# Rethinking Erasure-Coding Libraries in the Age of Optimized Machine Learning

Jiyu Hu
jiyuh@alumni.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Jack Kosaian
jkosaian@alumni.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

K. V. Rashmi
rvinayak@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

## Abstract

Erasure codes are critical tools for building fault-tolerant and resource-efficient storage systems. However developing and maintaining optimized erasure-coding libraries are challenging. We make the case that the growth of fast machine-learning (ML) libraries may serve as a lifeboat for easing the development of current and future optimized erasure-coding libraries: fast erasure-coding libraries for various hardware platforms can be easily implemented by using existing optimized ML libraries. We show that the computation structure of many erasure codes mirrors that common to matrix multiplication, which is heavily optimized in ML libraries. Due to this similarity, one can implement erasure codes using ML libraries in few lines of code and with little knowledge of erasure codes, while immediately adopting the many optimizations within these libraries, without requiring expertise in high-performance programming. We develop prototypes of our proposed approach using an existing ML library. Our prototypes are up to 1.75× faster than state-of-the-art custom erasure-coding libraries.

***CCS Concepts:*** • **Computer systems organization** → **Redundancy**.

***Keywords:*** erasure coding, machine learning, redundancy

**ACM Reference Format:**
Jiyu Hu, Jack Kosaian, and K. V. Rashmi. 2024. Rethinking Erasure-Coding Libraries in the Age of Optimized Machine Learning. In *16th ACM Workshop on Hot Topics in Storage and File Systems (HOTSTORAGE '24), July 8–9, 2024, Santa Clara, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3655038.3665943

## 1 Introduction

Data storage systems leverage redundancy to avoid data loss in the presence of failures. Erasure codes are key tools from the domain of coding theory that enable storage systems to reliably store data with significantly less storage overhead than replication [22]. Due to these properties, erasure codes are widely used in storage systems (e.g., [13, 14, 27, 34, 40]).

However, developing optimized erasure-coding libraries is currently challenging because of the following reasons: (1) it requires expertise in both the mathematical underpinnings of erasure codes and methods to achieve high performance on a given hardware platform; (2) Erasure coding needs to be performed on increasingly heterogeneous hardware, such as GPUs; and (3) the development is going to be even harder in the future with the rise of various forms of accelerators.

In this paper, we make the case that current trends in optimized machine learning (ML) libraries offer a promising lifeboat for the development of current and future erasure-coding libraries.

We propose that (1) Erasure codes can easily be implemented via ML libraries (and, thus, require little development effort); (2) Erasure codes implemented via ML libraries will immediately adopt the many performance optimizations currently within ML libraries and those that will come as hardware evolves (and, thus, require little optimization and maintenance effort); and (3) Erasure codes implemented via ML libraries are likely to be able to execute on a variety of hardware platforms.

To support these claims, we develop erasure-coding libraries using Apache TVM [9], a popular ML library. Atop TVM, we develop TVM-EC. We compare the performance of the prototype to state-of-the-art erasure-coding libraries targeting CPUs [2, 39]. TVM-EC achieves up to 1.75× higher encoding throughput than the state-of-the-art custom erasure-coding libraries. Furthermore, the prototype required little development effort.

These results showcase the promise of leveraging ML libraries to easily develop optimized erasure-coding libraries targeting current and future hardware platforms, and, thus, to usher in the next generation of erasure-coded systems.

## 2 Background and Related Work

**Erasure code.** An erasure code *encodes* $k$ data units to produce $r$ parity units and stores all $(k+r)$ data units on separate storage devices.[1] Thus, an erasure code withstands up to $r$ lost units with storage overhead of only $\frac{k+r}{k}$.

[1]It is also common to describe erasure codes via parameters $n$ and $k$, where $n$ is the total number of data and parity units (i.e., $n = k + r$).


*HOTSTORAGE '24, July 8–9, 2024, Santa Clara, CA, USA*

ACM ISBN 979-8-4007-0630-1/24/07
https://doi.org/10.1145/3655038.3665943

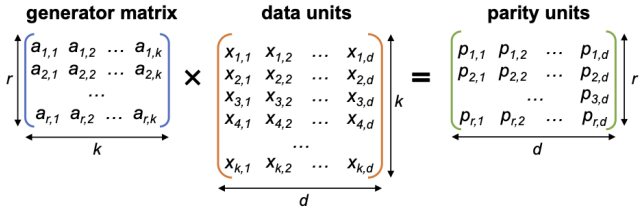



**Figure 1.** High-level depiction of encoding operation

We next describe the high-level operation of encoding in an erasure code. The decoding process is very similar to that of encoding.

