VectorCDC: Accelerating Data Deduplication with Vector Instructions

Sreeharsha Udayashankar, Abdelrahman Baba,
and Samer Al-Kiswany, *University of Waterloo*

https://www.usenix.org/conference/fast25/presentation/udayashankar

This paper is included in the Proceedings of the
23rd USENIX Conference on File and Storage Technologies.

February 25–27, 2025 • Santa Clara, CA, USA

ISBN 978-1-939133-45-8

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

VectorCDC: Accelerating Data Deduplication with Vector Instructions

Sreeharsha Udayashankar
University of Waterloo
s2udayas@uwaterloo.ca

Abdelrahman Baba
University of Waterloo
ababa@uwaterloo.ca

Samer Al-Kiswany
University of Waterloo
alkiswany@uwaterloo.ca

Abstract

Content-defined Chunking (CDC) algorithms dictate the overall space savings achieved by deduplication systems. However, due to their need to scan each file in its entirety, they are slow and often the main performance bottleneck within data deduplication. This paper presents VectorCDC, a method to accelerate hashless CDC using SSE/AVX CPU instructions. Our evaluation shows that VectorCDC achieves $21-46\times$ higher throughput than existing vector acceleration techniques, without affecting the space savings achieved.

1 Introduction

The amount of data generated and stored on the internet is growing at an explosive rate [1]. Cloud storage providers are racing to support this data growth by using novel storage paradigms [2, 3], deploying distributed file systems [4, 5] and caches [6, 7], using mechanisms such as data deduplication [8, 9], and investing in data protection [10].

Previous studies by Microsoft [11] and EMC [12] show that a large amount of redundancy exists in the data stored on the cloud. Data deduplication [8] is used to conserve storage space by identifying and eliminating this redundant data. Data deduplication consists of four phases [9], of which *data chunking* and *chunk hashing* are the most compute-intensive [8, 13]. In the data chunking phase, incoming data is divided into small chunks, typically of size $1-64$ KB. Numerous data chunking algorithms exist in current literature [14–19] and can be broadly classified into hash-based and hashless algorithms [13]. As chunking occurs whenever new data is uploaded, this phase runs millions of times on the critical path, making it a prime candidate for optimization.

We present VectorCDC, a technique to accelerate data chunking algorithms using vector instructions. Vector instructions [20] are supported by most modern CPUs, and have been previously used to accelerate mathematical operations [21, 22] and multimedia applications [23].

SS-CDC [24] has previously explored accelerating hash-based chunking algorithms using vector instructions. Due to the difficulties of leveraging vector instructions for hash-based chunking, they resort to processing non-adjacent data regions within a single vector operation, leading to relatively small speedups (§3). Unlike SS-CDC, VectorCDC accelerates hashless chunking algorithms. Hashless algorithms run faster than most hash-based ones by avoiding computationally intensive rolling hash functions. However, they may achieve slightly lower space savings in certain data sets.

This paper identifies two phases common to all hashless algorithms; Extreme Byte Searches and Range Scans. We accelerate the search for extreme bytes with a novel *tree-based search*, dividing the scanned region into multiple sub-regions, processing each region using vector instructions, and using a tree-based approach to combine their results. We accelerate range scans with *packed scanning*, packing multiple adjacent bytes into vector registers and comparing them using a single vector operation. Our evaluation (§5) shows that using these methods, VectorCDC achieves $21\times - 46\times$ higher chunking throughput than SS-CDC's approach, without affecting the space savings achieved by hashless algorithms. We have made our code publicly available by integrating it with DedupBench[1] [13].

2 Background

Data deduplication consists of four phases [9]:

- *Data Chunking*: Data is divided into small chunks typically of size $1-64$KB using chunking algorithms.
- *Chunk Hashing and Comparison*: These chunks are hashed using a collision-resistant hashing algorithm such as MurmurHash3 [25] or SHA-256 [26]. Chunk hashes are compared against previously seen hashes to identify duplicate chunks.
- *Metadata Creation*: Metadata needed to reconstruct the original data from stored chunks i.e. *recipes* are created.

[1] https://github.com/UWASL/dedup-bench

- *Metadata and Chunk Storage*: Non-duplicate chunks and recipes to recreate the original data are saved on the storage medium.

Space savings [13] is an important metric in deduplication, representing the overall disk space conserved. It is defined as:

$$Space\ Savings = \frac{Original\ Size - Deduplicated\ Size}{Original\ Size} \times 100 \quad (1)$$

2.1 Data Chunking

Data chunking and chunk hashing are typically the most compute-intensive phases in deduplication [8]. While chunk hashing has been accelerated up to 53× using GPUs [27] and faster hashing algorithms [25, 28], data chunking needs more attention.

Dividing the data into fixed-size chunks is fast, but results in poor space savings on most datasets [17]. Deduplication systems in production instead use Content-Defined Chunking (CDC) algorithms, which divide data into chunks based on its characteristics. Numerous CDC algorithms exist in current literature [14–19, 29] and can be broadly classified into hash-based and hashless algorithms.

