计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(5): 1123−1135, 2025

一种面向纠删码的存储库优化方法

谢汶兵　关睿雪　张艺鸣　李佳梅　王　俊
（无锡先进技术研究院　江苏无锡　214126）
（xiewb@wxiat.com）

Efficient Optimization of Erasure Coding for Storage Library

Xie Wenbing, Guan Ruixue, Zhang Yiming, Li Jiamei, and Wang Jun
（*Wuxi Institute of Advanced Technology*, *Wuxi*, *Jiangsu* 214126）

Abstract　In the information stage, the importance of data storage lies in ensuring the reliability, consistency, security, and real-time accessibility of information. Erasure codes (EC) play a crucial role in data storage systems due to their ability to minimize storage overhead and handle multiple component failures. However, the process of encoding and decoding EC involves intensive computation, impacting storage system efficiency. This paper focuses on optimizing EC, with a special emphasis on the Galois field (GF) multiplication within multi-layer loops, a time-consuming aspect of EC. We first evaluate the pros and cons of three methods for GF multiplication calculation: the log table searching method, the complete multiplication table searching method, and the shift decomposition method. Subsequently, a 4 b splitting (SP) method is proposed to reduce memory access overhead during table searching in GF(2^8). We delve into the SP's analysis and leverage the 64 b modern processor architecture and vector instruction set characteristics to introduce data-level parallelism in multi-layer loops. This involves amplifying data access granularity and implementing single instruction multiple data (SIMD) vectorization. Based on the open-source Intel storage acceleration library (ISA-L), all optimization methods are implemented and tested on Sunway processor and x86 processor. The experimental results show the effectiveness of proposed optimization in improving EC performance across different data scalability scenarios. When compared with the original ISA-L, our optimizations exhibit an average performance speedup of 3.28× on Sunway, 2.36× on x86.

Key words　erasure codes; Galois field multiplication; 4b splitting method; data access granularity; SIMD vectorization; data lookup table

摘　要　信息时代，数据存储的可靠性、一致性、安全性和实时性至关重要. 纠删码（erasure code, EC）在允许多个存储设备发生故障的同时保证最低的存储开销，被大量应用在数据存储领域. 纠删码的编码与解码运算具有计算密集的特征，其性能高低直接影响存储系统的使用效率. 作为编码和解码运算中最耗时的部分，多层循环包裹的伽罗华域乘法计算是纠删码优化的一个焦点. 首先分析了伽罗华域乘法计算的查表方法中常用的log查表法（LT）、完全乘法查表法（MT）、移位分解法（SH）的优劣势，然后对已有的伽罗华域GF(2^8)查表方法进行了优化，提出4 b分割法以大幅减少查表开销. 在此基础上，利用64位现代处理器体系结构特点，从数据访问粒度扩展和单指令多数据（single instruction multiple data, SIMD）向量化利用实现数据级并行化2个角度优化了多层循环中的数据级访问粒度，提高了编码与解码的运算性能. 基于开源存储加速库（Intel storage acceleration library, ISA-L）在申威平台和x86平台上实现和验证了上述优化方法的有效性. 结果表明所提优化方法在不同数据规模下均有加速效果，申威平台与优化前

收稿日期：2024-02-03；**修回日期**：2024-10-11
通信作者：关睿雪（guanruixuely@126.com）

相比平均性能加速比为 3.28 倍, x86 平台与优化前相比平均性能加速比为 2.36 倍.

关键词 纠删码; 伽罗华域乘法; 4 b 分割法; 数据访问粒度; SIMD 向量化; 数据查表

中图法分类号 TP319

DOI: 10.7544/issn1000-1239.202440091 **CSTR:** 32373.14.issn1000-1239.202440091

当今社会各行各业都在广泛使用计算机和互联网, 产生的巨量数据在接入云或互联网后被存入数据中心, 其存储的数据可靠性、一致性、安全性和实时性至关重要, 其中可靠性是数据存储的关键. 传统的 RAID5 在 $k(k>1)$ 个数据块上保存数据, 任意 $(k-1)$ 个数据块都有完整的数据, 只允许单个数据块发生故障. 然而随着 k 的数据量增加, 多个数据块发生故障的概率随之增长, RAID5 不再适用. 纠删码(erasure code, EC)能够在多个数据块发生故障时恢复出原始数据, 提高了系统容错性[1]. 在允许相同数量级的数据块发生故障的算法中, 纠删码具有低存储开销的优势[2], 被广泛应用于各种数据中心的存储系统, 包括磁盘阵列系统[3]、点对点存储系统[4]、分布式存储系统[5-6]、云存储系统[7] 等.

虽然纠删码具有高可靠性和低存储开销的优势, 但其在编码和解码过程会引发大量密集型计算, 这不仅花费过长的运算时间, 也对处理器的运算性能和访存带宽带来巨大挑战[8-9]. 纠删码的运算通常是在有限域内进行, 常用的有限域是伽罗华域(Galois Field)GF(2^w). 伽罗华域内的运算包括加法和乘法, 其中加法运算等同于异或操作, 乘法运算等同于多项式运算. 乘法运算效率是伽罗华域运算乃至纠删码运行性能的一个重要的瓶颈点[10-12], 现有的伽罗华域乘法运算操作主要分为基于异或操作和基于查表操作 2 类.

基于异或操作方法将伽罗华域的乘法展开为 2 个或多个异或操作, 该方法由 Blomer 等人[12] 提出. Detchart 等人[10] 在文献 [12] 方法的基础上使用多项式环变换对异或操作的展开过程进行了优化, 减少了展开后的总操作数. Luo 等人[13] 分析了异或操作的执行顺序对执行效率的影响, 提出数据词和数据包引导的调度算法以提升异或操作的缓存命中率, 进而提高纠删码的运算性能. Uezato[11] 对冗余的异或操作进行合并和压缩, 改进了缓存命中率, 减少了不必要的访存操作. Zhou 等人[2] 对以上各类方法进行了总结, 将异或操作的合理调度、缓存命中的改善与数据级并行相结合, 旨在最大程度上优化纠删码的运算性能. 基于异或操作的方法能够将复杂的伽罗华域乘法转化为便于计算机实现的简单异或操作, 但依旧存在 3 个关键挑战:

1)优化效率依赖于复杂的冗余合并算法和调度优化算法, 优化难度大;

2)异或操作之间存在强数据依赖关系, 不利于数据级并行化处理;

3)伽罗华域 GF(2^w) 中位宽为 w 位的数据需要展开成维数为 $w\times w$ 的矩阵, 增加了数据运算的空间复杂度.

基于查表操作的方法避免了异或操作所面临的挑战, 提供了 log 查表法(logarithm table, LT)和完全乘法查表法(multiplication table, MT)两种方法来实现伽罗华域的乘法运算. LT 方法利用乘法的对数表达将伽罗华域乘法转化为对数查表和指数查表, 只需 3 次查表操作就可等价实现伽罗华域乘法运算. MT 方法则是将所有可能的伽罗华域乘法运算结果保存到一个数据表中, 使用时只需 1 次查表操作. 此外, 基于查表操作的方法可以最大程度地利用数据级并行化来优化有限域乘法运算效率, 提升纠删码运算性能[14-16]. 基于查表操作的方法具有实现简单、运行效率高的优点, 但也存在 3 个挑战.

1)查表过程引入的访存开销大. 查表避免了过度消耗处理器的计算资源, 但强依赖于表自身的容量大小和处理器的访存带宽.

2)按字节为单位的细粒度低效数据处理. 当有限域的数据位宽小于 64 b, 即 GF(2^w) 中的 $w<64$ 时, 在数据存取和数据运算过程中对现代处理器的硬件优势发挥不足.

3)利用体系结构相关的单指令多数据(single instruction multiple data, SIMD)指令加速优化设计的研究较少.

针对以上挑战, 本文基于查表操作的伽罗华域乘法提出了 3 种优化思路.

1)降低运算和查表开销. 设计 4 b 分割法, 组合采用查表方法和异或操作. 减少了查表空间的同时降低了运算数据之间的依赖程度, 为数据并行化提供基础.

2)提升数据处理粒度. 优化了 GF(2^8) 查表方法

的数据处理粒度，将循环内数据访问和数据运算的粒度从字节提升到长字，避免了重复查表操作.

3）支持数据级并行处理. 利用 SIMD 指令向量化内存存取、与/异或、移位和查表操作，充分发挥现代处理器算力优势.

本文基于 Intel 存储加速库（Intel storage acceleration library, ISA-L）[17] 实现了上述优化，并在申威平台和 x86 平台进行了代码实现. 基于不同数据规模分析和验证了优化效果.

1 相关背景

1.1 纠删码算法原理

对 k 个同样大小的数据块，添加 m 个校验块，保证 $(k+m)$ 个数据中任意丢失 $n(n\leqslant m)$ 个数据块或校验块时都能恢复到原始数据. 其中，由 k 个数据块计算得出 m 个校验块的过程称为数据编码，由 $(k+m-n)$ 个数据块和校验块恢复丢失数据的过程称为数据解码. 纠删码解码和解码过程如图 1 所示. 相较“1 份数据存储多个副本”的方式，纠删码算法在保证高可靠性的同时，减少了数据存储空间. 相较 RAID5，纠删码允许用户按照自身需求设定冗余信息的个数，提高了容错性.

