Quantifying Performance Gains of DirectStorage for the Visualisation of Time-Dependent Particle Data Sets

Christoph Müller[1*] and Thomas Ertl[1]

[1]Visualisierungsinstitut der Universität Stuttgart, Universität Stuttgart, Allmandring 19, Stuttgart, 70569, Baden-Württemberg, Germany.

*Corresponding author(s). E-mail(s): christoph.mueller@visus.uni-stuttgart.de;
Contributing authors: thomas.ertl@visus.uni-stuttgart.de;

Abstract

Visualisation is an essential tool for analysing unstructured particle data as they are produced, for instance, in molecular dynamics or astrodynamics simulations. Such simulations often comprise multiple time steps, and scientists produce data sets with more and more particles in each step. To handle the steadily increasing size of these simulations, many solutions based on organising the data efficiently or reducing their amount using compression have been proposed over the years. Recently, a new storage API called *DirectStorage* has been introduced on Xbox consoles and Windows. DirectStorage promises a dramatic reduction of loading times in games by making the transfer from NVMe drives to graphics memory more efficient. That begs the question of whether DirectStorage is also beneficial for visualising time-dependent particle data that have to be streamed from disk to the GPU and whether a potential improvement in throughput enables interactive streaming of frames that traditional APIs cannot handle. For that, we implemented a benchmarking application that supports different streaming methods. We report on the results of an extensive series of tests with varying parameters that influence the streaming performance, which show that the performance of DirectStorage is highly dependent on the choice of various parameters.

Keywords: Particle visualisation, Particle streaming, DirectStorage, Direct3D 12

1 Introduction

Unstructured particle data are used for a wide variety of scientific applications, such as molecular dynamics and cosmology simulations, but such data are also produced by laser scans or photogrammetry. Especially simulations can handle trillions of particles nowadays, thanks to the ever-increasing computational power of high-performance computing (HPC) clusters. Visualisation as an essential means of analysis for simulations has to deal with these amounts of data and has developed different strategies such as compression, hierarchical organisation and optimising streaming to the GPU, some of which are already integrated into how the simulation codes store their results. Nevertheless, disk I/O not keeping pace with the computing power of clusters and GPUs is a critical bottleneck for analysing time-dependent simulation trajectories. Even worse, writing results to disk is already problematic when running the simulations (Kumar et al., 2019; Usher et al., 2021; Hoang et al., 2021). Less extreme, but still a relevant issue, loading data from disk has become a bottleneck in games with massive amounts of assets that need to be prepared for the highly detailed levels of modern computer games. Microsoft addressed this by introducing a new I/O path on the Xbox Series X, which was later ported to personal computers as *DirectStorage* (Yeung, 2020). This new API was designed to exploit the high data transfer rate of NVMe SSD specifically and to introduce less overhead in the driver stack than traditional POSIX-style APIs. In particular, DirectStorage integrates with the copy queues of Direct3D 12 and enables programmers to issue I/O requests from disk directly into a GPU-resident buffer, bypassing any intermediate application-level buffer. Such a design seemingly fits the requirements of scientific applications that need to stream large amounts of particles from disk to GPU, albeit the data need not only to be read from disk interactively but also rendered sufficiently quickly. Therefore, the potential application area for such a technology would be data sets of single frames sufficiently small to be rendered quickly but which form a time series that is too large to be kept in RAM at once, let alone in video RAM (VRAM). To test the idea that using DirectStorage is beneficial in this application case, we developed a benchmark that streams particles using DirectStorage into a Direct3D 12 renderer, which in turn visualises these particles as spherical glyphs. Using this application, we performed a series of measurements to identify combinations of parameters – data set sizes but also the sizes of various buffers used by DirectStorage – that do or do not work well. Our tests also include the *GDeflate* feature (Uralsky, 2022), which allows DirectStorage to read compressed files from disk and decompress them on the GPU.

2 Related work

Nowadays, there are two main directions when it comes to interactively rendering large amounts of spherical glyphs for visualising simulation results: One uses the rasterisation-based graphics pipeline to create proxy geometry for each particle and compute the pixels of the implicitly parametrised spheres in the pixel shader following the basic technique developed by Gumhold (2003). Thanks to its scalability, this technique has found its way into scientific visualisation systems such as MegaMol (Grottel et al., 2015), VisIt (Childs et al., 2012) or VMD (Humphrey et al., 1996).