Hash-based algorithms [15–17, 19] slide a fixed-size window over the data. When the hash value of the window's contents matches a target mask, they insert a chunk boundary. This creates a new data chunk ranging between the current and previous chunk boundaries. As recomputing the hash each time the window moves is expensive [17], they use rolling hash functions. Note that these hash-based CDC algorithms are only used during the *Data Chunking* phase and do not affect *Chunk Hashing and Comparison*.

Hashless CDC algorithms [14, 18, 29] instead use local minima/maxima to insert chunk boundaries. For example, RAM [18] inserts chunk boundaries when a byte outside the window is at least as large as the maximum valued byte in the window. As these do not involve rolling hashes, they are up to $2-3\times$ faster than most hash-based CDC algorithms. While hashless algorithms may achieve slightly lower space savings on certain datasets (§5.1), they are still used in production systems as the difference is small.

3 Motivation: Hash-Based CDC with AVX

SS-CDC [24] proposed using AVX-512 instructions to accelerate hash-based CDC algorithms. They decouple the rolling hash and boundary detection phases, running the rolling hash on the entire source data to identify boundary candidates in the first phase, and determining boundaries sequentially in the second. This allows both stages to be independently accelerated with AVX instructions. However, many hash-based algorithms such as FastCDC [15] and TTTD [19] use minimum chunk-size skipping to improve throughput i.e., whenever a chunk boundary is found, they skip scanning the next `minimum_chunk_size` bytes. When SS-CDC runs the rolling hash phase on the entire source data, the throughput benefits of minimum chunk size skipping are eliminated.

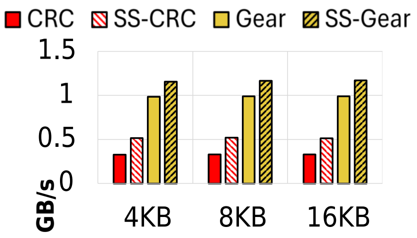


Figure 1: SS-CDC [24] Speedups on Random Data

Secondly, rolling hash algorithms rely on the hash value matching a target value to identify chunk boundaries. The rolling hash value at a particular byte depends on the hash value obtained by rolling until the previous byte, complicating vectorization efforts. SS-CDC [24] overcomes this by using AVX `scatter` and `gather` instructions to load 64 non-adjacent bytes into a single vector register and roll over them using one vector operation. However, these `scatter` and `gather` instructions are expensive [30], limiting performance gains.

Figure 1 shows the chunking throughput obtained by running AVX-512 accelerated versions of CRC (*SS-CRC*) and Gear-based chunking (*SS-Gear*) [24] against their native unaccelerated counterparts. This experiment used randomized data and an Intel Ice Lake machine described in §5. We ran each algorithm with chunk sizes of $4-16$ KB. SS-CRC achieves 0.51 GB/s, a speedup of 1.59× over CRC. Similarly, SS-Gear achieves 1.1 GB/s, a speedup of 1.18× over Gear. These small speedups result from the problems described above.

VectorCDC avoids all these issues by choosing hashless CDC algorithms. VectorCDC accelerates these algorithms using *tree-based search* and *packed scanning* approaches. VectorCDC does not use expensive `scatters` / `gathers` and is compatible with minimum chunk size skipping, resulting in higher throughput and speedups.

4 Design

Hashless CDC algorithms such as AE [14], RAM [18] and MAXP [29] slide windows over the source data to determine chunk boundaries. We identify two common phases across all hashless CDC algorithms: the *Extreme Byte Search* and *Range Scan* phases. We accelerate each of these phases using different AVX-based techniques, discussed in detail below.

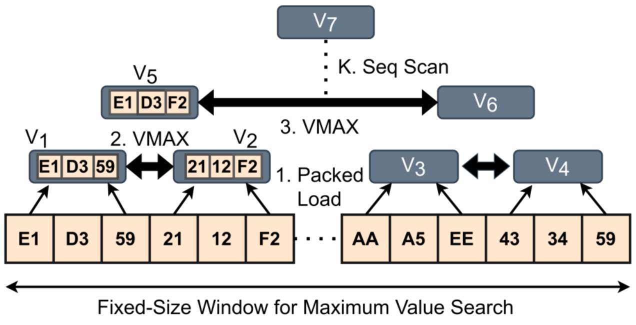

V7
K. Seq Scan
V5
E1 D3 F2
V6
3. VMAX
V1
2. VMAX
V2
E1 D3 59
21 12 F2
1. Packed Load
V3
V4
E1 D3 59 21 12 F2 AA A5 EE 43 34 59
Fixed-Size Window for Maximum Value Search