**Encoding.** The encoding process of an erasure code takes in $k$ data units and produces $r$ parity units, where each unit contains $d$ elements. Each parity unit is formed via a linear combination of the $k$ data units. The encoding of a single parity element can thus be viewed as the dot product between a vector of $k$ coefficients and a vector of $k$ data elements. Expanding this to the encoding of $r$ parity units, encoding can be viewed as a matrix-matrix multiplication between a *generator matrix* $E$ of size $(r \times k)$ and a *data matrix* $D$ of size $(k \times d)$ to produce a matrix $P$ of size $(r \times d)$ containing parity units. This is illustrated in Figure 1.

### 2.1 Optimizing erasure code implementations

We next describe techniques that have been used for developing high-performance erasure-coding libraries.

**Bitmatrix erasure coding.** The arithmetic used in encoding and decoding in erasure codes is done via finite-field arithmetic, which is often far more computationally expensive than traditional arithmetic. To ameliorate this expense, the so-called "bitmatrix" erasure-coding procedure transforms an erasure code that operates over a Galois Field of size $2^w$ into one operating over a Galois Field of size 2, that is, *binary* [7]. This enables all arithmetic operations to be performed using computationally-inexpensive operations, such as bitwise AND and XOR [7, 28, 44]. To do so, each element in the generator matrix is converted to a $w \times w$ matrix of binary elements, and each entry in the data matrix is converted to a length-$w$ column vector of binary elements. We refer the reader to the works of Bloemer et al. [7], Plank et al. [30], and Zhu et al. [44] for further details. The result of this conversion is an generator matrix of size $rw \times kw$ and a data matrix of size $kw \times d$, each of binary values.

The encoding of a single parity unit $p_i$ using a Galois Field of size $2^w$ (GF($2^w$)) can be viewed as the equation:

$$p_i = \alpha_{i,1}x_1 + \alpha_{i,2}x_2 + \ldots + \alpha_{i,k}x_k,$$

where each $\alpha_{i,j}$ belongs to GF($2^w$). Transitioning this to the corresponding bitmatrix erasure code, we now have

$$p_i = \beta_{i,1}b_1 + \beta_{i,2}b_2 + \ldots + \beta_{i,kw}b_{kw},$$

where each $\beta_{i,j}$ and $b_j$ belongs to GF(2), that is, they are *binary*. This results in products of the form $\beta_{i,j}b_j$ being performed as the bitwise AND between $\beta_{i,j}$ and $b_j$, and the summations being performed as bitwise XOR.

**System-level optimizations.** Many classic techniques for developing high-performance software on CPUs have similarly been used to accelerate erasure codes on CPUs. Examples of these include vectorization [31, 44] and techniques to make the best use of the memory hierarchy [21, 44]. Erasure codes have also been executed on GPUs [10, 19], FPGAs [8], SmartNICs [36, 37], and programmable switches [32]. These works each consider specific features of the target hardware in implementing erasure codes, and, thus, are typically not portable across different classes of hardware. As an example, recent Intel CPUs are equipped with Galois Field New Instructions (GFNI) [1], which provide hardware-accelerated arithmetic over Galois Fields. However, the same functionality is not available on AMD CPUs, which increases the complexity of developing portable high-performance erasure-coding libraries using this functionality.

**Algorithmic optimizations.** In addition to system-level optimizations, many optimizations have been developed that exploit properties specific to bitmatrix erasure codes.

One such technique is to search for generator matrices that achieve the desired level of fault tolerance with as few ones in the matrix as possible [6, 29]. As illustrated above, reducing the number of ones in the generator matrix reduces the number of XORs that must be performed in encoding.

Another algorithmic technique used to accelerate bitmatrix erasure codes is to schedule the XORs performed in encoding parities so as to minimize the total number of XORs performed.

### 2.2 Related work

**Optimized erasure coding.** Significant effort has gone into optimizing other aspects of erasure-coded systems, such as reducing network bandwidth used in decoding [14, 34, 35], scheduling decoding operations to optimize network transmission [18], and efficiently changing the level of redundancy used in an erasure code [16, 17, 42]. These techniques complement our work, as each benefits from using high-performance erasure-coding libraries.

