Multi-layer Caching and Parallel Streaming for Large Scale Cloud Optimized Point Cloud Data Visualization using WebGPU

Pravin Poudel
Utah State University
USA

Will Usher
Luminary Cloud
USA

Steve Petruzza
Utah State University
USA

***Abstract*—As LiDAR sensors become more precise and widely used, effectively managing and rendering large amounts of point data is becoming increasingly difficult. Available web based solutions for large scale point cloud visualization do not take advantage of modern cloud optimized data formats which are well suited for parallel and efficient data streaming. Furthermore, current web visualization tools do not take advantage of new capabilities like persistent file caching, losing their data when a tab/window is closed. In this work, we present the first web based viewer for point cloud data using a multi-layer cache system that allows to store data using both file and memory storage directly from a browser. Furthermore, we also introduce the first open source viewer for the cloud optimized point cloud (COPC) data format employing a parallel workflow using WebGPU to stream, process and render point cloud data. Experimental studies of the multi-layer cache system demonstrates to provide the best performance in different configurations, including offline data visualization.**

***Index Terms*—web visualization, cloud optimized data formats, point cloud**

I. Introduction

Point cloud data is employed in several domains, such as surveying and geo-mapping, city planning, computer vision and autonomous vehicle navigation, and processing. In these domains, extracting the required data and rendering it in real time is critical. As the use and precision of advanced depth sensors for collecting 3D data increases, so does the demand for efficient management of datasets to enable interactive visualization of point clouds. Similar to numerous visualization tools, the web provides improved accessibility, and modern browsers are facilitating seamless analysis and visualization of data, comparable to traditional standalone applications.

However, making large point cloud data readily available for transfer from the cloud can be challenging due to its unstructured nature and large size. While LAS/LAZ are the standard format in LiDAR, other formats like Potree [18] and Entwine Point Tile (EPT) divide the data into many small files to offer better flexibility and loading performance, but are not cloud friendly (i.e., requiring reading too many files). The Cloud Optimized Point Cloud (COPC) [10] has recently emerged as an alternative file format that allows for the storage of spatially clustered point data organized in an octree using a single file. This allows for incremental partial reads over HTTP and streaming a chunk of data pertaining to specific nodes from a large dataset. The only viewer for this promising cloud-optimized format is implemented with WebGL, and unfortunately it is not open source.

Interactively rendering point cloud data in real-time is challenging due to their large and complex nature, which requires parsing even after fetching. However, by temporarily storing parsed and processed point cloud data in a cache, the renderer can quickly retrieve the point data without the need for redundant fetching and processing, thus facilitating interactive rendering. Caching data structures built using the memory available to the browser are cleared automatically when closed. However recent APIs, such as Origin Private File System (OPFS), now finally allow browser applications to store data in persistent caches on disk. This not only enables applications to limit the data transferred across different sessions through caching, but also enables offline data visualization.

The contributions of this work are:

1) the first open-source viewer[1] of COPC data implementing a parallel workflow leveraging WebGPU [3] to efficiently stream, render, and store data asynchronously;
2) an implementation of a multi-layer caching scheme that leverages both non-persistent and persistent caching (using files, i.e., the first one on a browser), allowing users to replace slower remote data streaming with local reads even after a browser reload; and
3) experimental study of the performance of the multi-layer cache system including the use of a parallel pre-fetching strategy.

II. Background

Point cloud data contains positional information, intensity, and metadata of the scanner, among other details. Large-scale point data can pose challenges to memory usage and interactive rendering speed. Hierarchical datasets are a solution, as they reduce memory usage, improve rendering speed, and enable the handling of large datasets that cannot fit in memory at once. The first multiresolution rendering system for point data, QSplat, was developed by Rusinkiewicz and Levoy [17]. The system used a hierarchy of bounding spheres to gradually load the scanned model and utilized a binary tree to divide

[1] https://github.com/ComputingElevatedLab/WGPU-COPC-Viewer

the space along its longest axes, as opposed to the commonly used Octree. Gobbetti et al [6] expanded on this work by developing a GPU-friendly layered point cloud structure that stored a subsample of the point cloud in each node of the hierarchy and used a binary tree to split the data along its longest axis.