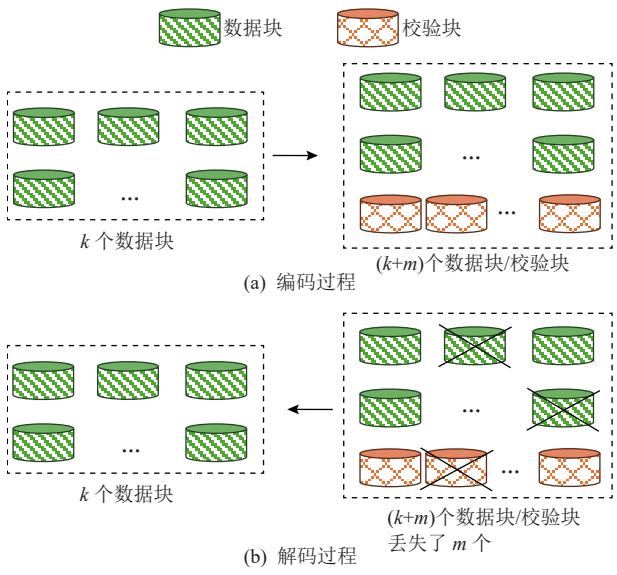


Fig. 1 Encoding and decoding of erasure codes

图 1 纠删码的编码与解码

纠删码算法编码过程的运算示意图如图 2 所示，k 个数据矩阵 $\boldsymbol{D}$ 左乘编码矩阵 $\boldsymbol{M}$ 后生成数据和校验矩阵 $\boldsymbol{E}$，表示为

$$\boldsymbol{MD}=\boldsymbol{E}, \tag{1}$$

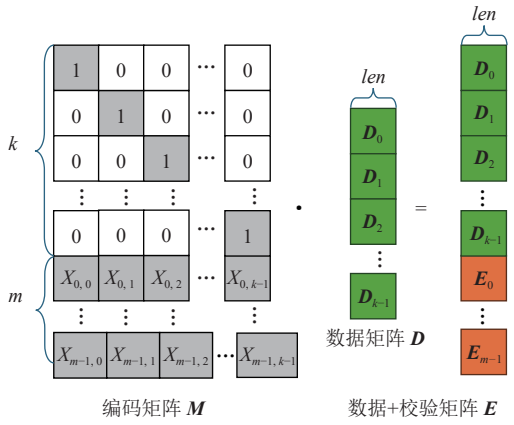


Fig. 2 Matrix representation of encoding

图 2 编码过程的矩阵表示

其中 $\boldsymbol{M}$ 为 $(k+m)\times k$ 维的编码矩阵，$\boldsymbol{D}$ 为 $k\times len$ 维的数据矩阵，$\boldsymbol{E}$ 为数据块+校验块组成的 $(k+m)\times len$ 维矩阵. 编码矩阵 $\boldsymbol{M}$ 的前 k 行组成一个 $(k\times k)$ 的单位矩阵，其后 m 行取自范德蒙矩阵或柯西矩阵. 编码矩阵 $\boldsymbol{M}$ 具备如下特性：任意 k 行线性不相关，即任意 k 行组成的方阵是可逆的. 假设有 m 个数据块/校验块丢失，形成了新的数据+校验矩阵 $\hat{\boldsymbol{E}}$，将 $\boldsymbol{M}$ 的对应行去掉后新组成矩阵 $\hat{\boldsymbol{M}}$，此时编码过程可表示为 $\hat{\boldsymbol{M}}\boldsymbol{D}=\hat{\boldsymbol{E}}$. 编码矩阵的特性决定了矩阵 $\hat{\boldsymbol{M}}$ 是可逆的 k 阶方阵，原始数据块 $\boldsymbol{D}$ 可以由 $(\hat{\boldsymbol{M}})^{-1}\hat{\boldsymbol{E}}$ 计算得到，丢失的数据得以恢复. 因此，纠删码算法解码过程可表示为

$$(\hat{\boldsymbol{M}})^{-1}\hat{\boldsymbol{E}}=\boldsymbol{D}. \tag{2}$$

本质上，纠删码算法的编码与解码过程均为矩阵相乘，唯一不同的是编码使用了编码矩阵，而解码使用了编码矩阵的逆矩阵.

实际应用中，数据块通常以 8 b、16 b、32 b、64 b 四种位宽存储在计算机中，数据块直接左乘编码矩阵很容易导致校验数据的溢出. 为了将校验数据控制在特定数值范围，纠删码算法采用了伽罗华域的代数运算. 伽罗华域的加法运算（记为$\oplus$）等同于异或操作，伽罗华域的乘法运算（记为$\otimes$）等同于与操作. $GF(2^w)$ 表示位宽为 w 位的伽罗华域，数值范围为 $[0, 2^w-1]$，w 通常取值为 8 b、16 b、32 b、64 b 等. $GF(2^w)$ 的数值可看作是一个多项式，数值的每个对应位数是多项式的系数，比如 $GF(2^8)$ 内的数值 a=10 011 101，数据块矩阵元素 a 的多项式形式可写为

$$a=1+X^2+X^3+X^4+X^7.$$

$GF(2^w)$ 内数据块矩阵元素对应的多项式 a 和编码矩阵元素对应的多项式 b 的加法表示为

$$a(X)+b(X)=(a_0\oplus b_0)+(a_1\oplus b_1)X+\cdots+(a_{w-1}\oplus b_{w-1})X^w. \tag{3}$$

GF(2^w) 内 2 个多项式 a 和 b 的乘法可理解为 2 个多项式的乘法, 写为

$$a(X)\otimes b(X)=c_0+c_1X+\cdots+c_jX^j+\cdots+c_{2w-2}X^{2w-2}. \quad (4)$$

式(4)中的 c_j 可写为

$$c_j=(a_0\otimes b_j)\oplus(a_1\otimes b_{j-1})\oplus\cdots\oplus(a_j\otimes b_0). \quad (5)$$

显然, 多项式的乘法结果超出 [0, 2^w–1], 这时将乘法结果对本原多项式 $p(X)$ 取模, 即可将计算结果重新限制在预设的数值范围内[18]. 本原多项式是不能被 GF(2^w) 中任何多项式整除的质多项式, 取模的过程就是逐步削去高阶多项式的过程.

由 GF(2^w) 的加法和乘法运算规则可知, 2 个数据的加法运算只需按位执行 1 次异或操作即可实现. 而乘法运算按照式(4)执行的步骤为:

1)使用移位和与操作按位提取对应操作数位值. 对于 w 位的数据提取需要 (2w–1) 次操作, 这样 2 个操作数共需 (4w–2) 次提取操作.

2)执行多项式乘法. 计算 c_k 需 (k+1) 次⊗操作和 k 次⊕操作. 这样遍历 c_0 至 $c_{2(w-1)}$, 共需要 $(2w-1)^2$ 次操作.

3)c_0 至 $c_{2(w-1)}$ 拼接成一个 (2w–1) 位的数据, 共需 (4w–3) 次操作.

4)执行 (w–1) 次, 削去 c_wX^w 至 $c_{2(w-1)}X^{2(w-1)}$ 高阶多项式, 若 c_k=1, c_kX^k 削去的过程本质上是将 $p(X)$ 移位后与 $c(X)$ 执行异或操作, 从 c_wX^w 至 $c_{2(w-1)}X^{2(w-1)}$, 共 (2w–2) 次操作.

总结式(3)中的 GF(2^w) 乘法, 共需 ($4w^2$+6w–6) 次操作. 显然伽罗华域乘法的算法复杂度远大于伽罗华域加法的复杂度.

1.2 纠删码函数及其应用

纠删码算法的核心计算就是伽罗华域乘法运算, 实现方法有基于异或操作和基于查表操作 2 种. 基于异或操作被 Jerasure[19] 广泛使用, 其支持 GF(2^4) 至 GF(2^{128}) 之间的广域运算, 但基于异或操作计算方法强依赖异或操作的冗余合并算法和调度优化; 基于查表操作被 ISA-L[17] 库广泛使用, ISA-L 是一套用于数据恢复、数据安全、数据压缩和数据加密的开源库. 为了通用性考虑, ISA-L 对数据位宽处理较为固定, 运算过程全部以字节单位为粒度, 即仅支持 GF(2^8) 运算, 但其在运算吞吐量方面远优于 Jerasure[20].

为了支持实际应用场景中的整体数据写入和部分数据修改, ISA-L 为纠删码设计了编解码生成函数和编解码更新函数 2 类接口.

编解码生成函数支持整体数据的写入, 可将所有数据块一次性集中编码生成校验块, 或在数据修复时根据校验块生成数据块. 考虑到编解码生成函数的核心为伽罗华域的矩阵乘, ISA-L 为编码和解码设计了 ENCODE 接口函数. ENCODE 的运算规则为 $\boldsymbol{M}\otimes\boldsymbol{D}=\boldsymbol{E}$, 当 2 维矩阵 $\boldsymbol{M}$ 退化为 1 维矩阵时, 运算规则变为点乘函数(记为 DOTP). 当 2 维矩阵 $\boldsymbol{M}$ 进一步退化为常数时, 运算规则变为基础乘函数(记为 MUL).