**Performance portability.** A growing body of work focuses on maintaining high performance of applications regardless of the hardware platform (e.g., Kokkos [38] and RAJA [5]). However, implementing erasure codes within such platforms still requires a deep understanding of erasure codes, as well as some basis of understanding how to write high-performance code. In contrast, our approach of leveraging ML libraries to implement erasure-coding libraries requires little understanding of erasure codes or hardware due to the significant similarities between erasure codes and matrix multiplication.

Our proposal differs from these works by adopting ML libraries to implement high-performance erasure codes with little development effort.

## 3 Need for rethinking erasure-coding libraries

While erasure codes are critical to many storage systems, we argue that the current approach to developing optimized erasure-coding libraries leaves much to be desired.

As described in §1 and §2.1, developing optimized erasure-coding libraries is currently challenging because it requires exploiting low-level hardware features *and* knowledge of the mathematical underpinnings of erasure codes. Expertise in these areas typically comes from the disparate domains of computer architecture and information theory, respectively, which makes it difficult to field a team of engineers well-equipped for developing an optimized erasure-coding library.

While the current development process of optimized erasure-coding libraries leaves much to be desired, we argue that it will be even more challenging in the future.

Hardware is becoming increasingly heterogeneous, with GPUs, FPGAs, and ASICs complementing, and at times replacing, CPUs. At the same time, accelerators are gaining better access to storage and network devices [33]. For example, NVIDIA GPUDirect enables GPUs to access storage and network without transferring data to the host CPU [3].

Alongside, and perhaps driving, this increase in accelerators is a growth in "accelerator-native" applications: applications that run primarily on an accelerator, rather than primarily on a host CPU. A popular accelerator-native application is ML training, in which most application logic and state is kept on accelerators, such as GPUs. Similarly, many scientific simulations are increasingly run atop accelerators (e.g., [41]).

The growth of accelerator-native applications also indicates that *much of the data that needs to be erasure coded in future systems will be generated on accelerators.* Consider ML training as an example. Training ML models is a resource- and time-intensive process that often requires hundreds of GPUs [25]. Due to the scale at which training takes place, it is common for nodes to fail [23]. Thus, a common practice to ensure fault tolerance in ML training is to periodically checkpoint the state of training [12, 23]. High-performance checkpointing libraries often leverage in-memory erasure coding across multiple nodes to reduce the time-overhead of writing checkpoints to stable storage [4, 24, 26]. Additionally, prior work has shown that erasure coding can potentially overcome overheads in checkpointing for ML systems [20]. It would be ideal for such applications to be able to perform erasure coding directly on the accelerator on top of which they run, rather than transferring data to the host CPU for erasure coding.

While the rise of accelerator-native applications and fast accelerator-storage datapaths calls for the ability to perform erasure coding on various hardware platforms, developing optimized erasure-coding libraries for this future exacerbates the challenges described above: a developer is now tasked with understanding the architectures of a variety of accelerators.

To usher in future erasure-coded systems in a developer-friendly way, a new approach to developing optimized erasure coding libraries is needed.

## 4 Case for erasure coding via ML libraries

We now make the case that developing erasure-coding libraries using machine learning (ML) libraries offers the potential to overcome the challenges in developing optimized erasure-coding libraries. We summarize our case as follows:

1. ML libraries are heavily optimized to support new hardware features, as well as to support efficient execution on a variety of hardware platforms (§4.1). This alleviates the burden of understanding hardware details from users of ML libraries, and eases portability across platforms.
2. Erasure codes have a structure closely matching a key operation accelerated by ML libraries (§4.2). Thus, one can easily represent erasure codes using ML libraries, and doing so enables one to adopt existing optimizations in ML libraries and their support for hardware heterogeneity.

### 4.1 ML libraries satisfy many of the desiderata of erasure-coding libraries

A desirable property of an erasure-coding library is the ability to achieve high performance on a variety of hardware platforms without requiring expertise in each of these platforms, and to keep pace with hardware as it evolves.