Figure 2: Accelerating Extreme Byte Search

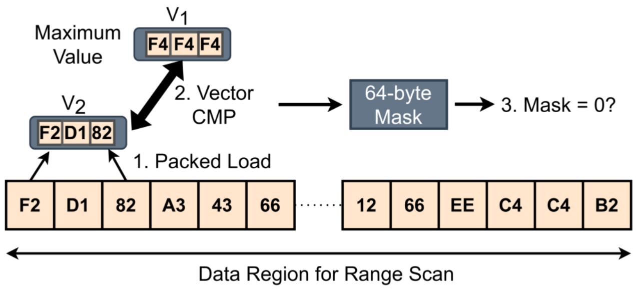


Figure 3: Accelerating Range Scan

4.1 Tree-based Extreme Byte Search

Algorithms such as AE [14], RAM [18] and MAXP [29] all consist of a subsequence that identifies the *extreme byte* (maximum/minimum) in a fixed-size window. The size of this window depends upon the expected average chunk size and can be as large as $4-8$KB. As this phase may need to be performed more than once per chunk, we propose accelerating it using a novel *tree-based search approach*. Let us consider the search for a maximum value using 512-bit AVX instructions (Figure 2).

We first divide the fixed-size window into smaller sub-regions, loading all the bytes into AVX-compatible `m512i` variables in *Step 1*. We load these bytes in a packed fashion i.e. each `m512i` variable contains 64 adjacent bytes. We then use vector `mm512_max` instructions to find the maximum among packed byte pairs (*Step 2*). For instance, among bytes "E1" and "21", byte value "E1" is the maximum. The resulting pairwise maximums are packed into a destination variable (V_5 in the figure).

Step 3 compares the resulting variables V_5 and V_6 from *Step 2* using vector instructions to find the maximum among byte pairs again. We repeat this process, building a tree of `m512i` variables until we are left with a single variable V_7 containing the maximum-valued 64 bytes from across the entire region. We scan these bytes sequentially in *Step K* to determine the maximum valued byte.

4.2 Packed Scanning for Range Scans

Hashless CDC algorithms also consist of a range scan subsequence, where bytes are serially compared against a target value. We propose to accelerate this scanning process using vector instructions. Let us consider a case where we compare bytes sequentially to see if they are greater than or equal to a target value (such as in RAM [18]). Figure 3 shows our proposal to accelerate this using *packed scanning* with AVX-512 instructions.

We first load the maximum value ("F4" in Figure 3) into an AVX-compatible `m512i` variable V_1. We then pack 64 adjacent bytes from the scan region into another `m512i` variable V_2. We compare these 2 registers using `mm512_cmpge` vector compare instructions, which generate a 64-bit integer mask containing the comparison results. If this mask has a value greater than 0, a chunk boundary exists within the scanned 64 bytes. Its exact position is determined using the mask value. If the mask equals 0, no boundary exists within the scanned region and we proceed with loading the next 64 bytes into V_2 to repeat the process.

It is worth noting that our *packed scanning* approach is compatible with minimum chunk size skipping. Unlike SS-CDC's approach, chunk boundary detection and insertion can both occur in *Range Scans* i.e., whenever a chunk boundary is detected, the next `minimum_chunk_size` bytes can be skipped.

4.3 Putting it together: AE and RAM

RAM [18] first scans a fixed-size window at the beginning of the chunk to find a maximum value (Figure 4a). After this, it inserts a chunk boundary at the first byte outside the window which is at least as large as the maximum valued byte. In VectorCDC, RAM is a combination of an *Extreme Byte Search* phase for a maximum value followed by a *Range Scan* phase.

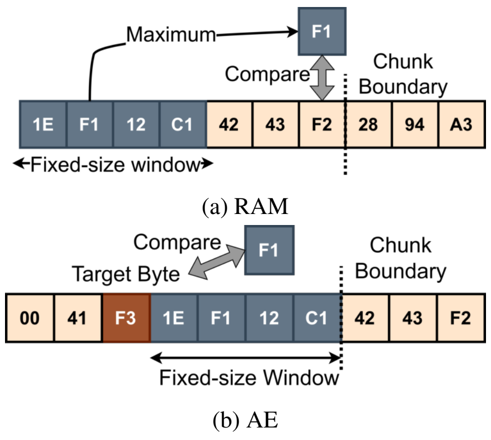


Figure 4: AE and RAM Algorithms

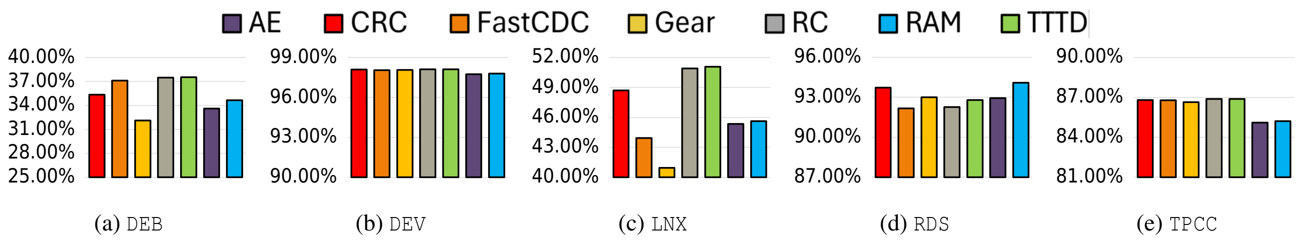


Figure 5: Space Savings with 8KB chunks

Similarly, AE [14] scans for a byte larger than all the bytes before it i.e. a target byte (Figure 4b). Once found, a fixed-size window after this byte is scanned to determine the maximum valued byte within it. If the target byte is larger than the maximum valued byte, a chunk boundary is inserted, else scanning continues for a new target byte. AE can be represented as a combination of multiple *Range Scan* phases each followed by a single *Extreme Byte Search* phase.