LAS/LAZ is a commonly-used standard format for storing point cloud data. However, its large file size can lead to inefficiencies in handling the data, particularly when rendering or querying, as these operations require loading the entire file into memory first. Potree and EPT are hierarchical point cloud data files that support out-of-core and LOD-based rendering [19] by enabling the loading and unloading of different data sections as needed. Plasio [23], PointCloudViz [16], and LidarView [5] are WebGL [7] based LAS/LAZ point cloud viewers which do not support hierarchical acceleration structure. Primarily designed to handle the hierarchical Potree file format, Potree is based on Three.js [1] and has the capability to render point clouds containing millions of points in real-time.

Various data organization libraries and tools, such as Entwine [9], [22], Potree Converter [18], and PDAL [2], are available to arrange LAS/LAZ point cloud data into an octree-based hierarchical structure to enhance rendering performance. Although these file formats give better rendering performance, they are not streaming cloud-friendly. In particular, these formats require data to be broken down into hundreds to millions of small files making it challenging to access and manage data when stored in the cloud.

HOBU Inc [9] addressed the issue of increased network latency and I/O overhead when accessing previous data files from the cloud in earlier hierarchical data files by developing the Cloud Optimized Point Cloud (COPC) in 2021. COPC overcomes the limitation of storing point cloud data in multiple files by storing all the hierarchical data in a single file and enables efficient and streamlined access to the file. The COPC format is essentially a LAZ file that includes indexes to access structured point data for each part of the dataset, allowing for the streaming of only the necessary data. COPC files can be supported by any LAZ viewer since it is itself a LAZ file.

For example, Potree [18] supports LAZ but has no dedicated support for COPC, which is used as a LAZ file. Potree achieves improved performance primarily due to the use of potree files, which are tiles of point clouds that are generated by combining nodes based on a hierarchical step size parameter. This hierarchical structure allows Potree to efficiently stream and visualize large point cloud datasets by loading only the tiles needed for the current view, rather than the entire dataset [21]. However, when using COPC files Potree would load the entire file into memory as a LAZ file.

HOBU Inc have also created a WebGL-based COPC viewer [8]. However, this viewer is closed source and it does not take advantage of any caching mechanism and thus must repeatedly reload regions of the data from the server that come back into view. This repeated reloading impacts the responsiveness of the system on slower connections and incurs additional bandwidth costs to the application. Our work presents the first open-source COPC viewer that supports parallel data fetching and parsing, multi-level caching, and prefetching as well as a WebGPU out-of-core point cloud renderer.

Prefetching is a widely used optimization technique, where data is fetched and cached from remote sources in parallel to the main program's execution. The purpose of prefetching is to hide data access latency in a system [13]. Most prefetching strategies are history-based predicting future data based on usage patterns or history [11], [12]. There have been applications of unsupervised machine learning to know the block of data that needs to be prefetched from the source [4]. For point cloud data, Schütz et al. [20] implemented a prefetching technique that fetched four points near the memory of each point fetched for unstructured point cloud data. The evaluation showed that this approach resulted in prefetching being up to three times faster compared to situations where prefetched options were not used. In this work we use a simple pre-fetching strategy where we fetch points close to the current view at a coarser resolution. This strategy is implemented only to test the multi-layer caching system and is not compared to other existing pre-fetching strategies.

It is crucial to emphasize that the contributions of this work do not focus on point cloud rendering, and therefore, we do not provide a qualitative comparison of rendering results. However, it is worth noting that systems utilizing different data formats, such as Potree [18] and advanced rendering pipelines like UnityPIC [24], can benefit from the contributions of this work by enhancing their data access strategy through file storage on the browser and implementing multi-layer caching. To be noted also that this is the first work leveraging Origin Private File System (OPFS) on a browser for storing files, this is a new feature just recently available on browsers.