ML libraries are good examples of software platforms structured toward achieving these goals. With the rise of accelerators targeting ML workloads, it has been recognized that it is untenable to develop custom optimizations for each platform. Thus, there has been an increasing focus on developing ML libraries that seamlessly achieve high performance on a variety of hardware platforms. For example, as described previously, ML compilers generate high-performance implementations of key ML operators and tune them for specific hardware backends. This enables one to achieve high performance without having expertise in computer architecture.

Furthermore, ML libraries are frequently updated to best exploit the latest hardware features. Thus, users of ML libraries can expect to continue to achieve high performance on new hardware.

### 4.2 Erasure coding via ML libraries?

We next make the case for why it is indeed possible and potentially fruitful to forgo custom erasure-coding libraries and instead implement erasure codes via ML libraries.

```
for i in range(M):
    for j in range(N):
        for k in range(K):
            C[i, j] += (A[i, k] * B[k, j])
```

**Listing 1.** (Unoptimized) GEMM

```
for i in range(ec_r * ec_w):
    for j in range(ec_d):
        for k in range(ec_k * ec_w):
            C[i, j] ^= (A[i, k] & B[k, j])
```

**Listing 2.** (Unoptimized) bitmatrix erasure code encoding

```
1  A = te.placeholder((M, K), name="A")
2  B = te.placeholder((K, N), name="B")
3  k = te.reduce_axis((0, K), name="k")
4
5  # GEMM
6  te.compute((M, N),
7      lambda i,j: sum(A[i,k] * B[k,j], axis=k))
8
9  # Bitmatrix erasure code
10 xor = te.comm_reducer(lambda i,j: i ^ j, name="xor")
11 te.compute((M, N),
12     lambda i,j: xor(A[i,k] & B[k,j], axis=k))
```

**Listing 3.** Python code for generating a GEMM and a bitmatrix erasure code in TVM

**ML libraries heavily optimize GEMM.** General matrix multiplication (GEMM) is a fundamental operator in many ML libraries due to its wide use in neural networks.

Decades of effort has been devoted to optimizing implementations of GEMMs. Such implementations build atop the naive version in Listing 1 by applying various optimizations, which have led to GEMMs being so well optimized that they are often used as benchmarks for determining the peak achievable performance of a hardware platform [15].

**Similarity between erasure codes and GEMMs.** Recall from §2 that a bitmatrix erasure code is equivalent to a GEMM, except with a bitwise XOR replacing summation and bitwise AND replacing multiplication. Comparing Listings 1 and 2 illustrates the similarity between GEMM and bitmatrix erasure coding. The entire nested looping structure of a bitmatrix erasure code matches that of a GEMM. The only difference comes in the innermost operation.

Due to the similarity between GEMM and bitmatrix erasure coding, a bitmatrix erasure code implemented via an ML library could potentially *automatically exploit the many optimizations performed in these libraries for GEMM.* In particular, we note that most of the optimizations performed in ML libraries for GEMMs target the portions of the code in Listings 1 and 2 that are identical: the nested looping structure. Examples of such optimizations include vectorization, loop reordering, and cache blocking. Thus, *erasure codes implemented via an ML library could immediately adopt many of the optimizations in ML libraries.*

The similarities between Listings 1 and 2 lead to another important conclusion: *bitmatrix erasure codes could be implemented via ML libraries with few additional lines of code.*

## 5 Implementation using TVM

Apache TVM [9] is an open-source compilation framework for optimizing neural networks on a variety of hardware platforms. While TVM also performs cross-kernel optimizations, we focus on its optimization of a single GEMM-like kernel.

TVM takes as input a description of a kernel written in Python using a high-level API called "Tensor Expressions" (represented as `te` in code). Given this specification, TVM performs various autotuning steps.

The process of declaring a kernel and performing the search and generation procedures above is typically carried out via Python in TVM. The kernel generated by autotuning can be exported to a C++ module for later use. Thus, the erasure codes developed through TVM can be readily used by storage systems developed in lower-level languages such as C++.

**Why TVM?** We prototype in TVM for multiple reasons:

(1) As an ML compiler, TVM can generate high performance kernels for multiple hardware platforms. While we focus our evaluation of the TVM-based prototype on CPUs, we consider the cross-platform nature of TVM promising for deploying erasure codes on various platforms.

(2) TVM provides an opportunity to evaluate the benefit of learning-based autotuning in erasure-coding libraries.