Thus, RAM requires only one *Extreme Byte Search* and *Range Scan* phase per chunk while AE may require multiple phases for each chunk. As discussed in §5.2, this causes AE to experience a lower speedup when accelerated with VectorCDC.

Finally, while other hashless algorithms such as MAXP [29] can also be accelerated using VectorCDC, their native versions are slower [14, 18] than AE and RAM and have been omitted from the rest of our paper.

5 Evaluation

This section outlines our efforts to evaluate VectorCDC against the state-of-the-art.

Implementation. We accelerate AE [14] and RAM [18] using VectorCDC with 700 lines of C++ code. We have made our code publicly available with DedupBench [13].

Testbed. We run all our experiments using machines from the Cloudlab [37] platform. We use an AMD EPYC Rome (*c6525-25g* from CloudLab Utah) and an Intel Ice Lake (*sm220u* from CloudLab Wisconsin) for our experiments. The AMD EPYC consists of a 16-core AMD7302P with hyperthreading, 128 GB of RAM, and two Mellanox 25 GBps NICs.

Dataset	Size	Information	XC
DEB	40GB	65 Debian VM Images obtained from the VMware Marketplace [31]	18.98%
DEV	230GB	100 backups of a Rust [32] nightly build server	83.17%
LNX	65GB	160 Linux kernel distributions in TAR format [33]	19.87%
RDS	122GB	100 Redis [34] snapshots with `redis-benchmark` runs	33.54%
TPCC	106GB	25 snapshots of a MySQL [35] VM running TPC-C [36].	37.39%

Table 1: Dataset Information

The Ice Lake consists of two 32-core Xeon Silver 4314 CPUs with hyperthreading, 256 GB of RAM and a 100GBps Mellanox NIC.

Note that all our runs are on the Ice Lake unless otherwise specified. All our results are the averages of 5 runs and the standard deviation was less than 5%.

Alternatives. We evaluate the following hash-based CDC algorithms:

- *CRC:* Native (unaccelerated) version of the CRC-32 chunking algorithm from SS-CDC [24].
- *FastCDC:* Native version of FastCDC [15].
- *Gear:* Native version of the Gear-hash based chunking algorithm [16].
- *RC:* Rabin's chunking algorithm from LBFS [17].
- *SS-CRC / SS-Gear:* AVX-512 versions of CRC and Gear accelerated using SS-CDC [24].
- *TTTD:* Two-Threshold Two-Divisor algorithm [19].

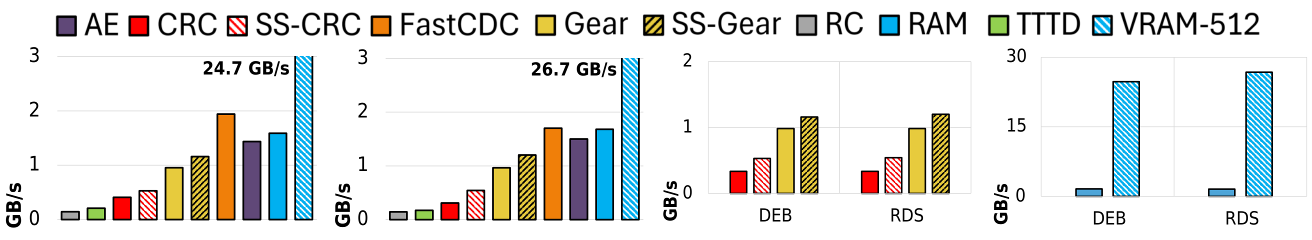
We also evaluate the following hashless CDC algorithms:

- *AE:* Native version of AE [14]. We use `AE-Max`.
- *RAM:* The native Rapid Asymmetric Maximum [18] algorithm.
- *VAE / VRAM*: SSE-128, AVX-256 and AVX-512 versions of AE and RAM accelerated with VectorCDC.

Datasets. Table 1 shows the datasets used within our evaluation as well as the space savings achieved by using fixed-size chunking (*XC*) on them with 8KB chunks . By comparing the space savings achieved by *XC* on these datasets to those achieved by native CDC algorithms (Figures 5a - 5e), we note that the datasets possess varying degrees of byte-shifting. For instance, *XC* achieves a space savings of only 37.39% on the `TPCC` dataset at 8KB while CDC algorithms achieve 86-87%. We have made the `DEB` dataset publicly available [2] [38].

