symp on Computer Architecture (ISCA). Piscataway, NJ: IEEE, 2024: 382−395

[4] Choi Y, Kim J, Rhu M. ElasticRec: A microservice-based model serving architecture enabling elastic resource scaling for recommendation models[C]//Proc of 2024 ACM/IEEE 51st Annual Int Symp on Computer Architecture (ISCA). Piscataway, NJ: IEEE, 2024: 410−423

[5] Lee Y, Kim H, Rhu M. PreSto: An in-storage data preprocessing system for training recommendation models[C]//Proc of 2024 ACM/IEEE 51st Annual Int Symp on Computer Architecture (ISCA). Pissataway, NJ: IEEE, 2024: 340−353

[6] Abramson J, Adler J, Dunger J, et al. Accurate structure prediction of biomolecular interactions with AlphaFold 3[J]. Nature, 2024, 630(8016): 493−500

[7] Batatia I, Kovacs D P, Simm G, et al. MACE: Higher order equivariant message passing neural networks for fast and accurate force fields[J]. Advances in neural information processing systems, 2022, 35: 11423−11436

[8] Deng Bowen, Zhong Peichen, Jun K J, et al. CHGNet as a pretrained universal neural network potential for charge-informed atomistic modelling[J]. Nature Machine Intelligence, 2023, 5(9): 1031−1041

[9] Sayood K. Introduction to Data Compression[M]. Cambridge, MA: Morgan Kaufmann, 2017

[10] Abali B, Blaner B, Reilly J, et al. Data compression accelerator on IBM POWER9 and z15 processors: Industrial product[C]//Proc of 2020 ACM/IEEE 47th Annual Int Symp on Computer Architecture (ISCA). Piscataway, NJ: IEEE, 2020: 1−14

[11] Deutsch P. GZIP file format specification version 4.3: RFC1952[R/OL]. RFC Editor, 1996: RFC1952[2021-12-10]. https://www.rfc-editor.org/info/rfc1952. DOI:10.17487/rfc1952

[12] Deutsch P. DEFLATE compressed data format specification version 1.3: RFC1951[R/OL]. RFC Editor, 1996: RFC1951[2021-12-12]. https://www.rfc-editor.org/info/rfc1951. DOI:10.17487/rfc1951

[13] Collet Y. Zstandard compression and the ‘application/zstd’ media type[EB/OL]. RFC Editor: United States, 2021[2024-12-24]. https://datatracker.ietf.org/doc/html/rfc8878

[14] Karandikar S, Udipi A N, Choi J, et al. CDPU: Co-designing compression and decompression processing units for hyperscale systems[C]//Proc of the 50th Annual Int Symp on Computer Architecture(ISCA). New York, NY, USA: Association for Computing Machinery, 2023: 1−17

[15] Gao Ruihao, Li Xueqi, Li Yewen, et al. MetaZip: a high-throughput and efficient accelerator for DEFLATE[C]//Proc of the 59th ACM/IEEE Design Automation Conf. San Francisco California: ACM, 2022: 319−324

[16] Gao Ruihao, Li Zhichun, Tan Guangming, et al. BeeZip: towards an organized and scalable architecture for data compression[C]//Proc of the 29th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems, Volume 3. La Jolla CA USA: ACM, 2024: 133−148

[17] Fowers J, Kim J Y, Burger D, et al. A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs[C]//Proc of the 2015 IEEE 23rd Annual Int Symp on Field-Programmable Custom Computing Machines. Vancouver, BC, Canada: IEEE, 2015: 52−59

[18] Rajeev S. opencomputeproject/Project-Zipline[CP/OL]. 2024[2024-12-24]. https://github.com/opencomputeproject/Project-Zipline

[19] Ziv J, Lempel A. A universal algorithm for sequential data compression[J]. IEEE Trans on Information Theory, 1977, 23(3): 337−343

[20] Collet Y. Zstandard: Real-time data compression algorithm[EB/OL]. [2024-12-25]. https://facebook.github.io/zstd/

[21] Adler M. A massively spiffy yet delicately unobtrusive compression library[EB/OL]. [2024-12-25]. https://www.zlib.net/

[22] Chiosa M, Maschi F, Müller I, et al. Hardware acceleration of compression and encryption in SAP HANA[J]. Proc of the VLDB Endowment, 2022, 15(12): 3277−3291

[23] Chen J, Daverveldt M, Al-Ars Z. FPGA acceleration of zstd compression algorithm[C]//Proc of 2021 IEEE Int Parallel and Distributed Processing Symp Workshops (IPDPSW). 2021: 188−191

[24] Qiao Weikang, Du Jieqiong, Fang Zhenman, et al. High-throughput lossless compression on tightly coupled CPU-FPGA platforms[C]// Proc of 2018 IEEE 26th Annual Int Symp on Field-Programmable Custom Computing Machines (FCCM). Boulder, CO, USA: IEEE, 2018: 37−44