III. Implementation

A fundamental challenge in large-scale point cloud rendering is retrieving large data in real-time while keeping the application interactive. To achieve efficient parallel data streaming, we only fetch the necessary data that are in the current user view and then start pre-fetching other data. Data is fetched in parallel using web workers [14], which enable multiple instances of JavaScript code to run independently in a global context separate from the main thread window context. This approach allows us to concurrently perform data reading and parsing operations, thereby enhancing performance. To ensure synchronization between the worker threads, we wrap each worker instance within a promise. This allows us to synchronize all threads by waiting for all promises to resolve. The COPC format's hierarchical octree data structure, along with the information about the size and file offset of each data chunk provided by the format, allows workers to independently and in parallel fetch the necessary data using *range queries*.

The rendering process in our system utilizes a Hierarchical Level of Detail (HLOD) technique. This means that the root node of the tree is always rendered, and progressively lower-level nodes in the octree are added to the scene based on the quality required for each region. This enables the renderer

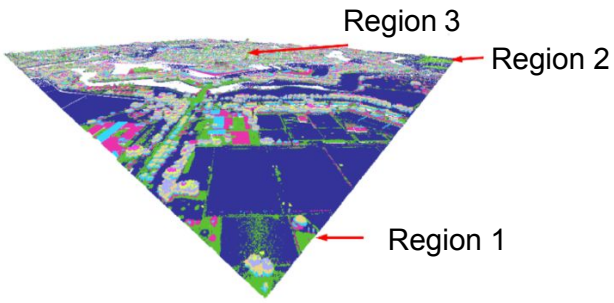

Fig. 1: Our system performs view dependent LOD rendering. Region 1 near the camera is detailed (using high resolution data), while Regions 2 and 3 further away are rendered using a coarser data resolution. Points are colored based on elevation.

to display varying levels of detail based on the current view (see Figure 1). There are three main types of LOD techniques available: hierarchical, continuous, and discrete. In our renderer, we have developed a hierarchical LOD renderer where a change in resolution is gradual, as more and more points are added continuously based on the level of resolution required. We make use of the COPC header information to traverse the tree efficiently. The COPC header provides information about each node's geometry and the number of points in it, allow us to skip nodes outside the region of interest.

A. Caching Strategies

To improve the performance of our point cloud viewer, we integrated a hash map-based LRU Cache in JavaScript. This enables us to store and retrieve point data efficiently using key-value pairs. By using an LRU update strategy, we prioritized frequently accessed data and removed the least recently used data once the cache reached its capacity. While the implementation of an LRU Cache is a commonly used technique, it was critical for achieving our performance goals in our open-source COPC renderer platform. In contrast, the proprietary COPC viewer does not appear to utilize a cache.

It is important to note that the cache used here is non-persistent, meaning it can only temporarily hold data in memory (i.e., until a tab, a window or a session is terminated). On the other hand, a persistent cache provides data durability and fault tolerance, allowing for the safe storage and retrieval of data, even in the event of system crashes or unexpected failures.

We utilize an Origin Private File System (OPFS) to maintain persistent cache data, where datasets are stored. We create individual binary files for each octree node to align with both the COPC structure and buffers in WebGPU, allowing for efficient data access and updates for each node. This strategy also avoids need for additional octree traversal. By adhering to the COPC format, we can access data for each node effectively during point cloud rendering by simply reading the corresponding file from the disk.

File system resources are also limited, and thus we must balance between ensuring relevant data is present in the cache and its size on disk. To achieve this, we implement an LRU updating policy for the persistent cache as well, using a map stored in a file (i.e., *persistent map*) which is expected to be updated whenever a file is stored in or removed from the client's OPFS. Files are evicted from OPFS when the number of cached nodes exceeds the maximum allowed.