Metrics. We evaluate each alternative's achieved space savings, chunk size distribution, and chunking throughput on all the described datasets.

[2] https://www.kaggle.com/datasets/sreeharshau/vm-deb-fast25



(a) DEB (b) RDS (c) Speedups with SS-CDC [24] (d) Speedup with VectorCDC

Figure 6: Chunking Throughput with AVX-512 instructions and 8KB chunks

5.1 Space Savings

Figures 5a - 5e show the space savings achieved by all alternatives with 8KB chunks. We omit the results for other chunk sizes as the trends were similar.

Hash-based algorithms exhibit space savings values close to each other across datasets. *RAM* and *AE* achieve slightly lower space savings than the best hash-based algorithm on some datasets (Figure 5c). On the other hand, they slightly outperform all hash-based algorithms on other datasets (Figure 5d). Overall, *RAM* and *AE* achieve space savings values within 6% of hash-based algorithms on all datasets and chunk sizes, showing that *hashless algorithms remain competitive with hash-based algorithms for data deduplication*.

Note that *SS-CRC*, *SS-Gear*, *VAE*, and *VRAM* achieve the same space savings as their native counterparts i.e. *vector-acceleration does not impact the space savings achieved by CDC algorithms.* This aligns with the results previously observed for SS-CRC and SS-Gear [24]. To ensure the correctness of our vector-accelerated implementations, we compared their chunk size distributions to those of their native counterparts and verified that they were equal. We omit these results from the paper due to space constraints.

5.2 Chunking Throughput

Figures 6a and 6b show the throughput achieved by all algorithms on DEB and RDS with a chunk size of 8KB. Note that we have cropped the y-axis to 3 GB/s to avoid the figure being skewed by *VRAM*. The results on other datasets and chunk sizes were similar. We have omitted them for clarity.

Throughput Comparison. The fastest among the native hash-based algorithms are *FastCDC* [15], *Gear* [16], and *CRC* [24] achieving 2 GB/s, 0.95 GB/s, and 0.4 GB/s respectively. We have accelerated each of these using SS-CDC; SS-Gear achieves 1.2 GB/s and SS-CRC achieves 0.53 GB/s.

The native hashless algorithms *AE* [14] and *RAM* [18] come in at $1.5-1.6$ GB/s, much faster than most of their hash-based counterparts. *VRAM*, our vector-accelerated *RAM* implementation, achieves 24-26 GB/s, 21× and 46× faster than *SS-GEAR* and *SS-CRC* respectively. The throughputs of all algorithms do not vary significantly across datasets.

Speedup Comparison. Figures 6c and 6d show the speedups achieved by AVX-512 accelerated algorithms over their native counterparts. Accelerating hash-based algorithms using SS-CDC achieves a speedup of $1.2-2\times$. For instance on DEB, *SS-CRC* exhibits a throughput of 0.53 GB/s over *CRC* at 0.4 GB/s. On the other hand, *VRAM* achieves a speedup of 16× over *RAM*. This demonstrates that *AVX instructions can be leveraged far more efficiently to accelerate hashless algorithms compared to hash-based algorithms*.

FastCDC. We did not observe any speedup when accelerating *FastCDC* [15] with SS-CDC [24]. One of the main optimizations used by *FastCDC* is minimum chunk size skipping. However, as noted in §3, decoupling the rolling-hash phase from the boundary identification phase eliminates the throughput benefits of minimum chunk size skipping, nullifying any speedup provided by vector-acceleration. VectorCDC achieves 12× higher throughput than *FastCDC*.

Accelerating AE. We also accelerated AE [14] using VectorCDC. Figure 7 shows the speedup achieved by *VAE* when accelerated using SSE-128/AVX-256 instructions on the DEB dataset with 8KB chunks. The results for other datasets and chunk sizes were similar. We see that *VAE-128* and *VAE-256* only achieve 2.9 GB/s and 5.2 GB/s, speedups of 2× and 3.45× over *AE* while *VRAM-128* and *VRAM-256* achieve speedups of 8.7× and 12.3× over *RAM*. However, note that *VAE* is still faster than every other CDC algorithm.

Unlike *RAM*, the rolling window in *AE* has a target condition depending on all the bytes before it (§4.3). The byte at the head of the window must be greater than all the bytes before it. The target byte is found using a *Range Scan*. Each time such a byte is discovered, *Extreme Byte Search* is run to find the maximum value in the window following the target byte. To insert a chunk boundary, this maximum valued byte must be smaller than the target byte. If not, the *Range Scan* begins again. Thus, *AE* requires multiple iterations of *Range Scan* and *Extreme Byte Search* per chunk while *RAM* only requires one iteration of each, causing the lower speedup seen in Figure 7. This demonstrates that *RAM is inherently more vector friendly than AE.*