**Implementing erasure codes via TVM.** We now describe the implementation of TVM-EC, which leverages TVM to perform bitmatrix erasure coding. To illustrate the simplicity of implementing a bitmatrix erasure code in TVM, Listing 3 compares the Tensor Expressions description for a GEMM in TVM, and that we have added for bitmatrix erasure coding.

In declaring a GEMM, one declares placeholder variables for matrix operands $A$ and $B$ and declares the axis over which a so-called reduction operation (i.e., an operation that reduces multiple elements into a single element, typically via summation) will take place (lines 1–3). One then defines the computation to be performed using a `te.compute` statement. In the case of GEMM, the statement in lines 6 and 7 states that an output of size $M \times N$ is to be generated in which the element at row $i$ and column $j$ is formed by taking the sum of the elementwise multiplication of row $i$ of matrix $A$ and column $j$ of matrix $B$ (each of which contain $K$ elements).

Declaring a bitmatrix erasure code is similar. The only difference comes in in the `te.compute` statement: rather than performing a sum of pairwise multiplications, we wish to perform an XOR of pairwise ANDs. To implement this,

one needs only to declare a new reduction operator that performs the XOR of all elements (line 10), and replace the summation performed in GEMM with this XOR, and the multiplication performed in GEMM with AND (lines 11 and 12). Using these declared statements, one can autotune the bitmatrix erasure code using the same process for autotuning a GEMM.

Overall, our TVM-based prototype *did not require any source-level changes to TVM*; all implementations used existing APIs in TVM, requiring only around 40 lines of code.

One aspect must be considered when using ML libraries within higher-level erasure-coded systems: ML libraries typically expect that operands to GEMM-like calculations be contiguous in memory, whereas higher-level systems may not guarantee this. For example, Jerasure represents the $k$ data units to be encoded as $k$ pointers to separate allocations in memory. We find that performing `memcpy` operations to reorganize these distinct pointers into a contiguous buffer adds considerable time overhead (up to 84% in our experiments).

In most cases, representing the data to be encoded/decoded as a contiguous allocation of memory is natural and can be done easily. For example, encoding is typically performed over fixed-size chunks of data that are smaller than the entire unit they are part of (e.g., 1 MB chunks). The encoder waits for $k$ such chunks to be passed to it before encoding, and keeps each chunk in memory. This memory must be managed by the storage system itself, rather than by the process that passes the data in, to ensure that no chunks are deallocated before the $k$ chunks are encoded. Thus, the encoder must copy data into its own allocation of memory. A system can easily allocate a contiguous region of memory sufficient for hosting $k$ chunks, and copy incoming data chunks to different pointer offsets in this region. The contiguous region of memory can then be passed to the ML library once all $k$ chunks have arrived.

## 6 Evaluation

We now evaluate our approach of implementing erasure codes using ML libraries. The highlights are as follows:

- TVM-EC is up to 1.75× faster than state-of-the-art erasure-coding libraries on CPUs for Reed Solomon codes.
- Our prototypes require only tens of lines of code to implement an erasure code, and TVM-EC in particular enables one to develop using languages that are typically considered more user-friendly (Python) than those used in current optimized erasure-coding libraries (e.g., C, Rust).
- TVM-EC automatically discovers complex optimizations that would otherwise need to be implemented by hand.

### 6.1 Evaluation setup

**Hardware platforms evaluated.** We use an eight-core Intel Xeon D-1548 at 2.0 GHz with 64 GB of memory.

**Prototypes and baselines.** We evaluate TVM-EC on the CPU platform, and compare it to two state-of-the-art custom erasure-coding libraries optimized for CPUs: (1) the work of Uezato [39], a recent approach to optimizing erasure codes that leverages classic techniques from compiler theory alongside the optimizations described in §2.1. This work has been shown to outperform other optimized erasure-coding libraries (e.g., [44]), which themselves outperform popular libraries such as Jerasure [28]. (2) Intel's ISA-L library [2], which is a production-grade erasure-coding library optimized for CPUs. ISA-L does not convert an erasure code to a bitmatrix, and thus provides a competitive baseline showcasing the performance of erasure coding with higher finite field sizes. We note that the relative performance of Uezato's work and ISA-L is different from that reported by Uezato [39]. After discussing with the author, we believe that this can be attributed to a difference in evaluation platform: whereas we evaluate on a server-grade CPU, Uezato [39] evaluates on a MacBook.