To avoid the overhead caused by File I/O during frequent updates of the persistent map during every file read or deletion, we implemented a non-persistent LRU cache as a *staging map* (i.e., key is the file name, value is the last time it was used, entries are stored in order). This allows the *persistent map* file to be updated efficiently in memory to reduce file system access. At the program start, the staging map is initialized with the state of the persistent map. We implemented a throttle system to periodically update the *persistant map*. This ensures that the *staging map* has sufficient time to process and consolidate any changes before writing them to the *persistent map* file (i.e., the two caches are always in synch).

Based on the camera's position and orientation, we traverse the tree and identify the list of nodes that require rendering. We then verify if any of these nodes exist in the GPU memory in the current render pass. For new nodes that are not in GPU memory, we first check if they are present in the LRU non-persistent cache. If the nodes are not found in the LRU non-persistent cache, we then check the persistent cache before resorting to a network request to retrieve the nodes from a remotely hosted file(see Figure 2).

B. Improving Caching Performance

We apply a number of techniques to improve cache latency and reduce the size of data that must be retrieved.

a) Pre-fetching strategy: Our pre-fetching strategy is based on the use of frustum culling, to reduce the number of rendered points and to prefetch only visible or nearly visible nodes. We check each node's bounding box against the frustum to see if it is visible. However, our goal is not to only include the visible points, but to also guarantee the existence of a coarser level of data in the scene for nodes that may soon become visible. Prefetching this data ensures that when the camera moves some data will be immediately available for display. The coarser node to be prefetched is defined as the level one coarser from when nodes start being culled by frustum culling. This coarse node will be prefetched to provide data for the nodes that have been culled. Frustum culling efficiently identifies coarser level nodes to prefetch in our system, ensuring that we always have points at a certain specified coarser level.

For each node that needs to be prefetched, we check if it is in the nonpersistent or persistent cache. If the node is in the nonpersistent cache, it has already been loaded and is available. If the node is in the persistent cache we read the binary file and load it into the nonpersistent cache. If the node is not in either cache, it must be fetched asynchronously from the remote server and is place in both caches. The prefetching follows the same strategy as fetching with the difference that we do not create a buffer but rather only make data available in the cache. Prefetching is done between the end of the fetching

operation and before the start of another traversal and fetching operation.

b) Screen Space error: To further optimize the point cloud rendering process and improve performance, the decision to traverse deeper into the octree is made by considering both frustum culling and screen space error. Screen space error involves calculating the projected radius of the octree node in screen space based on its distance from the camera and camera orientation. This projected radius is compared to a threshold value, which is based on the desired balance between visual quality and rendering performance, to determine whether the node should be rendered or not.

c) Aborting Stale Asynchronous Requests: One potential issue when fetching nodes asynchronously is that, when the camera moves rapidly, it is possible to flood the system with a set of requests that quickly become irrelevant because they have moved out of view again by the time they are loaded. To address this issue, we used the *AbortController*, which allows passing a signal to the running operation that needs to be canceled. Each time the event is fired to traverse the octree and fetch new data, a new instance of AbortController is created and a previously existing instance signal is set to aborted. This signal is checked in the previous operation, if it is not finished we wait for the current batch of threads to finish the operation and checks if the abort signal was sent from outside the function. If the signal was external we terminate the operation early, freeing up the thread for use by another operation. We apply the same abort strategy to prefetching operations as well.

d) Buffer Memory Management: Point cloud data sets can be quite large, and thus we must ensure that our renderer stays within the memory limits of the end user's GPU. This is especially a concern in WebGPU, where GPU buffer objects that go out of scope do not release their corresponding GPU memory until the JavaScript garbage collector runs and destroys them. To reduce our memory footprint we track the buffers in use by the application and immediately destroy them when they are no longer needed, instead of waiting for the garbage collector. Buffers are also cached and re-used if possible in multiple render passes, to reduce the number of buffers created and destroyed to improve performance.