The advantage of the idea is that it is effectively a brute-force approach that works on unstructured data without preprocessing, although there are methods to reduce the rendering load by culling invisible particles (Grottel et al., 2010). Under some circumstances, it can be beneficial to reduce overdraw by investing in an additional view-dependent sorting step that allows the GPU to discard invisible proxy geometry before generating fragments (Müller et al., 2019). Furthermore, hierarchical data structures can scale the approach to even larger data sets (Fraedrich et al., 2009). While the computations in the pixel shader are mostly the same for all the variants of Gumhold's approach, a multitude of techniques exist for generating the proxy geometry. These range from point primitives in OpenGL over instancing quads, using the geometry shader, using the tessellation shader to using the more recently introduced mesh shader. Gralka et al. (2023) provided a comprehensive overview of many of these techniques implemented in OpenGL and included a comparison of their performance on the latest hardware generations. A previous performance survey by Bruder et al. (2020) used Direct3D implementations but lacked the then non-existing mesh shader.

Gralka et al. (2023) also included the other technique that has become increasingly popular: ray tracing. Since the introduction of NVIDIA's Turing architecture (NVIDIA, 2017), GPUs have added dedicated support for ray tracing, previously mainly used for non-interactive high-quality renderings on the CPU. The CUDA-based Optix (Parker et al., 2010) is one way to use the ray tracing features of GPUs, but similar programming models have subsequently been added to Vulkan and Direct3D 12. Being an image-order approach, ray tracing typically needs some kind of acceleration structure, be it bounding volume hierarchies (Clark, 1976), kd-trees (Bentley, 1975) or octrees (Samet, 1989), to be efficient in processing large amounts of particles. While the former are mostly used for GPU-accelerated ray tracing, ray tracing on the CPU oftentimes uses the latter two. For instance, Wald et al. (2015) proposed a variant of kd-trees specialised for rendering large amounts of particles in the CPU ray tracing framework OSPRay. They argued that the large amount of relatively cheap main memory makes ray tracing on the CPU viable as long as the ray tracer can efficiently organise memory accesses and the data structures it uses have a low memory overhead like their proposed P-k-d-tree. Using their approach, they achieved interactive frame rates for millions of particles. The idea of P-k-d trees was later ported by Gralka et al. (2020) to the GPU. In their work, they compared it to different spatial partitioning strategies, the previously mentioned brute-force rasterisation approach on the GPU, and an OSPray implementation. They concluded that – if memory permits – bounding volume hierarchies are fastest, but if memory is scarce, different trade-offs have to be made when choosing the best fitting approach. While ray tracing approaches with appropriate acceleration structures outperform rasterising unstructured particles, our focus is on the latter because data that require a time-consuming preprocessing step before rendering cannot reasonably be streamed from disk to GPU.

Especially for simulation data, which typically comprise multiple frames to represent the temporal evolution of the simulation, GPU memory – and even main memory in some cases – does not suffice any more to hold all of the data. At this point, it becomes necessary to stream data from RAM to the GPU or even from disk to the GPU. Grottel et al. (2009) investigated the performance of many combinations of streaming particles to the GPU and rendering them using OpenGL. Later work (Grottel, 2012) also looked into streaming the data from disk. This study compared different I/O methods ranging from C++ streams over POSIX-style I/O functions to memory mapping and block sizes of 4 KB and 4 MB. While looking not specifically into unstructured particles but into streaming triangle geometry, Wiedemann and Kranzlmüller (2019) performed an experiment that investigated various options that influence data upload from main memory to VRAM. In their experiment, they tested millions of combinations and subsequently relied on statistical data analysis to draw their conclusions.

For I/O already being a bottleneck when creating particle data sets in a simulation, there is extensive work on optimising parallel I/O during the simulation to reduce the downtime in computation due to saving data (Usher et al., 2021). Kumar et al. (2019) designed a parallel I/O scheme specifically tailored towards particle data that not only addresses the simulation's needs but simultaneously produces output suitable for interactive visualisation. However, we focus only on local read operations for streaming in this work, while additional parallel layers might be built on top of that. Likewise, we do not consider any particle compression techniques like the one described by Hoang et al. (2021), quantisation techniques (e. g. Grottel et al. (2009); Fraedrich et al. (2009)) or special on-disk layouts of data (Fraedrich et al., 2009) except for the data-agnostic built-in compression techniques provided by DirectStorage.