[25] Terrell N. Zstd in the Linux kernel[EB/OL]. [2024-12-25]. https://github.com/facebook/zstd/blob/dev/contrib/linux-kernel/README.md

[26] Santosh V. Compression methods in MongoDB: Snappy vs. Zstd[EB/OL]. [2024-12-25]. https://www.percona.com/blog/compression-methods-in-mongodb-snappy-vs-zstd/

[27] Liang Xin, Di Sheng, Tao Dingwen, et al. Error-controlled lossy compression optimized for high compression ratios of scientific datasets[C]//2018 IEEE Int Conf on Big Data (Big Data). Seattle, WA, USA: IEEE, 2018: 438−447

[28] Zhao Kai, Di Sheng, Liang Xin, et al. Significantly improving lossy compression for HPC datasets with second-order prediction and parameter optimization[C]//Proc of the 29th Int Symp on High-Performance Parallel and Distributed Computing. Stockholm Sweden: ACM, 2020: 89−100

[29] Liang Xin, Zhao Kai, Di Sheng, et al. SZ3: A modular framework for composing prediction-based error-bounded lossy compressors[J]. IEEE Trans on Big Data, 2023, 9(2): 485−498

[30] Hennessy J L, Patterson D A. A new golden age for computer architecture[J]. Communications of the ACM, 2019, 62(2): 48−60

[31] Dally W J, Turakhia Y, Han Song. Domain-specific hardware accelerators[J]. Communications of the ACM, 2020, 63(7): 48−57

[32] Skibinski P. inikep/lzbench[CP/OL]. 2015 (2024-12-25)[2024-12-25]. https://github.com/inikep/lzbench

[33] Deorowicz S. Silesia compression corpus[EB/OL]. 2003[2021-12-13]. http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia

[34] Will B, Qian D, Khade A, et al. Intel® QuickAssist technology zstandard plugin, an external sequence producer for zstandard[EB/OL]. (2023-08-16)[2024-12-26]. https://community.intel.com/t5/Blogs/Tech-Innovation/Artificial-Intelligence-AI/Intel-QuickAssist-Technology-Zstandard-Plugin-an-External/post/1509818

[35] Powell M. Canterbury and Calgary compression corpora[EB/OL]. 1997[2021-12-13]. https://corpus.canterbury.ac.nz/descriptions/

[36] nvCOMP: High-speed data compression using NVIDIA GPUs[EB/OL]. [2024-12-26]. https://developer.nvidia.com/nvcomp

[37] Balasubramonian R, Kahng A B, Muralimanohar N, et al. CACTI 7.0: New tools for Interconnect exploration in Innovative Off-Chip Memories[J]. ACM Trans on Architecture and Code Optimization, 2017, 14(2): 1−25

[38] Karandikar S, Mao H, Kim D, et al. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud[C]//Proc of 2018 ACM/IEEE 45th Annual Int Symp on Computer Architecture (ISCA). Piscataway, NJ: IEEE, 2018: 29−42

[39] AMD. AMD Alveo U280: Product brief[EB/OL]. 2020[2024-12-26]. https: //www.xilinx.com/publications/product-briefs/alveo-u280-product-brief.pdf

[40] AMD. Vitis data compression library 2022.1[EB/OL]. 2022[2024-12-26]. https://xilinx.github.io/Vitis_Libraries/data_compression/2022.1/benchmark.html

[41] Intel. Intel QAT: Performance, Scale, and Efficiency[EB/OL]. 2020[2024-12-26]. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html

[42] Tian Jiannan, Di Sheng, Zhang Chengming, et al. waveSZ: a hardware-algorithm co-design of efficient lossy compression for scientific data[C]//Proc of the 25th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. San Diego California: ACM, 2020: 74−88

Gao Ruihao, born in 1998. PhD candidate. His main research interests include domain-specific architecture, data compression algorithm, and RTL simulation acceleration.

高睿昊, 1998 年生. 博士研究生. 主要研究方向为领域定制架构、数据压缩算法、RTL 仿真加速.

Shi Shunchen, born in 2000. PhD candidate. His main research interests include domain-specific hardware accelerators, processing in memory, and computer architecture.

史舜晨, 2000 年生. 博士研究生. 主要研究方向为领域定制硬件加速器、存算一体架构、计算机体系结构.

Li Xueqi, born in 1991. PhD, associative professor. His main research interests include domain-specific hardware accelerators, processing in memory, and system-architecture cross-layer optimization.

李雪琦, 1991 年生, 博士, 副研究员. 主要研究方向为领域定制硬件加速器、存算一体架构、系统架构跨层优化.

Tan Guangming, born in 1980. PhD, professor. His main research interests include parallel programming and algorithms design, domain-specific architecture, and bioinformatics.

谭光明, 1980 年生. 博士, 研究员. 主要研究方向为并行程序与算法设计、领域专用体系结构、生物信息学.