编解码更新函数(记为 UPDATE)支持对数据块的修改. UPDATE 的功能是更新内存数据 $\boldsymbol{E}$. 继续以图 2 为例, 将式(2)可展开为

$$\boldsymbol{M}_0\boldsymbol{D}_0+\boldsymbol{M}_1\boldsymbol{D}_1+\cdots+\boldsymbol{M}_j\boldsymbol{D}_j+\cdots+\boldsymbol{M}_{k-1}\boldsymbol{D}_{k-1}=\boldsymbol{E}, \quad (6)$$

其中 $\boldsymbol{M}_j$ 表示编码矩阵 $\boldsymbol{M}$ 的第 j 行, $\boldsymbol{D}_j$ 表示第 j 个数据块. 若部分数据块 $\boldsymbol{D}_j$ 修改为 $\hat{\boldsymbol{D}}_j$, 那么校验块 $\boldsymbol{E}$ 也应修改为 $\hat{\boldsymbol{E}}$. 此时使用编码公式全部重新计算校验块, 计算复杂度等于 (k+m)×k 维矩阵 $\boldsymbol{M}$ 和 k×len 维矩阵 $\boldsymbol{D}$ 在伽罗华域的乘法计算复杂度. 为了降低计算复杂度, 编解码更新函数从式(6)演变为

$$\begin{aligned}\hat{\boldsymbol{E}}&=\boldsymbol{M}_0\boldsymbol{D}_0+\boldsymbol{M}_1\boldsymbol{D}_1+\cdots+\boldsymbol{M}_j\hat{\boldsymbol{D}}_j+\cdots+\boldsymbol{M}_{k-1}\boldsymbol{D}_{k-1}=\\&\boldsymbol{E}+\boldsymbol{M}_j(\hat{\boldsymbol{D}}_j-\boldsymbol{D}_j)=\boldsymbol{E}+\boldsymbol{M}_j\Delta\boldsymbol{D}_j.\end{aligned} \quad (7)$$

UPDATE 重新计算了部分数据块的修改值 $\Delta\boldsymbol{D}_j$ 所对应的校验块 $\boldsymbol{M}_j\Delta\boldsymbol{D}_j$, 并与原校验块相加得到更新后的校验块. 由于伽罗华域加法的计算复杂度远低于伽罗华域乘法, UPDATE 的计算复杂度仅需考虑 $\boldsymbol{M}_j$ 与 $\Delta\boldsymbol{D}_j$ 相乘部分, 编码更新函数的计算复杂度等同于维数为 (k+m)×1 的矩阵 $\boldsymbol{M}_j$ 和维数为 1×len 的矩阵 $\Delta\boldsymbol{D}_j$ 在伽罗华域的乘法计算复杂度. 可见, 使用编码更新的方法在计算复杂度上远低于全部重新计算校验块的方法.

对式(7)进行抽象, 得到 UPDATE 的运算规则, 记为 $(\boldsymbol{M}\otimes\boldsymbol{D})\oplus\boldsymbol{E}=\hat{\boldsymbol{E}}$, 这里的 $\boldsymbol{M}$ 是 $\boldsymbol{M}_j$ 的简写, $\boldsymbol{D}$ 是 $\Delta\boldsymbol{D}_j$ 的简写. 具体实现时, $\boldsymbol{E}$ 和 $\hat{\boldsymbol{E}}$ 处于同一段内存空间. 当 UPDATE 的一维矩阵 $\boldsymbol{M}$ 退化为常数 a 时, 运算规则退化为乘加函数(记为 MAD).

总结起来, 编解码函数和编码更新函数细化为 5 类:

1)ENCODE 表示 2 维矩阵 $\boldsymbol{M}$ 与数据矩阵 $\boldsymbol{D}$ 相乘;

2)DOTP 表示 1 维矩阵 $\boldsymbol{M}$ 与数据矩阵 $\boldsymbol{D}$ 相乘;

3)MUL 表示 2 维矩阵 $\boldsymbol{M}$ 中的常数 a 与数据矩阵 $\boldsymbol{D}$ 相乘;

4)UPDATE 表示 1 维矩阵 $\boldsymbol{M}$ 与数据矩阵 $\boldsymbol{D}$ 相乘后再与校验矩阵 $\boldsymbol{E}$ 相加;

5)MAD 表示 1 维矩阵 $\boldsymbol{M}$ 中的常数 a 与数据矩阵 $\boldsymbol{D}$ 相乘后再与校验矩阵 $\boldsymbol{E}$ 相加.

5类函数涵盖了纠删码的主要使用场景，属于纠删码的核心函数. 表1所列函数的核心算法都是多层循环求解伽罗华域乘法与伽罗华域加法，在原理上是一致的.

Table 1 Implementation Functions of Erasure Codes Algorithm

表1 纠删码算法的实现函数

函数功能说明	函数名	运算规则	函数名缩写
基础乘	*gf_vect_mul*	$a\otimes \boldsymbol{D}=\boldsymbol{E}$	MUL
点乘	*gf_vect_dot_prod*	$\boldsymbol{M}\otimes \boldsymbol{D}=\boldsymbol{E}$	DOTP
乘加	*gf_vect_mad*	$(a\otimes \boldsymbol{D})\oplus \boldsymbol{E}=\hat{\boldsymbol{E}}$	MAD
编解码	*gf_encode_data*	$\boldsymbol{M}\otimes \boldsymbol{D}=\boldsymbol{E}$	ENCODE
编码更新	*gf_encode_data_update*	$(\boldsymbol{M}\otimes \boldsymbol{D})\oplus \boldsymbol{E}=\hat{\boldsymbol{E}}$	UPDATE

接下来以编码生成函数ENCODE为例开展优化算法.

编码生成函数的数据块 $\boldsymbol{D}$ 的维数为 $k\times len$，编码矩阵 $\boldsymbol{M}$ 的维数为 $m\times k$，校验块 $\boldsymbol{E}$ 的维数为 $m\times len$. 依据式(1)，编码函数的计算过程如算法1所示，属于一个 *m*—*len*—*k* 的3层循环. 在第2层循环中申请字节类型临时变量 *temp*(行③)，在最内层循环中，循环执行 *M*[*l*][*j*] 和 *D*[*j*][*i*] 的 GF(2^8) 乘积并累加至 *temp*(行⑤)，最后将 *temp* 运算结果赋予 *E*[*l*][*i*](行⑦)，编码计算结束.

算法 1. 编码生成函数核心算法.

① for $l=0:m-1$
② for $i=0:len-1$
③ $temp=0$;
④ for $j=0:k-1$
⑤ $temp=temp\oplus(M[l][j]\otimes D[j][i])$;
⑥ end for
⑦ $E[l][i]=temp$;
⑧ end for
⑨ end for

算法1中，GF(2^8) 加法使用异或操作实现，GF(2^8) 乘法涉及到LT方法和MT方法. LT方法将2个字节类型的数据 *a* 和 *b* 的 GF(2^8) 乘法求解转换为查表过程，包括2次对数表查找与1次指数表查找，具体计算过程见表2. LT会提前将1B大小的数据的所有可能的指数结果(表2的 *gf_exp*)与对数结果(表2的 *gf_log*)保存起来，计算乘积时，借助 $a\otimes b=2\wedge(\mathrm{lb}\,a+\mathrm{lb}\,b)$ 即可完成运算. 考虑到字节数据处理粒度，LT方法的指数表与对数表通常比较小，不会超出处理器的缓存大小，因此其运行效率对处理器缓存大小不敏感，GF(2^8) 乘法计算默认采取该方法. 但LT方法的查表次数较多，查表引入访存开销较大.

Table 2 Summary of Four Methods for GF(2^8) Multiplication

表2 GF(2^8) 乘法的4种计算方法汇总

方法	伪代码	表大小/B	查表次数	运算次数
LT	$c=gf_exp[gf_log[a]+gf_log[b]]$	512	3	1
MT	$c=gf_mul_table[b\times 256+a]$	65 536	1	2
SH	$c=0$; $c=c\wedge((a\ \&\ 0x1)?\ t_0:0)$; $c=c\wedge((a\ \&\ 0x2)?\ t_1:0)$; $c=c\wedge((a\ \&\ 0x4)?\ t_2:0)$; $c=c\wedge((a\ \&\ 0x8)?\ t_3:0)$; $c=c\wedge((a\ \&\ 0x10)?\ t_4:0)$; $c=c\wedge((a\ \&\ 0x20)?\ t_5:0)$; $c=c\wedge((a\ \&\ 0x40)?\ t_6:0)$; $c=c\wedge((a\ \&\ 0x80)?\ t_7:0)$;	0	0	24
SP	$low=a\ \&\ 0x0f$; $high=(a\gg 4)\ \&\ 0x0f$; $c=gf_split[low]\wedge gf_split[high+16]$;	32	2	4

注：∧表示异或操作.