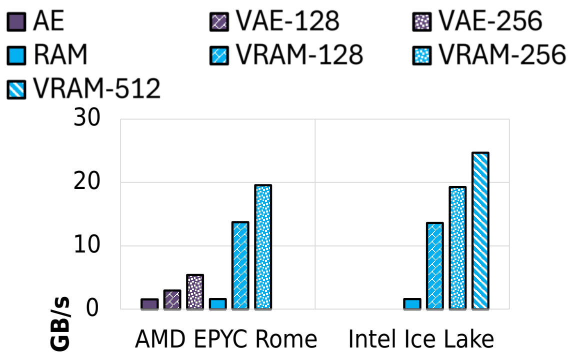


Figure 7: VectorCDC on `DEB` with SSE-128/AVX-256 instructions and 8KB chunks

5.2.1 Speedups with AVX-256 and SSE-128 instructions

AVX-512 instructions are currently only supported by a handful of Intel CPUs. A large number of Intel and AMD CPUs do not support AVX-512 but support SSE-128 and AVX-256 instructions. While §4 discusses VectorCDC's design using AVX-512 instructions, the same methods can be applied to AVX-256 and SSE-128 instructions as well. Figure 7 shows the throughput achieved by *VRAM* implemented using these instructions (*VRAM-128* and *VRAM-256*). We use the `DEB` dataset and 8KB chunks for this experiment, running it on both Intel and AMD machines.

VRAM achieves similar throughputs on both platforms, achieving 13.3 GB/s and 19.5 GB/s with SSE-128 and AVX-256 instructions respectively. This demonstrates that *VectorCDC is compatible with a large range of CPUs while retaining its throughput benefits over hash-based AVX algorithms.*

5.2.2 Throughput breakdown

Figure 8 shows the individual impact of accelerating *Extreme Byte Search* and *Range Scan* in *VRAM* using SSE-128 instructions. We use the `DEB` and `LNX` datasets for this experiment and run *VRAM* with an 8KB chunk size. *VRAM-EBS* represents RAM running with only *Extreme Byte Search* acceleration while *VRAM-128* accelerates both phases.

In `DEB`, we see that *VRAM-EBS* achieves a throughput of 10 GB/s. Accelerating *Range Scan* provides an additional speedup of 3 GB/s. On the other hand, *VRAM-EBS* only achieves 5.7 GB/s on `LNX` while accelerating *Range Scan* provides an additional speedup of 11 GB/s.

This is related to the datasets' characteristics and the *RAM* algorithm. On average, *RAM* finds chunk boundaries faster on `DEB` than `LNX` once the *Extreme Byte Search* is complete. *RAM*'s actual average chunk size on `DEB` is 1KB smaller than that on `LNX`. Thus, *RAM* spends more time in *Range Scan* on `LNX` than `DEB`, explaining Figure 8. Thus, *accelerating both phases using vector instructions is crucial to performance, as the impact of each phase depends on dataset characteristics.*

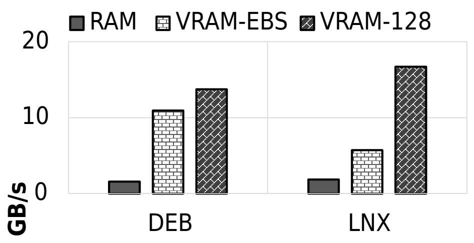


Figure 8: *VRAM* Throughput Breakdown with SSE-128

6 Related Work

Deduplication optimizations. Several other efforts exist to optimize the remaining phases of data deduplication. StoreGPU [39] accelerates chunk hash computation using GPUs, SiLo [40] and Sparse Indexing [41] target hash comparison, and HYDRAStor [42] targets chunk and metadata storage. These are orthogonal to our efforts as we accelerate the data chunking phase.

Chunking optimizations. RapidCDC [43] uses chunk locality to accelerate chunking throughput. MUCH [44] and P-Dedupe [45] use multiple threads to accelerate chunking. All these techniques implement their optimizations on top of existing CDC algorithms and VectorCDC is compatible with all of them. Our previous work [46] that examines the impact of low-entropy on CDC algorithms is orthogonal as vector-acceleration does not impact algorithm characteristics.

Secure deduplication systems. Several efforts build end-to-end deduplication systems for encrypted data [47]. They mainly target encryption schemes [48, 49] for the underlying data or focus on reducing attacks on the system [50]. VectorCDC is compatible with all these approaches.

7 Conclusion

We present VectorCDC, a methodology for accelerating content-defined chunking using vector instructions. VectorCDC avoids the pitfalls of previous work that accelerates CDC algorithms by choosing hashless CDC instead. VectorCDC accelerates these algorithms using novel *tree-based search* and *packed scanning* methods. Our evaluation shows that VectorCDC achieves $21-46\times$ higher throughput than existing AVX-based CDC techniques. We have made our code publicly available by integrating it with DedupBench [13].