IV. Evaluation

We performed an experimental study to evaluate the performance of the system using different caching strategies: no caching; LRU non-persistent cache; and LRU persistent and non-persistent cache. As our work focuses on improving caching and load times, we only report results on the time to retrieve the data to be rendered for our out-of-core renderer. The experiments are run in Chrome Canary 115.0.5741.0 on a laptop with a 2.3 GHz i7-12700H processor, 32 GB DDR5 RAM and a 24GB Geforce RTX 3070 Ti GPU.

We report results on two LiDAR datasets: (i) a dataset captured in Logan (UT) containing 184,510,697 data points. The original LAZ (compressed LAS) file was sourced from Open Topography [15]; (ii) a larger dataset of the SOFI

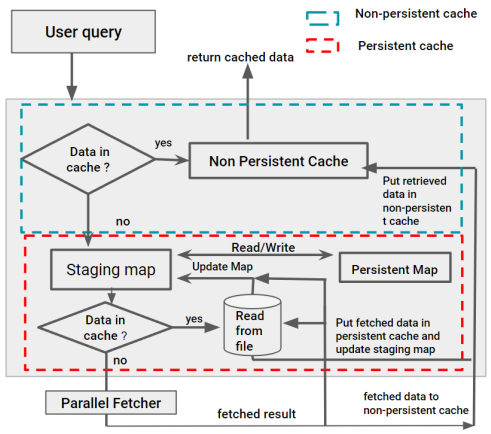

Fig. 2: The workflow of how a user's query is processed through our two-layer cache system. If the data is already available in the non-persistent cache this is returned to the viewer for rendering. Otherwise, our system checks for availability in the persistent cache accessing the *persistent map*. If data is not found in either cache the parallel fetcher requests the data from the remote resource.

stadium containing 300,000,000 points provided by the COPC website. To read, process, and write the LAZ file into a hierarchical COPC file format, the Point Data Abstraction Library (PDAL) was employed with the LAS reader to read and the COPC writer filters to write the processed data. The datasets in COPC format were both hosted for evaluation on GitHub and accessed using two different network connections: a very fast 330Mbps and a modest 5Mbps. We perform tests at such different speeds to highlight the benefits of caching and streaming techniques introduced in this paper in different scenarios, from high speed connections to applications operating in a mobile context or in rural areas at much lower speed.

The evaluation is performed over a close camera orbital motion with seven different positions around the scene equidistant from the center of the dataset. We set the screen space error threshold to 50, the perspective camera is configured with a field of view of 90 degrees, the frustum's near plane is at 0.1 and far plane is at 20,000. We set the maximum number of worker threads that a batch can spawn to 19 out of the 20 logical cores available to ensure that the main thread is not included in the count of worker threads. We set the max number of nodes allowed to be stored in non-persistent cache to 150, and 250 for the persistent cache (each node has on average 69,000 points, $\sim$2.4MB). In order to access the performance of different caching options, we recorded the performance of each strategy individually. For persistent caching, we recorded performance twice to examine its benefit when the browser or active tab is closed and reopened.

Figure 3 presents the performance of different caching strategies (and network speed) during our 7-point orbital navigation, with the x-axis representing the forward position during orbital motion. Figure 4 displays the time required for each individual step (at speed 5Mbps).