为了避免LT方法引入的多次查表问题，MT方法将2个字节类型变量所有可能的相乘结果保存起来，计算乘积时直接从乘法表(表2的 *gf_mul_table*)中查找即可. 相比于LT的3次查表，MT查表次数仅为1次，具体计算过程见表2. MT方法虽然在查表次数上有所减少，但乘法表占用空间较大，通常为64 KB，可能超出L1d缓存的大小(如Intel Xeon E5-2620的L1d Cache为32 KB，SW3231的L1d Cache为32 KB). 实际运行时，MT方法的查表操作会从L1d缓存访问跨越对L2缓存访问，访问时间成倍增加，导致查表效率下降.

2 纠删码算法的优化研究

经分析，算法1的运算耗时集中在2个部分：1)伽罗华域的乘法运算；2)访问矩阵 $\boldsymbol{M}$, $\boldsymbol{D}$, $\boldsymbol{E}$ 自身开销. 对于伽罗华域的乘法运算，LT方法的对数表和指数表占用空间较小，但查表次数较多；MT方法的查表次数较少，但乘法表占用空间大. 对于 $\boldsymbol{M}$, $\boldsymbol{D}$, $\boldsymbol{E}$ 矩阵的访问，算法1在存取矩阵数据时以字节为单位，属于一种细粒度的数据操作. 但事实上，64位的现代处理器支持以长字为单位的数据粒度操作. 另外，对矩阵 $\boldsymbol{D}$ 采取按行存储和按列读取的方式，属于离散访

存, 不利于缓存命中.

这样, 针对伽罗华域乘法, 在查表时间开销与表存储空间占用的均衡就变得很关键. 假如矩阵访存使用 64 位内存访问指令一次性读取矩阵 $\boldsymbol{D}$ 的 8 列数据, 将访存粒度从 8 b 提升到 64 b, 则可以极大提升循环内数据处理效率. 此外, 本文基于申威指令集提出了 SIMD 数据并行访问优化, 以提升纠删码函数的运算效率.

2.1 伽罗华域乘法优化

已有的 LT 和 MT 伽罗华域乘法计算方法存在查表次数多和表占用空间大的问题, 在性能上依然有提升空间. 首先考虑到的是增加运算量来减少查表操作, 本文尝试使用移位分解法(shift, SH), 采用移位和异或操作按位进行运算以完成所有字节的乘法运算. 然而以位粒度为单位的运算会显著增加运算开销, 额外引入的运算开销会抵消查表减少所带来的收益. 因此, 本文结合查表法与移位分解法的优点, 提出 4 b 分割法, 将运算单位粒度从 1 b 提升至 4 b. 在减少运算开销的同时控制查表开销.

2.1.1 SH 方法

SH 方法将多项式以位为单位进行分割, 然后分别对 a 的每个位与多项式 b 进行运算, 最后将所有位的运算结果进行累加.

具体来说, SH 方法将 2 个字节类型的数据 a 和 b 的 GF(2^8) 乘法运算分解为: 1)a 的每个位与 b 做 GF(2^8) 乘积; 2)所有乘积结果做 GF(2^8) 加. SH 方法可表达为

$$a\otimes b=\sum_i a_i^{\text{bit}}\otimes b, \tag{8}$$

其中 a_i^{bit} 表示 a 中第 i 个位左移 i 位. 当 a_i=1 时, $a_i^{\text{bit}}\otimes b$=$(1\ll i)\otimes b$, 记为 t_i=$(1\ll i)\otimes b$, 当 a_i=0 时, $a_i^{\text{bit}}\otimes b$=0, 因此式(8)可写为

$$a\otimes b=\sum_i (a\&(1\ll i))?t_i:0. \tag{9}$$

SH 方法预先计算出所有 t_i, $i=0,1,\cdots,7$, 并且在运算前从内存中取出, 这样可以减少 SH 方法的运算次数. 对于式(9), 由于 a 按位做分割, SH 方法的循环次数为 8, 每次循环中共有 1 次与操作、1 次三目运算(也叫条件选择操作)、1 次异或操作. 完成 a 与 b 的乘法计算共需要执行 24 次运算, 具体运算过程见表 2.

SH 方法无需查表操作, 但运算次数显著增加. 假设每次运算所耗平均时间为 N_{ca}, L1d 缓存访问时间为 N_{L1d}, 内存访问时间为 N_{m}. 现代处理器中, N_{ca} 通常为 0.5~1 个时钟周期, N_{L1d} 为 3~4 个时钟周期, N_{m} 从几十到上百个时钟周期不等[21-22]. 所有查表操作的平均耗时记为 N_{av}, 那么 N_{av} 为

$$N_{\text{av}}=ratio\times N_{\text{L1d}}+(1-ratio)\times N_{\text{m}},$$

其中 $ratio$ 为缓存命中率, N_{av} 的值介于 N_{L1d} 与 N_{m} 之间.

考虑到现代计算机的指令执行多以乱序执行, 不存在数据依赖的指令可依赖超标量引擎并行发射[21-22], LT 方法的执行流程见图 3(a). 图中的①~③可总结为: ①同时执行 2 个 *gf_log* 查表, 耗时 N_{av}; ②执行指令加法, 耗时 N_{ca}; ③执行 *gf_exp* 查表, 耗时 N_{av}. LT 方法所耗总时间为 $N_{\text{av}}+N_{\text{ca}}+N_{\text{av}}=2N_{\text{av}}+N_{\text{ca}}$.

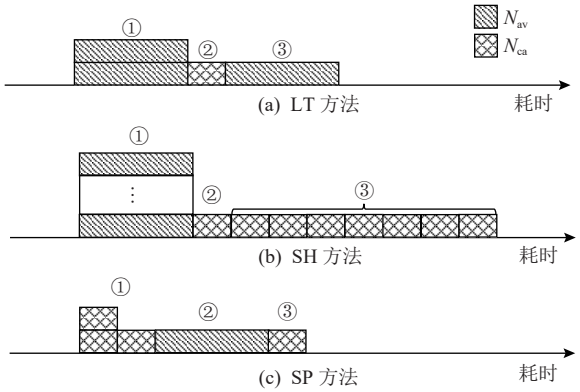


Fig. 3 Execution flow of GF(2^8) multiplication

图 3 GF(2^8) 乘法的执行流程

SH 方法的执行流程如图 3(b). 图中的①~③总结为: ①并行地从内存中读取所有 t_i, $i=0,1,\cdots,7$, 同时执行所有的与操作, 耗时 N_{av}; ②同时执行所有的条件选择操作, 耗时 N_{ca}; ③逐个执行异或操作共 7 次, 耗时 $7N_{\text{ca}}$. SH 方法所耗总时间为 $N_{\text{av}}+N_{\text{ca}}+7N_{\text{ca}}$ = $N_{\text{av}}+8N_{\text{ca}}$.

对比 LT 方法与 SH 方法所耗时间. 当 $N_{\text{av}}<7N_{\text{ca}}$ 时, $2N_{\text{av}}+N_{\text{ca}}<N_{\text{av}}+8N_{\text{ca}}$, 意味着 SH 方法耗时比 LT 方法耗时长. 可见缓存命中率 $ratio$ 较高时, SH 方法运行效率低于 LT 方法. 显然一味地避免表查找并不能提升伽罗华域乘法的运算效率. 另外, 以位为运算单位的方式会导致运算数量激增, 额外引入的运算开销会抵消查表减少所带来的收益.

为了进一步减少伽罗华域乘法的运算耗时, 本文结合 MT 方法与 SH 方法, 提出了 4 b 分割法(4-bit splitting, SP), 在减少运算开销的同时控制查表开销.

2.1.2 SP 分割法

SP 方法在 SH 方法基础上, 将数据块中每个多项式 a 分割为 2 部分, 同时利用查表法将 SH 方法的运算转换为查表操作.

具体来说，SP 方法将数据块中每个多项式 a 分割为位宽为 4 b 的 *high* 与位宽为 4 b 的 *low* 这 2 部分，预先计算 *high*, *low* 与编码矩阵中每一位相乘的所有可能的值存入 *gf_split* 表，在计算乘积时只需开展 2 次查表即可得到 a 与 b 的运算结果，具体计算过程见表 2.

考虑到 *high* 与 *low* 均为 4 b，那么 *high*, *low* 与字节类型的变量 b 相乘的结果表均为 16 B. 这里将该表称为 *gf_split* 表，它的前 16 B 保存 *high* 与 b 的相乘结果，后 16 B 保存 *low* 与 b 的相乘结果，总大小为 32 B. SP 方法计算时使用 1 次移位和 2 次与操作提取 *high* 和 *low*，然后对 *gf_split* 查表 2 次，执行 1 次异或操作后得到运算结果.

SP 方法中，变量 b 的不同取值得到不同 *gf_split* 表，这里将编码矩阵中每个字节看作 b，在进行编码操作前，预先计算出所有 *gf_split* 表，使用时查表完成. 因此，SP 方法在对编码矩阵中不同变量 b 进行运算时，需频繁读取 *gf_split* 表，伴随有内存数据的换入换出. 此外，SP 方法会增加存储开销，存储编码矩阵的内存空间增加至 32 倍（编码矩阵的维数为 $(k+m)\times k$，前 k 行组成单位矩阵，只有后 m 行保存在内存中，因此 SP 方法为编码矩阵提供的内存空间大小为 $(32\times k\times m)$B）. 考虑到纠删码的运算性能，k 和 m 的取值一般较小，工程实现中 (k, m) 通常取值有 (3, 2), (6, 3), (10, 4), (17, 3)[9]，因此 SP 方法为编码矩阵提供的内存空间大小为 100~2 048 B，存储开销在可接受范围.