8 Acknowledgments

We thank the anonymous reviewers and our shepherd, Avani Wildani, for their feedback. We thank Lori Paniak for helping us enable our experiments. An NSERC Discovery Grant and an Acronis Research Grant supported this work. Sreeharsha is supported by an Ontario Graduate Scholarship.

A Artifact Appendix

Abstract

We have made the code for *VRAM-128*, *VRAM-256* and *VRAM-512* publicly available on GitHub as a part of DedupBench[3] [13].

Scope

DedupBench [13] allows for the quick and easy comparison of numerous CDC algorithms on any dataset. It reports metrics such as the deduplication space savings and chunking throughput for each algorithm. Using DedupBench, the following claims in our paper can be verified:

- **Space Savings:** Accelerating hashless algorithms with VectorCDC does not affect the space savings they exhibit.
- **Chunking Throughput:** *VRAM*, powered by VectorCDC, achieves 16×–42× higher chunking throughput than alternative CDC algorithms. *VRAM-512* achieves a 16× speedup over *RAM*.

Contents

We have bundled the following CDC algorithms into the DedupBench repository:

- *AE:* The hashless Asymmetric Extremum [14] algorithm.
- *CRC:* The native CRC-32 based chunking algorithm from SS-CDC [24].
- *FastCDC:* FastCDC [15] with chunk size normalization.
- *Gear:* Gear-based chunking [16].
- *Rabin:* Rabin's chunking algorithm from LBFS [17].
- *RAM:* Rapid Asymmetric Maximum [18] algorithm.
- *SeqCDC:* The hashless SeqCDC [51] algorithm.
- *TTTD:* Two-Threshold Two-Divisor Algorithm [19].
- *VRAM:* SSE-128, AVX-256 and AVX-512 versions of *VRAM*.

For ease of use, we have also provided scripts to run all algorithms on any user-defined dataset and plot graphs. The repository README file contains further details and usage instructions.

[3] https://github.com/UWASL/dedup-bench

Hosting

The code for *VRAM* is hosted on Github within the DedupBench [13] public repository. It can be obtained from the `main` branch via the *git* [52] framework. Commit `17c5209` or later contains all the code used within this paper.

The `DEB` dataset used in our evaluation is publicly hosted on Kaggle [4] [38]. It can be downloaded and used with DedupBench to obtain some of the results from our paper.

Requirements

The machines used to develop DedupBench were obtained from CloudLab [37]. The details can be found in §5.

All of the native (unaccelerated) CDC algorithms in DedupBench are universally compatible with all processors. *VRAM-128*, *VRAM-256* and *VRAM-512* require CPUs with SSE-128, AVX-256 and AVX-512 instruction set support respectively.

References

[1] Statista. Worldwide data created from 2010 to 2025, 2024.

[2] Mark Carlson, Alan Yoder, Leah Schoeb, Don Deel, Carlos Pratt, Chris Lionetti, and Doug Voigt. Software Defined Storage. *Storage Networking Industry Association Working Draft*, pages 20–24, 2014.

[3] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[4] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. Ieee, 2010.

[5] Sage Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*, pages 307–320, 2006.

[6] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, 2004.

[7] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. TAO: Facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[4] https://www.kaggle.com/datasets/sreeharshau/vm-deb-fast25

[8] Dutch T Meyer and William J Bolosky. A study of practical deduplication. *ACM Transactions on Storage (ToS)*, 7(4):1–20, 2012.

[9] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[10] Deyan Chen and Hong Zhao. Data security and privacy protection issues in cloud computing. In *2012 International Conference on Computer Science and Electronics Engineering*, volume 1, pages 647–651. IEEE, 2012.

[11] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary Data Deduplication — Large scale study and system design. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 285–296, 2012.

[12] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *USENIX Conference on File and Storage Technologies (FAST)*, volume 12, pages 4–4, 2012.

[13] Alan Liu, Abdelrahman Baba, Sreeharsha Udayashankar, and Samer Al-Kiswany. DedupBench: A Benchmarking Tool for Data Chunking Techniques. In *2023 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 469–474. IEEE, 2023.

[14] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1337–1345. IEEE, 2015.

[15] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 101–114, 2016.

[16] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation*, 79:258–272, 2014. Special Issue: Performance 2014.

[17] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 174–187, 2001.

[18] Ryan NS Widodo, Hyotaek Lim, and Mohammed Atiquzzaman. A new content-defined chunking algorithm for data deduplication in cloud storage. *Future Generation Computer Systems*, 71:145–156, 2017.

[19] Kave Eshghi and Hsiu Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30(2005), 2005.

[20] James E Smith, Greg Faanes, and Rabin Sugumar. Vector instruction set support for conditional operations. *ACM SIGARCH Computer Architecture News*, 28(2):260–269, 2000.

[21] Somaia A Hassan, Mountasser MM Mahmoud, AM Hemeida, and Mahmoud A Saber. Effective implementation of matrix–vector multiplication on Intel's AVX multicore processor. *Computer Languages, Systems & Structures*, 51:158–175, 2018.