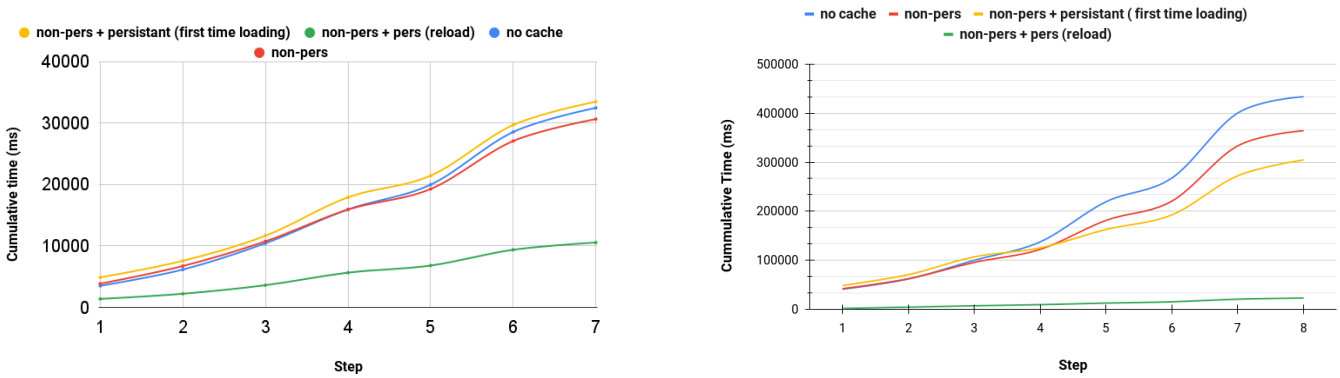

Fig. 3: Cumulative data retrieval time performance of multiple data caching options at different bandwidth speeds. The combination of persistent and non-persistent cache is better than any other when the tab/window is reloaded which is as expected (green, 2-3 times speed up). For very fast networks (left, 330Mbps) the use of caching provide little (with persistent caching) to no performance improvement. Great benefits are instead visible for slower networks (right, 5Mbps).

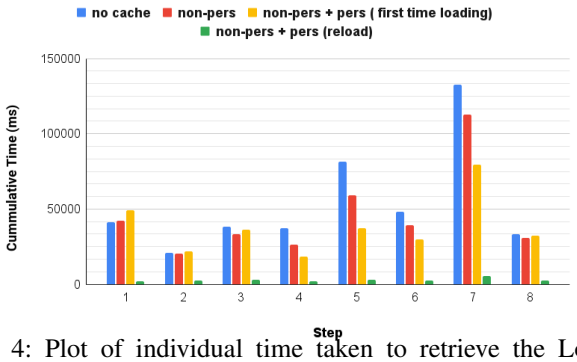

Fig. 4: Plot of individual time taken to retrieve the Logan dataset at a certain step while using different data caching options (at speed 5Mbps).

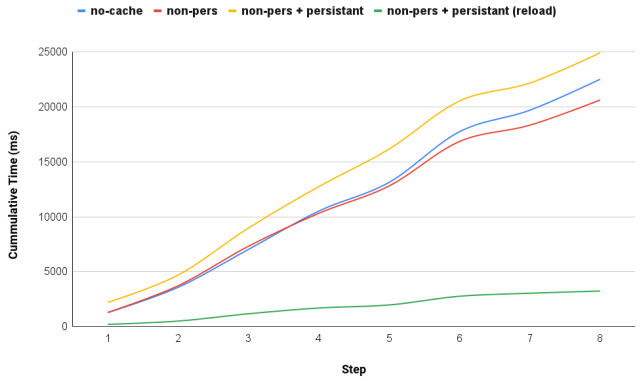

Fig. 5: Cumulative data retrieval time performance of multiple data caching strategies using very large dataset SOFI stadium (3M points) on a 330Mbps network. Performance results confirm trends seen for the smaller dataset.

These results show that when using very fast network (a) the overhead of accessing both layers of cache (persistent and non-persistent) is the slowest strategy. After the first few steps the LRU non-persistent cache configuration improves performance as data can be loaded from the cache rather than the network. In the case of a slower network (b) instead, the network latency is much higher than the caching search and retrieval overhead favoring strategies that use cache. In particular at later steps the combination of both cache provides the best performance (for first time loading). In Figure 3, we report a breakdown of the various timing at different steps.

When evaluating the system's performance with both caching layers after re-opening a closed tab/window, the performance benefits of persistent caching are quite significant (see Figure 4 and 3) providing a $2-3\times$ faster data retrieval. We expect this performance gain to grow over time as less data must be retrieved from the server. Furthermore, experiments using the larger dataset (i.e., 3M points SOFI stadium) show similar trends, for we report only results using a 330Mbps network speed in Figure 5.