SP 方法对应实现为算法 2.

算法 2. 编码函数的 SP 方法实现.

① for $l = 0 : m-1$
② for $i = 0 : len-1$
③ $temp = 0$;
④ for $j = 0 : k-1$
⑤ Load $D[j][i]$ to *src*;
⑥ Point P to *gf_spit* of $M[l][j]$;
⑦ $low = src$ & 0x0f;
⑧ $high = (src \gg 4)$ & 0x0f;
⑨ $s = P[low] \wedge P[high+16]$;
⑩ $temp \wedge= s$;
⑪ end for
⑫ Store *temp* to $E[l][i]$;
⑬ end for
⑭ end for

算法 2 的实施步骤为：

1）将 $D[j][i]$ 一次性加载到临时变量 *src* 中（行⑤）.

2）将指针 P 指向 $M[l][j]$ 所对应的 *gf_split* 查找表（行⑥）.

3）执行移位、与操作（行⑦⑧）.

4）对 P 查表，并执行与操作（行⑨）.

5）异或操作得到伽罗华域和（行⑩）.

SP 方法的执行流程见图 3（c），图中的①~③总结为：①同时计算 *low* 和 *high*，计算 *low* 耗时 N_{ca}，计算 *high* 耗时 $2N_{ca}$，总耗时记为 $2N_{ca}$；②并行执行 2 个查表操作，耗时记为 N_{av}；③执行异或操作，耗时记为 N_{ca}. SP 方法所耗总时间为 $2N_{ca}+N_{av}+N_{ca} = N_{av}+3N_{ca}$. 可见，SP 方法耗时小于 SH 方法耗时，考虑到 $N_{av} > 2N_{ca}$，SP 方法耗时也小于 LT 方法耗时.

2.2 利用 CPU 资源优化

SP 方法平衡了运算和查表访问开销，但其依旧存在一个问题：继承了 GF(2^8) 的细粒度计算，即运算仍然按字节为处理粒度开展. 事实上，各个字节之间的读取过程并不总是存在数据依赖关系，对 a 和 b 的数据装入和运算粒度可以提升至字或者长字，这样既利用了现代处理器高位宽的寄存器和指令特点，又减少了不必要的内存访问.

2.2.1 数据访问粒度扩展

分析算法 2，发现有 3 点不足：

1）数组 $\boldsymbol{D}$ 在内存中按行存储，相应地，数据存储时也按行存储，但变量 $D[j][i]$ 的值是按列加载到 *src* 中. 引发离散访存，对缓存局部性利用不足.

2）移位、与操作都以字节为单位开展，没有充分发挥现代处理器高位宽的寄存器和指令特点.

3）在最内层循环中，指针 P 被不断访问，所指向的内存被频繁地从缓存换入换出，引发缓存颠簸.

针对这 3 点不足，本文在算法 2 基础上进一步扩增了数据访问粒度，从原来的按字节访问扩展为按长字操作，形成优化算法 3. 该算法基于一次内存访问、多次数据利用的思路，减少了访存次数. 具体实施步骤如下.

1）访存粒度从字节提升为长字，一次性读取 64 b 位宽的数据 $D[j][8\times i:8\times i+7]$ 到 *src*. 对应的最外层的循环次数减少为原来的 1/8.

2）移位、与的运算处理粒度从字节提升为长字.

3）将 64 b 位宽的数据 *low* 对 P 的查表按字节展开成 8 个 8 b 位宽的数据对 P 的查表；将 1 个字节型变量 *temp* 扩展为 8 个字节型 *temp* 变量（记为 $temp_0$ 至 $temp_7$）用于保存临时计算结果. 分别对 8 个 *temp* 变量执行查表计算，最后将 8 个变量结果依次保存到 $E[l][8\times i:8\times i+7]$ 中.

算法 3. 数据访问粒度扩展优化算法.

① $mask$ = 0x0f0f0f0f0f0f0f0f;
② for l = 0 : m–1
③ for i = 0 : (len–1)/8
④ $temp_0$ = 0, $temp_1$ = 0, $temp_2$ = 0, $temp_3$ = 0;
⑤ $temp_4$ = 0, $temp_5$ = 0, $temp_6$ = 0, $temp_7$ = 0;
⑥ for j = 0 : k–1
⑦ Load $D[j][8\times i : 8\times i+7]$ to src;
⑧ Point P to gf_spit of $M[l][j]$;
⑨ $low = src \,\&\, mask$;
⑩ $high = (src \gg 4) \,\&\, mask$;
⑪ $temp_{0\sim7} \wedge= P[(low \gg 4(8\times(0\sim7))) \,\&\, 0x0f] \wedge P[((high \gg 4(8\times(0\sim7))) \,\&\, 0x0f)+16]$;
⑫ end for
⑬ Store $temp_{0\sim7}$ to $E[l][8\times i:8\times i+7]$;
⑭ end for
⑮ end for

2.2.2 数据级并行优化

从算法 2 到算法 3 的优化, 扩展了 GF(2^8) 在单循环内的数据访问粒度, 提升了内存访问效率, 另外变量 $temp_0$ 至 $temp_7$ 之间的独立性为数据并行优化提供了条件. 但是算法 3 还是以标量的代码实现为主, 对于支持超长字的处理器, 提出用 SIMD 对数据开展进一步并行化.

以申威 SIMD 指令为分析对象, 对算法 2 开展数据级并行优化. 申威 SIMD 支持多种向量数据类型, 包括: 长度为 32 B 的整数向量(32×8 b)、长度为 16 半字的整数向量(16×16 b)、长度为 8 b 的整数向量(8×32 b)等.

然而要对算法 2 进行数据并行优化, 需要解决 2 个关键问题: 如何支持位宽为 256 b 的数据在 GF(2^8) 的运算(C1); 如何提升查表效率, 减少对数组 P 的访问次数(C2).

对于 C1 问题, 预先装入表 P (即 gf_tbl 表)的内容到 4 个宽度为 8×32 b 的向量寄存器, 记为 Va_1, Vb_1, Va_2, Vb_2. 这 4 个向量寄存器的每一个 32 b 数据都由表 P 中的一个 8 b 数据扩展得到. 如图 4, Va_1 对应表 P 的第 0~7 个 8 b 数据, Vb_1 对应表 P 的第 8~15 个 8 b 数据; Va_2 对应表 P 的第 16~23 个 8 b 数据, Vb_2 对应表 P 的第 24~31 个 8 b 数据. 那么算法 2 中 $P[low]$ 是针对 Va_1, Vb_1 的操作, $P[high+16]$ 是针对 Va_2, Vb_2 的操作.

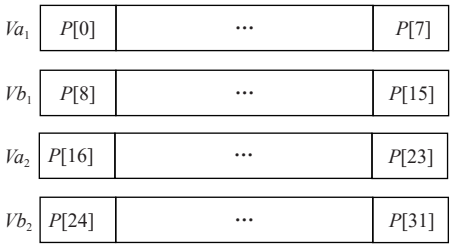


Fig. 4 Data structure of Va_1, Vb_1, Va_2, Vb_2

图 4 Va_1, Vb_1, Va_2, Vb_2 的数据结构

对于 C2 问题, 申威 SIMD 指令将 P 一次性载入到向量寄存器中, 然后使用字向量混洗(word vector shuffle, VSHFW)指令对向量寄存器中的数据进行重排以替代多字节的查表操作. 然而, 将 P 直接载入到向量寄存器时, 会引入一个新的重复载入问题: 算法 2 的 3 层循环次序为 m—len—k, 表 P 指向的内存 $M[l][j]$ 的循环次序是 m—k, 那么在算法 2 的最内层循环里, 表 P 重复 len 次指向 $M[l][j]$, 表 P 重复 len 次载入到向量寄存器. 为了避免该问题, 我们调整了循环之间的顺序.

结合内存存取、与/异或、移位和查表指令的 SIMD 实现, 申威平台上实现纠删码算法的 SIMD 优化版本见算法 4.

算法 4. 算法 2 的向量优化版本.

① $mask$ = 0x0f0f0f0f;
② for l = 0 : m–1
③ for j = 0 : k–1
④ Set Va_1, Vb_1, Va_2, Vb_2 by gf_tbl of $M[l][j]$;
⑤ for i = 0 : (len–1)/4
⑥ Load $D[j][4\times i : 4\times i+3]$ to src_{32};
⑦ $low_{32} = src_{32} \,\&\, mask$;
⑧ $high_{32} = (src_{32} \gg 4) \,\&\, mask$;
⑨ $Vc = VSHFW\,(Va_1, Vb_1, low_{32})$;
⑩ $Vd = VSHFW\,(Va_2, Vb_2, high_{32})$;
⑪ $Vd = Vd \wedge Vc$;
⑫ Extract $temp$ from Vd;
⑬ $E[l][4\times i : 4\times i+3] \wedge= temp$;
⑭ end for
⑮ end for
⑯ end for

算法 4 的核心是 3 层循环 m—len—k, 最内层循环的计算步骤为:

1)将32b 数据$D[j][4\times i:4\times i+3]$ 加载到src_{32} 中(行⑥).