[22] Shay Gueron and Vlad Krasnov. Fast quicksort implementation using AVX instructions. *The Computer Journal*, 59(1):83–90, 2016.

[23] Robert L Bocchino Jr and Vikram S Adve. Vector LLVA: a virtual vector instruction set for media processing. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 46–56, 2006.

[24] Fan Ni, Xing Lin, and Song Jiang. SS-CDC: A two-stage parallel content-defined chunking for deduplicating backup storage. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 86–96, 2019.

[25] Jingwei Li, Zuoru Yang, Yanjing Ren, Patrick PC Lee, and Xiaosong Zhang. Balancing storage efficiency and data confidentiality with tunable encrypted deduplication. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[26] Dian Rachmawati, JT Tarigan, and ABC Ginting. A comparative study of Message Digest 5 (MD5) and SHA256 algorithm. In *Journal of Physics: Conference Series*, volume 978, page 012116. IOP Publishing, 2018.

[27] Kiatchumpol Suttisirikul and Putchong Uthayopas. Accelerating the cloud backup using GPU based data deduplication. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 766–769. IEEE, 2012.

[28] Chunlin Song, Xianzhang Chen, Duo Liu, Jiali Li, Yujuan Tan, and Ao Ren. Optimizing the Performance of Consistency-Aware Deduplication Using Persistent Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[29] Nikolaj Bjørner, Andreas Blass, and Yuri Gurevich. Content-dependent chunking for differential compression, the local maximum approach. *Journal of Computer and System Sciences*, 76(3-4):154–203, 2010.

[30] Patrick Lavin, Jeffrey Young, Richard Vuduc, Jason Riedy, Aaron Vose, and Daniel Ernst. Evaluating Gather and Scatter Performance on CPUs and GPUs. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '20, page 209–222, New York, NY, USA, 2021. Association for Computing Machinery.

[31] VMWare. VMWare marketplace. https://marketplace.cloud.vmware.com/services, 2023.

[32] Rust. GitHub - rust-lang/rust: Empowering everyone to build reliable and efficient software. https://github.com/rust-lang/rust, 2023.

[33] Linux. The Linux Kernel Archives. https://www.kernel.org/, 2023.

[34] Redis. Redis. https://redis.io/, 2023.

[35] MySQL. MySQL. https://www.mysql.com/, 2023.

[36] Transaction Processing Council. TPC-C Overview. https://www.tpc.org/tpcc/detail5.asp, 2023.

[37] 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. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[38] Sreeharsha Udayashankar, Abdelrahman Baba, and Samer Al-Kiswany. VM Images for Deduplication. https://www.kaggle.com/dsv/10561721, 2025.

[39] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, and Matei Ripeanu. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, page 165–174, New York, NY, USA, 2008. Association for Computing Machinery.

[40] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. Similarity and Locality Based Indexing for High Performance Data Deduplication. *IEEE Transactions on Computers*, 64(4):1162–1176, 2015.

[41] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *USENIX Conference on File and Storage Technologies (FAST)*, volume 9, pages 111–123, 2009.

[42] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: A scalable secondary storage. In *USENIX Conference on File and Storage Technologies (FAST)*, volume 9, pages 197–210, 2009.

[43] Fan Ni and Song Jiang. RapidCDC: Leveraging Duplicate Locality to Accelerate Chunking in CDC-Based Deduplication Systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 220–232, New York, NY, USA, 2019. Association for Computing Machinery.

[44] Youjip Won, Kyeongyeol Lim, and Jaehong Min. MUCH: Multithreaded Content-Based File Chunking. *IEEE Transactions on Computers*, 64(5):1375–1388, 2015.

[45] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Zhongtao Wang. P-Dedupe: Exploiting Parallelism in Data Deduplication System. In *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*, pages 338–347, 2012.

[46] Mu'men Al Jarah, Sreeharsha Udayashankar, Abdelrahman Baba, and Samer Al-Kiswany. The Impact of Low-Entropy on Chunking Techniques for Data Deduplication. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pages 134–140, 2024.

[47] Youngjoo Shin, Dongyoung Koo, and Junbeom Hur. A Survey of Secure Data Deduplication Schemes for Cloud Storage Systems. *ACM Computing Surveys*, 49(4), Jan 2017.

[48] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 296–312. Springer, 2013.

[49] Jian Liu, N. Asokan, and Benny Pinkas. Secure Deduplication of Encrypted Data without Additional Independent Servers. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 874–885, New York, NY, USA, 2015. Association for Computing Machinery.

[50] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side Channels in Cloud Services: Deduplication in Cloud Storage. *IEEE Security and Privacy*, 8(6):40–47, 2010.

[51] Sreeharsha Udayashankar, Abdelrahman Baba, and Samer Al-Kiswany. SeqCDC: Hashless Content-Defined Chunking for Data Deduplication. In *Proceedings of the 25th International Middleware Conference*, pages 292–298, 2024.

[52] Diomidis Spinellis. Git. *IEEE software*, 29(3):100–101, 2012.