Finally we evaluated our system using pre-fetching, i.e., loading in parallel also a coarse level data for areas that are not currently visible. From both cumulative (Figure 7) and individual (Figure 6) timing plots it is evident that the initial loading and reloading of the system, the starting time remains constant, as prefetching begins only after the first step. It is worth noting that the difference between the system's performance with and without prefetching is very distinguishable in Figure 7 where, as the camera moves, the gap between prefetching and not increases for first time loads and reloads.

Prefetching in orbital motion involves each camera step moving forward in an anti-clockwise direction. This can result in cache pollution, i.e., due to cache memory limitations pre-fetched data that is not being rendered is replacing useful data in the LRU cache. This can be observed in step 4 of Figure 6, both during initial loading and reloading. However, we find that the impact of this cache pollution may not be significant as the pre-fetched data contain few points over a large spatial domain, and thus has a high probability of being used.

V. Conclusions

This work introduces the first open source viewer for COPC data which leverages both WebGPU for parallel data retrieval and rendering, and a multi layer cache system, consisting of both persistent and non-persistent cache to enhance caching also when a browser tab/window is closed. Our findings

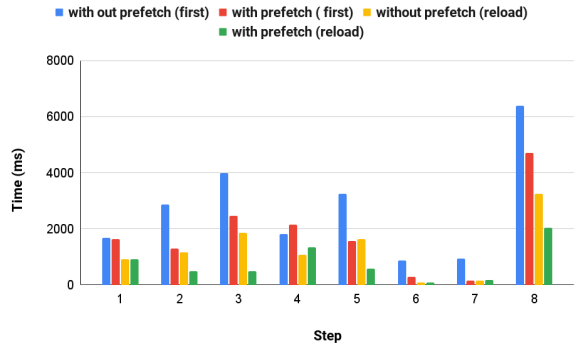

Fig. 6: Comparison of the time required to retrieve data in a system with a persistent and non-persistent cache at a specific step with or without prefetching

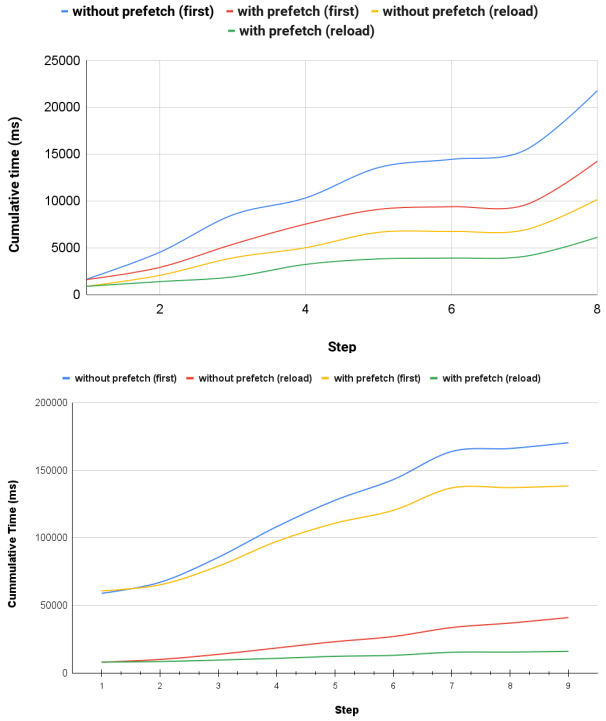

Fig. 7: Cumulative Retrieval Time with Prefetching and Without Prefetching on Adjoined Cache (persistent and non-persistent cache) for First-Time Loading and Reloading Conditions using different network speeds (top 330Mbps, bottom 5Mbps). Overall the pre-fetching improves performance.