2)对 src_{32} 执行移位、与操作得到 low_{32} 与 $high_{32}$ (行⑦⑧).

3)使用查表指令 VSHFW 得到 Vc 和 Vd(行⑨⑩).

4)Vc 和 Vd 异或, 结果存入 Vd(行⑪).

5)将 Vd 的有效部分抽取为 32 b 数据 $temp$.

6)$temp$ 和 $E[l][4\times i:4\times i+3]$ 异或, 结果存入 $E[l][4\times i:4\times i+3]$.

3 实验结果

为了验证本文所提出的 SP 方法和利用 CPU 资源优化的加速效果，我们在 x86 平台和申威平台做了性能加速比测试与分析. 基于 Intel ISA-L 开源库实现了上述优化，并对其纠删码中最核心的 5 类函数进行验证.

3.1 实验设计

3.1.1 实验环境

所选择的 x86 平台和申威平台测试环境配置如表 3 所示.

Table 3 Experimental Environment

表 3 实验环境

环境细则	申威平台	x86 平台
CPU	SW3231 @ 2.40 GHz 64 核	Intel Xeon E5-2620 @ 2.10 GHz 32 核
一级缓存/KB	32	32
二级缓存/KB	512	256
三级缓存/KB	65 536	20 480
操作系统	UnionTech OS Server v20 Enterprise	CentOS Linux release 8.1.1911
编译器	GCC 8.3.0	GCC 8.3.1
ISA-L 库	V2.30.0	V2.30.0

3.1.2 测试函数

测试函数选用 ISA-L 中纠删码所用到的 MUL, DOTP, MAD, ENCODE, UPDATE 共 5 类函数. 选取 ISA-L 的纠删码算法中自带的 perf 测试集来调用上述 5 种函数功能.

在数据规模选取上(矩阵 $\boldsymbol{E}$ 和 $\boldsymbol{D}$ 占用的总空间量可描述为数据规模)，考虑到 SW3231 和 E5-2620 的实际缓存容量大小，我们通过动态调整矩阵 $\boldsymbol{E}$ 和 $\boldsymbol{D}$ 的数据规模，确保数据规模覆盖缓存范围内和缓存范围外的不同情况(范围选取 8~256 MB). $\boldsymbol{M}$ 的维数为 4×10.

3.1.3 实验方法

本文基于吞吐量来评估优化效果. 吞吐量作为衡量纠删码算法的常用标准[17]，其表示在单位时间内处理的数据规模能力. 为了减少吞吐量计算带来的误差，所有测试数据均采取重复运行 10 遍取算术平均值得到.

当前开源 ISA-L 库中纠删码运算默认采取 LT 方法实现，因此我们选取 LT 方法为对比基准. 不同方法的优化效果用加速比来衡量，加速比的计算方式为不同优化方法的吞吐量与 LT 方法吞吐量的比值.

3.2 不同伽罗华域乘法运算性能对比

我们从 2 个方面评估伽罗华域乘法性能：1)计算基于 SH, MT, LT, SP 这 4 种方法时纠删码算法执行的吞吐量；2)分析上述 4 种方法所引入的存储开销.

3.2.1 不同伽罗华域乘法的吞吐量对比

图 5 和图 6 为申威平台和 x86 平台上基于 SH, MT, LT, SP 这 4 种方法时纠删码算法执行的吞吐量对比情况.

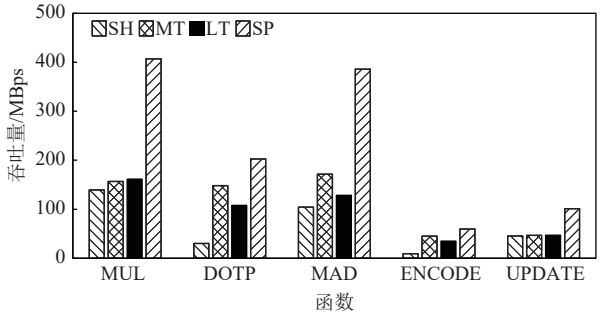


Fig. 5 Throughput comparison of four methods for GF(2^8) multiplication on Sunway platform

图 5 申威平台 GF(2^8) 乘法的 4 种方法吞吐量对比

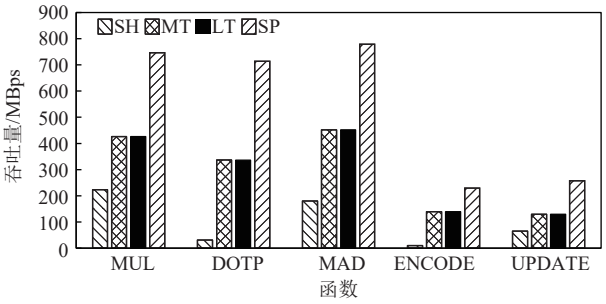


Fig. 6 Throughput comparison of four methods for GF(2^8) multiplication on x86 platform

图 6 x86 平台 GF(2^8) 乘法的 4 种方法吞吐量对比

在申威平台，MT 方法的平均吞吐量是 LT 方法的 1.19 倍，SH 方法的平均吞吐量是 LT 方法的 0.64 倍，SP 方法的平均吞吐量是 LT 方法的 2.25 倍；在 x86 平台，MT 方法的平均吞吐量是 LT 方法的 1.00 倍，SH 方法的平均吞吐量是 LT 方法的 0.32 倍，SP 方法的平均吞吐量是 LT 方法的 1.85 倍.

从结果看出，SP 方法的吞吐量最高，MT 方法和 LT 方法次之，SH 方法最低.

具体看来，申威平台上 SH 方法在 MUL, MAD 和 UPDATE 上的吞吐量分别达到 LT 方法的 86.24%, 81.29%, 96.26%，但 DOTP 和 ENCODE 函数只达到 LT 方法的 28.11% 和 26.79%. 这是由于 MUL, MAD, UPDATE 函数采取 2 层循环运算，而 DOTP, ENCODE 函

数是 3 层循环运算. SH 方法每一次执行最内层循环时, 都需要根据不同的编码元素 $M[l][j]$ 重新从内存中读取式(9)中的 t_l, 影响对缓存数据的刷新频率与命中率.

进一步分析 SH, MT, LT, SP 在运算时对内存操作的影响, 统计其缓存失效率. 在申威平台上, 4 种 GF(2^8) 乘法的缓存失效率结果如图 7 所示, SH 方法执行 MUL, MAD, UPDATE 函数的缓存失效率是 LT 方法的 0.97 倍、1.00 倍和 0.98 倍, 执行 DOTP 和 ENCODE 函数时缓存失效率则为 1.40 倍和 1.27 倍. SH 方法在执行 DOTP 和 ENCODE 函数时性能差, 主要归结于缓存失效率较高.

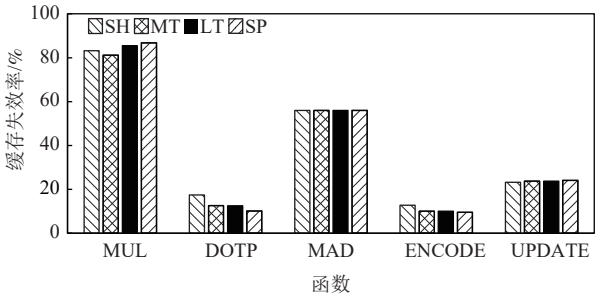


Fig. 7 Cache miss ratios comparison of four methods for GF(2^8) multiplication on Sunway platform

图 7 申威平台 GF(2^8) 乘法的 4 种方法缓存失效率对比

同样地, x86 平台上 4 种 GF(2^8) 乘法的缓存失效率测试结果见图 8, 同样验证了 SH 方法在执行 DOTP 和 ENCODE 函数时存在缓存未命中率较高的现象.

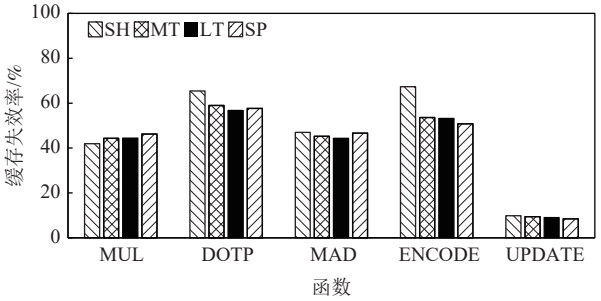


Fig. 8 Cache miss ratios comparison of four methods for GF(2^8) multiplication on x86 platform

图 8 x86 平台 GF(2^8) 乘法的 4 种方法缓存失效率对比

3.2.2 不同伽罗华域乘法的存储开销对比

本文的存储开销是指纠删码在运算过程中对内存资源的占用, 因此以程序执行过程中的最大内存消耗作为量化指标, 以评估不同伽罗华域乘法方法的存储开销.

表 4 和表 5 为申威平台和 x86 平台上 SH, MT, LT, SP 这 4 种不同计算方法的存储开销对比情况.