**Erasure codes and parameters.** We evaluate our prototype with Reed Solomon codes, the most commonly used erasure code method. We use the parameters $k$, $r$, and $w$ that encompass those used in the prior works to which we compare. We use $k$ of 8–10 with $r$ of 2–4 and $w$ fixed at 8. Each data unit is 128 KB. For the work of Uezato, we evaluate various cache blocking factors, but typically find the performance using a blocking factor of 2 KB to provide the highest performance, and thus report results only for this factor.

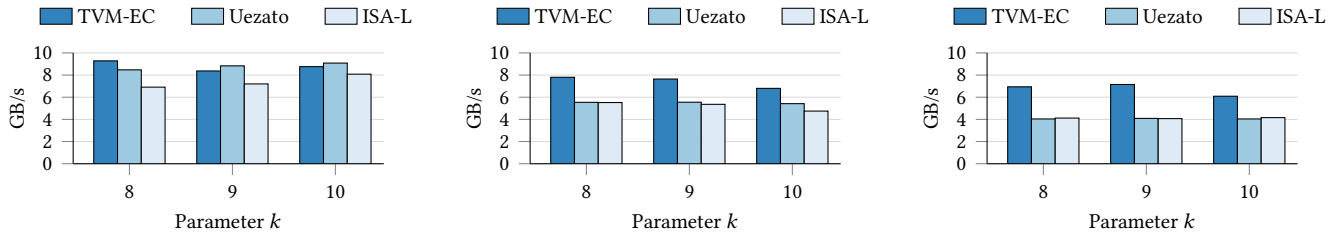
**Measurement setup.** Our prototypes use autotuning provided by TVM to find a high-performing implementation for a particular set of erasure-coding parameters $k$, $r$, and $w$.

TVM-EC uses TVM's learning-based Autoscheduler [43]. This is standard within TVM for achieving high performance among any kernel. TVM-EC tunes for 20000 trials, and uses the best configuration found in final evaluation. We report the mean of 1000 executions of encoding/decoding.

### 6.2 Results

Figure 2 compares the encoding throughput of TVM-EC with Uezato's erasure-coding library [39] and ISA-L [2] for the values of $k$ and $r$ used by Uezato. TVM-EC achieves similar or higher throughput than these custom erasure-coding libraries for all values of $k$ and $r$ considered. In particular, TVM-EC achieves up to 1.75× higher throughput than these custom-built libraries.

**Effect of parameter $r$.** All erasure-coding libraries in Figure 2 achieve lower encoding throughput with higher values of parameter $r$ (i.e., when encoding more parities). Holding $k$ constant, a higher value of $r$ increases the computational intensity of the erasure code. Thus, it takes longer to encode a given amount of data, which results in lower encoding throughput.




**(a)** $r = 2$ **(b)** $r = 3$ **(c)** $r = 4$

**Figure 2.** Encoding throughput in GB/s of TVM-EC and CPU-optimized erasure-coding libraries

We also find that TVM-EC attains its most significant speedups over the baselines with higher values of $r$: whereas the performance of the libraries is closer for $r = 2$, TVM-EC is up to 1.4× and 1.75× faster for parameter $r$ of 3 and 4, respectively. We attribute this improved performance with a higher value of $r$ to the significant optimization of ML libraries for more-computationally-intense GEMMs. With a higher value of parameter $r$, and thus higher intensity, TVM-EC is able to leverage these optimizations to achieve higher performance improvement over custom erasure-coding libraries.

## 7 Discussion

### 7.1 Integration effort

While machine learning libraries such as TVM support various underlying hardware, the interfaces are usually available only in higher-level languages (e.g., Python). However, many storage systems are written in lower-level languages such as C/C++. Integrating an erasure code implemented via machine learning library into a storage system requires efficiently exposing high-level-language APIs to low-level language, which may require non-negligible effort.

Fortunately, in the case of TVM, exposing autotuned erasure coding schedules to C/C++ is inherently supported by the library. However, executing the exported schedule module still requires a C/C++ TVM runtime library and expects the data to be organized in a particular layout. As a consequence, the storage system needs to be modified to adhere to the requirements of the exposed API. A better approach to integration needs to be explored.