indicate that utilizing this integrated cache system yields significant improvements in the performance of web-based point cloud data visualization. By combining both types of cache, frequently accessed data can be swiftly retrieved from the least recently used (LRU) cache, while data that has been removed from the non-persistent cache due to memory constraints can be accessed from the persistent cache. Furthermore, we incorporated a simple pre-fetching strategy into our, which preemptively fetch coarser resolution data before they are requested for rendering, further improving performance. Our experimental study demonstrates the benefits of using both types of cache. In particular, non persistent cache improves the performance of the system after just a few steps of data exploration, while the persistent cache allows retrieving data for rendering after a tab/window is closed from 2 to 3 times faster than fetching and traditional non-persistent caching. This functionality also allows for offline (or limited access) data visualization and is highly applicable to virtually any web-based visualization tool that needs to fetch large data from remote data sources.

VI. Acknoledgments

This work was funded in part by NSF SHF award 2221812.

References

[1] R. Cabello. JavaScript 3D Library. https://threejs.org/, 2010. [Online; accessed 27-April-2023].
[2] H. co. *Point Data Abstraction Library*. Honu Inc.
[3] W. W. W. Consortium. WebGPU. https://www.w3.org/TR/webgpu/, 2023. [Online; accessed 27-April-2023].
[4] G. O. Ganfure, C.-F. Wu, Y.-H. Chang, and W.-K. Shih. Deepprefetcher: A deep learning framework for data prefetching in flash storage devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3311–3322, 2020.
[5] X. GmbH. Online lidar point cloud viewer which works directly in your browser without transferring any data to the internet. http://lidarview.com/.
[6] E. Gobbetti and F. Marton. Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics*, 28(6):815–826, 2004.
[7] K. Group. Low-Level 3D Graphics API Based on OpenGL ES. https://www.khronos.org/webgl/, 2017. [Online; accessed 27-April-2023].
[8] H. Inc. Copc and ept via 3d tiles viewer. https://viewer.copc.io/.
[9] H. Inc. Hobu, inc. develops innovative open source solutions for point cloud data management. hobu.co.
[10] H. Inc. Cloud optimized point cloud specification 1.0. https://copc.io/copc-specification-1.0.pdf, 2021. [Online; accessed 24-April-2023].
[11] R. Koller and R. Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010.
[12] D. Kotz and C. S. Ellis. Practical prefetching techniques for parallel file systems. 1991.
[13] T. C. Mowry, A. K. Demke, O. Krieger, et al. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *OSDI*, vol. 96, pp. 3–17, 1996.
[14] Mozilla. *https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API*. Mozilla.
[15] OpenTopography. *opentopography: High-Resolution Topography Data and Tools*. opentopography.
[16] S. B. Rafael Gaitan. Free 3d lidar display and processing tool. PointCloudViz.com.
[17] S. Rusinkiewicz and M. Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 343–352, 2000.
[18] M. Schütz et al. Potree: Rendering large point clouds in web browsers. *Technische Universität Wien, Wiedeń*, 2016.
[19] M. Schütz, B. Kerbl, P. Klaus, and M. Wimmer. Gpu-accelerated lod generation for point clouds. *arXiv preprint arXiv:2302.14801*, 2023.
[20] M. Schütz, B. Kerbl, and M. Wimmer. Software rasterization of 2 billion points in real time. *arXiv preprint arXiv:2204.01287*, 2022.
[21] M. Schütz, G. Mandlburger, J. Otepka, and M. Wimmer. Progressive real-time rendering of one billion points without hierarchical acceleration structures. In *Computer Graphics Forum*, vol. 39, pp. 51–64. Wiley Online Library, 2020.
[22] E. P. Tile. *https://entwine.io/en/latest/entwine-point-tile.html*. Hobu Inc, 2021. Hi.
[23] U. Verma and H. Butler. Drag-n-drop In-browser LAS/LAZ point cloud viewer. https://plas.io/, Feb 2014.
[24] Y. Wu, H. Vo, J. Gong, and Z. Zhu. Unitypic: Unity point-cloud interactive core. *Parallel graphics and visualisation*, 2021.