Table 4 Memory Overhead Comparison of Four Methods for GF(2^8) Multiplication on Sunway Platform

表 4 申威平台 GF(2^8) 乘法 4 种方法的存储开销对比 KB

函数	SH	MT	LT	SP
MUL	136 752	136 824	136 824	136 824
DOTP	44 424	44 424	44 424	44 436
MAD	44 304	44 368	44 304	44 304
ENCODE	43 888	43 928	43 936	44 004
UPDATE	193 932	193 960	193 968	194 024

Table 5 Memory Overhead Comparison of Four Methods for GF(2^8) Multiplication on x86 Platform

表 5 x86 平台 GF(2^8) 乘法 4 种方法的存储开销对比 KB

函数	SH	MT	LT	SP
MUL	132 876	132 848	132 880	132 880
DOTP	44 384	44 432	44 408	44 436
MAD	44 376	44 376	44 376	44 376
ENCODE	43 948	44 004	43 980	44 032
UPDATE	193 992	19 4 016	193 992	19 4 016

对于申威平台的 5 类函数的平均存储开销, SH 方法与 LT 方法相差–0.036%, MT 方法与 LT 方法相差 0.024%, SP 方法与 LT 方法相差 0.042%; 在 x86 平台的平均存储开销, SH 方法与 LT 方法相差–0.026%, MT 方法与 LT 方法相差 0.019%, SP 方法与 LT 方法相差 0.039%.

从结果看, 4 种方法的存储开销差异很小, 平均存储开销最高的 SP 方法与 LT 方法相差不超过 0.05%. 综上, SP 方法的存储开销略高于 LT 方法, 但在可控范围内. 综合考虑 SP 方法带来的性能提升, SP 方法优于 LT 方法, 适用于纠删码的运算.

3.3 SP 方法性能分析

图 9 为申威平台在不同数据规模情况下, 基于算法 2 的 SP 方法的性能加速比, 其中 MUL, DOTP, MAD,

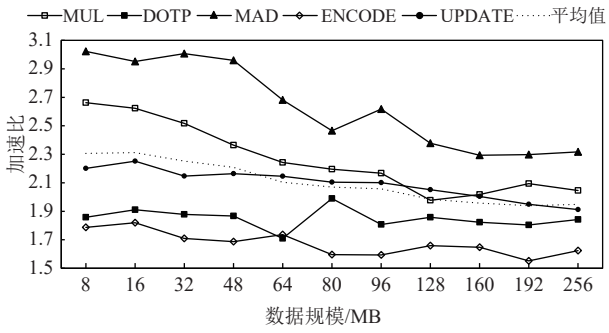


Fig. 9 Speedup with different data scalability by using SP method on Sunway platform

图 9 申威平台采用 SP 方法在不同数据规模的加速比

ENCODE, UPDATE 分别表示 5 类函数. 平均值为 5 类函数的加速比的算术平均值. 整体看来, 在不同数据规模下, 算法 2 的加速比在 1.55~3.02 之间, 平均加速比为 2.10. 其中, 算法 2 对于 MAD 和 MUL 的加速效果最显著. 分析原因, MAD 表示 1 维矩阵 ***M*** 中的常数 a 与数据矩阵 ***D*** 相乘后与校验矩阵 ***E*** 相加, MUL 表示 2 维矩阵 ***M*** 中的常数 a 与数据矩阵 ***D*** 相乘, 两者功能存在重合, 造成加速比趋同的现象.

图 10 为 x86 平台在不同数据规模情况下, 基于算法 2 的 SP 方法加速比. 整体看来, 在不同数据规模下, 算法 2 的加速比在 1.63~2.16 之间, 平均加速比为 1.85. 其中, 算法 2 对于 DOTP 和 UPDATE 的加速效果最显著, 同样地, DOTP 和 UPDATE 的功能存在重合, 原因与申威平台类似.

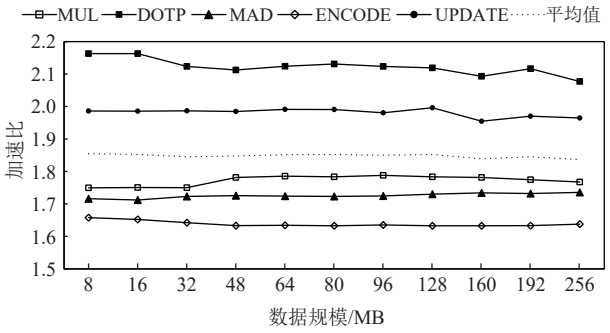


Fig. 10 Speedup with different data scalability by using SP method on x86 platform

图 10 x86 平台采用 SP 方法在不同数据规模的加速比

3.4 数据访问粒度扩展优化实验

图 11 为申威平台在不同数据规模情况下, 基于算法 3 的数据访问粒度扩展优化的加速比. 算法 3 对于 DOTP 运算加速效果最显著, 对于 MUL 加速效果相对较弱. 整体看来, 在不同数据规模下, 算法 3 的加速比在 2.63~4.32 之间, 平均加速比为 3.28. 对比算法 2 中 SP 方法的加速效果, 算法 3 提出的数据访问粒度扩展优化能够在算法 2 的基础上进一步地提升加速比. 有趣的是, 随着数据规模继续增大, 算法 3 在申威平台的加速比呈下降趋势, 当数据规模达到 128 MB 时, 下降幅度有所减缓. 分析原因, 当数据规模增加、访问内存数据块变大时, 查表操作成为制约运算性能的一个重要因素. 此时运算矩阵数据主要是从内存中读取, 缓存失效数量趋于稳定. 总结看来, 数据粒度扩展优化在数据规模较小的情况可以充分利用缓存数据, 效果更好.

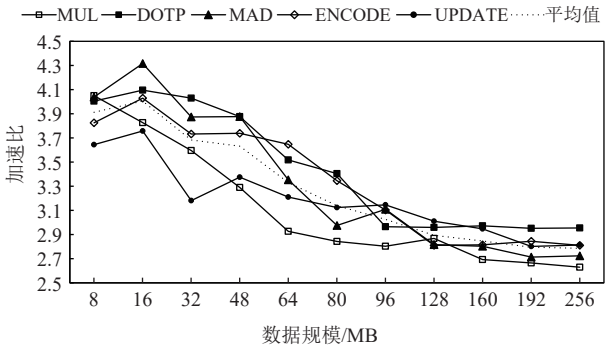


Fig. 11 Speedup with different data scalability by using data granularity amplification optimization on Sunway platform

图 11 申威平台采用数据粒度扩展优化在不同数据规模的加速比

图 12 为 x86 平台在不同数据规模情况下, 基于算法 3 的数据访问粒度扩展优化的加速比. 算法 3 对于 UPDATE 运算加速效果最为显著, 对于 ENCODE 加速效果最弱. 整体看来在不同数据规模下, 算法 3 的加速比在 1.93~2.69 之间, 平均加速比为 2.36. 相比于申威平台, x86 平台上加速比随着数据规模的变化幅度趋于平稳, 主要原因是随着数据规模的增加, 申威平台的访存瓶颈凸显, 而 x86 平台上尚未达到访存瓶颈.

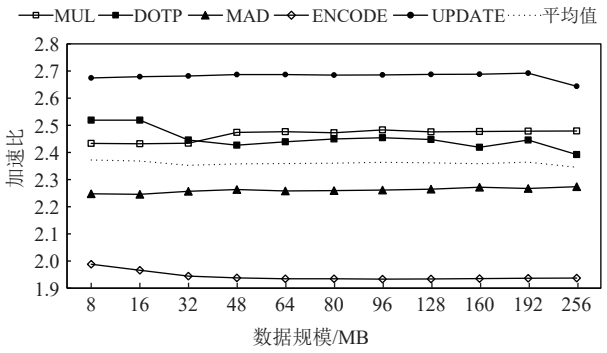


Fig. 12 Speedup with different data scalability by using data granularity amplification optimization on x86 platform

图 12 x86 平台采用数据粒度扩展优化在不同数据规模的加速比

3.5 数据级并行优化实验

在申威平台基于算法 4 实现的 SIMD 并行优化加速效果见图 13, 对于 DOTP 运算加速效果最显著, 对于 MUL 加速效果相对较弱. 整体看来, 在不同数据规模下, 算法 4 的加速比在 2.66~4.19 之间,平均加速比为 3.25. 对比算法 3 的加速效果, 算法 4 基于的 SIMD 并行优化效果整体上与算法 3 效果相当. 算法 4 实现利用了申威 SIMD 指令并行特性, 基于 256 b 位宽的访存指令并行访问缓解了算法 3 中的访存带宽瓶颈问题.

总结图 11 和图 13 发现, 数据规模小于 64 MB 时,

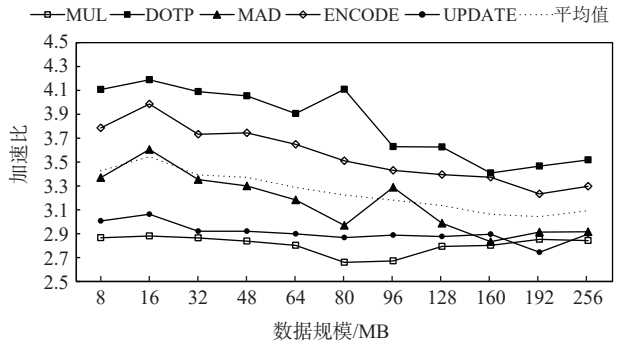


Fig. 13 Speedup with different data scalability by using SIMD on Sunway platform

图 13 申威平台采用 SIMD 并行优化在不同数据规模的加速比

申威平台数据访问粒度扩展优化的加速比大于数据级并行优化的加速比; 当数据规模大于 64 MB 时, 数据级并行优化加速效果更加显著. 算法 3 和算法 4 优化可以针对不同应用场景的数据规模灵活切换算法. 算法 4 引入的基于 SIMD 指令的数据并行优化, 实现上会依赖于特定的体系结构 SIMD 指令的支持情况, 在向其他平台的可扩展性方面存在一定的限制, 但是算法 3 的方法则无此限制.