### 7.2 Potential limitations

Using ML libraries enables developers to easily apply techniques from parallel programming and GEMM-like optimizations to erasure coding libraries. However, this makes applying erasure coding specific optimization difficult as they cannot be written in a way similar to GEMM. Therefore, we can anticipate that on hardware with few parallel computing resources (e.g. single core, no vectorization), erasure coding libraries via ML libraries may perform worse than manually optimized erasure coding libraries.

Also, it is worth noting that GEMM-like optimizations focus on parallel executions such as multi-core execution, whereas erasure code-specific optimizations do not. Therefore, one potential limitation of erasure code implemented via ML libraries is that they may lead to higher CPU utilization.

## 8 Conclusion and Future Work

Using ML libraries to develop erasure-coding libraries is a promising approach to ease the development and maintenance effort. Based on our experiment results in 6, our prototype TVM-EC gives up to 1.75× performance benefit over state-of-the-art erasure-coding libraries. Moreover, the development effort is far less.

In the future, we plan to include other classes of codes in our prototype, such as local reconstruction codes (LRCs) [14, 35]. Theoretically, all linear codes can be developed via a highly optimized GEMM routine. We plan to perform a more full-pledged evaluation, including measuring the throughput and latency of the prototype for different $r$ and $w$ parameters as well as comparing the CPU utilization of our prototype and other erasure coding libraries. We will also investigate the learning-based tuning performed by TVM and reason about the optimizations it performs on the generated erasure code code. Last but not least, we plan to develop prototypes on various hardware platforms, such as GPUs, and integrate our prototype into real storage systems to verify the practicality and measure the performance on real storage workloads.

## Availability

The codebase for this work is open-sourced at https://github.com/Thesys-lab/tvm-ec.git.

## Acknowledgments

We thank the anonymous reviewers and our shepherd Jingyuan Zhang for their constructive feedback. We thank Cloudlab [11] for the infrastructure support for running experiments. This work was supported in part by a Sloan Foundation Fellowship, and in part by a VMware Systems Research Award.

## References


[1] 2021. Intel Galois Field New Instructions (GFNI) Technology Guide. https://tinyurl.com/35w8pebx. Last accessed 12 December 2022.

[2] 2021. Intel Intelligent Storage Acceleration Library. https://www.intel.com/content/www/us/en/developer/tools/isa-l/overview.html. Last accessed 17 September 2022.

[3] 2024?. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect. Last accessed 28 May 2024.

[4] Quentin Anthony and Donglai Dai. 2021. Evaluating Multi-Level Checkpointing for Distributed Deep Neural Network Training. In *2021 SC Workshops Supplementary Proceedings (SCWS 21)*.

[5] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*.

[6] Mario Blaum and Ron M Roth. 1999. On Lowest Density MDS Codes. *IEEE Transactions on Information Theory* 45, 1 (1999), 46–59.

[7] Johannes Bloemer, Malik Kalfane, Richard Karp, Marek Karpinski, Michael Luby, and David Zuckerman. 1995. *An XOR-Based Erasure-Resilient Coding Scheme.* Technical Report TR-95-048. University of California, Berkeley.

[8] Guoyang Chen, Huiyang Zhou, Xipeng Shen, Josh Gahm, Narayan Venkat, Skip Booth, and John Marshall. 2016. OpenCL-Based Erasure Coding on Heterogeneous Architectures. In *2016 IEEE 27th International Conference on Application-Specific Systems, Architectures and Processors (ASAP 16)*.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

[10] Matthew L Curry, Anthony Skjellum, H Lee Ward, and Ron Brightwell. 2011. Gibraltar: A Reed-Solomon Coding Library for Storage Applications on Programmable Graphics Processors. *Concurrency and Computation: Practice and Experience* 23, 18 (2011), 2477–2495.

[11] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[12] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. Check-N-Run: A Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*.

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP 03)*.

[14] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure Coding in Windows Azure Storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*.

[15] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVIDIA Turing T4 GPU via Microbenchmarking. *arXiv preprint arXiv:1903.07486* (2019).

[16] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, KV Rashmi, and Gregory R Ganger. 2022. Tiger: Disk-Adaptive Redundancy Without Placement Restrictions. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*.

[17] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, KV Rashmi, and Gregory R Ganger. 2020. PACEMAKER: Avoiding HeART Attacks in Storage Clusters with Disk-Adaptive Redundancy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[18] Shiyao Lin, Guowen Gong, Zhirong Shen, Patrick PC Lee, and Jiwu Shu. 2021. Boosting Full-Node Repair in Erasure-Coded Storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*.

[19] Chengjian Liu, Qiang Wang, Xiaowen Chu, and Yiu-Wing Leung. 2018. G-CRS: GPU Accelerated Cauchy Reed-Solomon Coding. *IEEE Transactions on Parallel and Distributed Systems* 29, 7 (2018), 1484–1498.

[20] Kaige Liu, Jack Kosaian, and KV Rashmi. 2021. ECRM: Efficient Fault Tolerance for Recommendation Model Training via Erasure Coding. *arXiv preprint arXiv:2104.01981* (2021).

[21] J. Luo, M. Shrestha, L. Xu, and J. S. Plank. 2013. Efficient Encoding Schedules for XOR-based Erasure Codes. *IEEE Transactions on Computing* (May 2013).

[22] Florence Jessie MacWilliams and Neil James Alexander Sloane. 1977. *The Theory of Error Correcting Codes*. Vol. 16. Elsevier.

[23] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark C Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, et al. 2021. CPR: Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery. In *The Fourth Conference on Systems and Machine Learning (MLSys 21)*.

[24] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. 2010. Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 10)*.

[25] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 21)*.

[26] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. 2019. VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS 19)*.

[27] David A. Patterson, Garth Gibson, and Randy H. Katz. 1988. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 88)*.

[28] J Plank and K Greenan. 2014. Jerasure: A Library in C Facilitating Erasure Coding for Storage Applications. *Univ. Tennessee, Knoxville, TN, USA, Tech. Rep. CS-07–603* (2014).

[29] J. S. Plank. 2008. The RAID-6 Liberation Codes. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*.

[30] J. S. Plank, K. M. Greenan, and E. L. Miller. 2013. *A Complete Treatment of Software Implementations of Finite Field Arithmetic for Erasure Coding Applications.* Technical Report UT-CS-13-717. University of Tennessee.

[31] James S Plank, Kevin M Greenan, and Ethan L Miller. 2013. Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*.

[32] Yi Qiao, Menghao Zhang, Yu Zhou, Xiao Kong, Han Zhang, Mingwei Xu, Jun Bi, and Jilong Wang. 2022. NetEC: Accelerating Erasure Coding Reconstruction With In-Network Aggregation. *IEEE Transactions on Parallel and Distributed Systems* 33, 10 (2022), 2571–2583.

[33] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seung Won Min, Amna Masood, Jeongmin Park, Jinjun Xiong, CJ Newburn, Dmitri Vainbrand, I Chung, et al. 2022. BaM: A Case for Enabling Fine-grain High Throughput GPU-Orchestrated Access to Storage. *arXiv preprint*

*arXiv:2203.04910* (2022).

[34] K. V. Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2014. A Hitchhiker's Guide to Fast and Efficient Data Reconstruction in Erasure-Coded Data Centers. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM 14)*.

[35] Mahesh Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment* 6, 5 (2013).

[36] Haiyang Shi and Xiaoyi Lu. 2019. TriEC: Tripartite Graph Based Erasure Coding NIC Offload. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 19)*.

[37] Haiyang Shi and Xiaoyi Lu. 2020. INEC: Fast and Coherent In-Network Erasure Coding. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 20)*.

[38] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817.

[39] Yuya Uezato. 2021. Accelerating XOR-Based Erasure Coding Using Program Optimization Techniques. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 21)*.

[40] Hakim Weatherspoon and John D Kubiatowicz. 2002. Erasure Coding vs. Replication: A Quantitative Comparison. In *International Workshop on Peer-to-Peer Systems (IPTPS 2002)*.

[41] David B Williams-Young, Wibe A De Jong, Hubertus JJ Van Dam, and Chao Yang. 2020. On the Efficient Evaluation of the Exchange Correlation Potential on Graphics Processing Unit Clusters. *Frontiers in chemistry* 8 (2020), 581058.

[42] Si Wu, Zhirong Shen, and Patrick PC Lee. 2020. Enabling I/O-Efficient Redundancy Transitioning in Erasure-Coded KV Stores via Elastic Reed-Solomon Codes. In *2020 International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 246–255.

[43] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[44] Tianli Zhou and Chao Tian. 2019. Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*.