4 结 论

纠删码的编码与解码具有计算和访存密集的特点, 主要瓶颈点聚焦于多层循环包裹的伽罗华域乘法运算. 基于查表操作的 LT 和 MT 方法实现伽罗华域乘法运算, 被业界广泛使用, 但是其执行效率受限于查表的构造和数据运算次数, 此外以字节为单位的伽罗华域乘法操作在数据并行度利用方面也存在不足.

本文首先针对 LT 方法的不足, 提出了 SP 方法, 将 GF(2^8) 的数据块元素对应的多项式 a 分割为 2 部分, 分别与编码矩阵对应多项式 b 相乘并存入查询表中, 然后在实际 GF(2^8) 运算时基于查表和异或操作完成. 此外, 在 SP 方法基础上, 利用现代 64 位处理器数据位宽特性, 提出了数据访问粒度扩展优化和数据级并行方法. 本文基于 Intel ISA-L 开源库实现了上述优化, 在申威平台和 x86 平台开展实验分析. 对比分析了不同优化方法相比于基准 LT 方法的吞吐量和加速比. 实验表明, 本文提出的优化方法在吞吐量方面优于 LT 与 MT 方法, 在不同的数据规模下, 数据访问粒度扩展优化和数据级并行优化均有加速效果. 申威平台, 数据粒度扩展优化相比 LT 平均加速比达 3.28, SIMD 向量化相比 LT 平均加速比达 3.25; x86 平台, 数据粒度扩展优化相比 LT 平均加速比达 2.36. 表明了本文提出的优化方法的有效性.

此外, SP 方法和数据访问粒度扩展优化的实现方法与架构无关, 具备较强的通用性和普适性, 可直接应用于其他不同硬件平台. 而基于申威指令集的 SIMD 并行方案为其他指令集架构的数据级并行优化提供了参考意义.

作者贡献声明: 谢汶兵提出了算法思路、设计了论文框架; 关睿雪完成论文撰写、完成实验环境部署与实验测试; 张艺鸣完成实验数据处理与实验分析; 李佳梅完成实验数据整理与实验分析; 王俊提出指导意见, 参与论文修改.

参 考 文 献

[1] Plank J S. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems[J]. Software Practice & Experience, 2015, 27(9): 995−1012

[2] Zhou Tianli, Tian Chao. Fast erasure coding for data storage: A comprehensive study of the acceleration techniques[J]. ACM Transactions on Storage, 2020, 16(1): 1−24

[3] Chen P M, Lee E K, Gibson G A, et al. RAID: High-performance, reliable secondary storage[J]. ACM Computing Surveys, 1997, 26(2): 145−185

[4] Wilcox-O'Hearn Z, Warner B. Tahoe: The least-authority filesystem[C]//Proc of the 4th ACM Int Workshop on Storage Security and Survivability. New York: ACM, 2008: 21−26

[5] Ishengoma F. HDFS+: Erasure-coding based Hadoop distributed file system[J]. International Journal of Scientific & Technology Research, 2013, 2(9): 190−197

[6] Hu Yuchong, Cheng Liangfeng, Yao Qiaori, et al. Exploiting combined locality for wide-stripe erasure coding in distributed storage[C]//Proc of the 19th USENIX Conf on File and Storage Technologies (FAST'21). Berkeley, CA: USENIX Association, 2021: 233−248

[7] Bowers K D, Juels A, Oprea A. HAIL: A high-availability and integrity layer for cloud storage[C]//Proc of the 16th ACM Conf on Computer and Communications Security. New York: ACM, 2009: 187−198

[8] Shi Haiyang, Lu Xiaoyi, Shankar et al. UMR-EC: A unified and multi-rail erasure coding library for high-performance distributed storage systems[C]//Proc of the 28th Int Symp on High-Performance Parallel and Distributed Computing (HPDC'19). New York: ACM, 2019: 219−230

[9] Shi Haiyang, Lu Xiaoyi, Panda D K. EC-Bench: Benchmarking onload and offload erasure coders on modern hardware architectures[C]//Proc of the 2018 Benchmarking, Measuring, and

Optimizing: First BenchCouncil Int Symp. New York: ACM, 2019: 215−230

[10] Detchart J, Lacan J. Improving the coding speed of erasure codes with polynomial ring transforms[C/OL]//Proc of the 2017 IEEE Global Communications Conf. Piscataway, NJ: IEEE, 2017[2024-09-27]. https://dl.acm.org/doi/10.1109/GLOCOM.2017.8255009

[11] Uezato Y Y. Accelerating XOR-based erasure coding using program optimization techniques[C/OL]//Proc of the Int Conf for High Performance Computing, Networking, Storage and Analysis (SC'21). New York: ACM, 2021[2024-09-27]. https://doi.org/10.1145/3458817.3476204

[12] Blomer J, Kalfane M, Karp R, et al. An XOR-based erasure-resilient coding scheme[R]. Berkeley, CA: Technical Report at International Computer Science Institute, 1995

[13] Luo Jianqiang, Shrestha, Mochan, et al. Efficient encoding schedules for XOR-based erasure codes[J]. IEEE Computer Society, 2014, 63(9): 2259−2272

[14] Feng Kai, Ma Wentao, Huang Wei et al. Speeding up Galois field arithmetic on Intel MIC architecture[G]//LNTCS 2172: Proc of the 10th IFIP Int Conf on Network and Parallel Computing. Berlin: Springer, 2013: 143−154

[15] Günther S M, Riemensberger M, Utschick W. Efficient GF arithmetic for linear network coding using hardware SIMD extensions[C/OL]// Proc of the 2014 Int Symp on Network Coding. Piscataway, NJ: IEEE, 2014[2024-10-01]. https://ieeexplore.ieee.org/document/6892123

[16] Shin S R, Choo S Y, Park J S. Accelerating random network coding using 512-bit SIMD instructions[C]//Proc of the 2019 Int Conf on Information and Communication Technology Convergence (ICTC). Piscataway, NJ: IEEE, 2019: 1099−1103

[17] Intel. Optimizing storage solutions using the Intel® intelligent storage acceleration library[EB/OL]. (2014-09-24)[2024-05-30]. https://www.intel.com/content/www/us/en/developer/articles/technical/optimizing-storage-solutions-using-the-intel-intelligent-storage-acceleration-library.html

[18] Macwilliams F J, Sloane N J A. The Theory of Error-correcting Codes. Parts I, II[M]. Amsterdam: North-Holland, 1977

[19] Plank J S. Jerasure: A library in C/C++ facilitating erasure coding for storage applications[R]. Knoxville: University of Tennessee, 1995

[20] Ueno R, Homma N, Sugawara Y. Highly efficient GF(2^8) inversion circuit based on redundant GF arithmetic and its application to AES design[G]//LNCS 9293: Proc of the 2015 Cryptographic Hardware and Embedded Systems(CHES'15). Berlin: Springer, 2015: 63−80

[21] Bakhvalov D. Performance Analysis and Tuning on Modern CPUs[M]. Translated by Zhu Jinpeng, Li Chengdong. Beijing: China Machine Press, 2023 (in Chinese)
(Bakhvalov D. 现代 CPU 性能分析与优化[M]. 朱金鹏, 李成栋, 译. 北京: 机械工业出版社, 2023)

[22] Robey R, Zamora Y. Parallel and High Performance Computing[M]. Greenwich: Manning, 2021



Xie Wenbing, born in 1989. PhD candidate. His main research interests include binary translation, runtime libraries optimization, program analysis, and compiler optimization.

谢汶兵, 1989 年生. 博士研究生. 主要研究方向为二进制翻译、运行时库优化、程序分析、编译优化.



Guan Ruixue, born in 1993. Master. Her main research interests include binary instrumentation and runtime libraries optimization.

关睿雪, 1993 年生. 硕士. 主要研究方向为二进制插桩、运行时库优化.



Zhang Yiming, born in 1993. Master. Her main research interests include binary instrumentation, debug tools, runtime libraries optimization.

张艺鸣, 1993 年生. 硕士. 主要研究方向为二进制插桩、调试工具、运行时库优化.



Li Jiamei, born in 1996. Master. Her main research interests include program analysis and runtime libraries optimization.

李佳梅, 1996 年生. 硕士. 主要研究方向为程序分析、运行时库优化.



Wang Jun, born in 1981. Master. His main research interests include program analysis, and autonomous and controllable basic software ecology.

王　俊, 1981 年生. 硕士. 主要研究方向为程序分析、自主可控基础软件生态.