DirectStorage is a technology first introduced on Xbox Series X consoles as part of the so-called *Velocity architecture* and later ported to Windows PCs (Yeung, 2020; Microsoft, 2022). Having this heritage, speeding up the process of loading the ever-growing amount of assets in modern games is the obvious purpose. At GDC 2022, Serandour and Ono (2022) presented data from an actual game, which showed an improvement of around 15 % compared to loading a level using Win32 API I/O functions from an NVMe drive. A year later, AMD provided further guidance on effectively using DirectStorage (Ziman, 2023). In their synthetic tests, they achieved significantly larger speedups than Serandour and Ono, not least as they also included the on-GPU GDeflate decompression feature of DirectStorage (Uralsky, 2022). Furthermore, they included the results of a series of raw I/O tests indicating that current generations of NVMe drives perform best with a queue depth of 16 or higher and with block sizes of at least 32 KB. To accelerate I/O in machine learning and HPC applications, NVIDIA has developed a technology named *GPUDirect Storage* (GDS) (Thompson and Newburn, 2019), which can perform DMA from NVMe drives directly to the GPU while bypassing main memory. In that way, DirectStorage and GPUDirect Storage serve similar purposes of speeding up transfers from disk to VRAM, but while the former is only available on Windows, the latter is only available on Ubuntu and Red Hat Enterprise Linux. Inupakutika et al. (2022) provided performance data for GDS using synthetic I/O benchmarks and actual deep learning workloads. Their study not only includes numbers for a single NVMe drive, but they also investigated the use of a distributed file system. The 14 % increase of single-drive throughput they found is in line with the numbers of Serandour and Ono (2022) for DirectStorage. Besides GDS, CUDA provides means for asynchronously overlapping I/O and computation via CUDA streams, allowing for a programming approach similar to DirectStorage combined with Direct3D 12 fences. However, in order to maximise the benefit of overlapping transfers and computation, the work must be split such that compute and copy engines are kept equally busy. Gómez-Luna et al. (2012) therefore performed an empirical study to derive guidelines for finding optimal parameters. One of the goals of our experiment goes in the same direction, i. e. we also want to get an idea of how the different parameters of DirectStorage can be used to tweak the upload performance.

3 Implementation

As raycasting spherical glyphs is one of the most widely used ways of visualising the unmodified results of particle-based simulations, we based our streaming implementation using DirectStorage on such an approach. For that, we ported one of the Direct3D 11 methods used by Bruder et al. (2020) to Direct3D 12[1]. Specifically, we used the raycasting shader on instanced quad glyphs, which was the most efficient in their work. This port was more or less straightforward: Shaders could be used without modifications across the APIs, but we opted to embed the root signatures in HLSL rather than build them in C++. The rationale here is that the framework generates the shaders for different data formats at build time, which provides a convenient way to create matching root signatures as well, without the need to handle any of that in the host code.

On the host side, we use double buffering, i. e. a pipeline depth of two, to achieve a behaviour comparable to the original Direct3D 11 implementation. For each of these two frames, we record a command list which instances four vertices for each particle in the data set. Based on the vertex ID, the vertex shader subsequently creates a unit quad without a vertex buffer being bound and transforms this quad to its position in world space based on the per-particle data read from a structured buffer view at the position of the instance ID. The pixel shader then produces the image of the sphere on the fragments generated by the quad following the method by Gumhold (2003). The only difference between the command lists is the render target, which is enabled at their respective start. In the most basic case, these lists remain unchanged during rendering and are executed alternatingly.

3.1 Streaming particles

The trivial port of using a command list per frame does not work anymore when it comes to streaming because it becomes necessary to change the resource bindings during a frame. Furthermore, the contents of the buffer holding the particle data change for every draw call. GPU memory in Direct3D 12, which holds these buffers, is organised in so-called *heaps*. Heaps are sections of VRAM that can have different properties, like being accessible by the GPU only or allowing for transfer between main memory and GPU memory. Often, data accessed by shaders are allocated on heaps only accessible by the GPU, particularly if the data are immutable. This is, for example, the case for the aforementioned implementation of sphere raycasting on Direct3D 12. However, when streaming particles, the host code would have to transfer the data first

[1] https://github.com/UniStuttgart-VISUS/trrojan

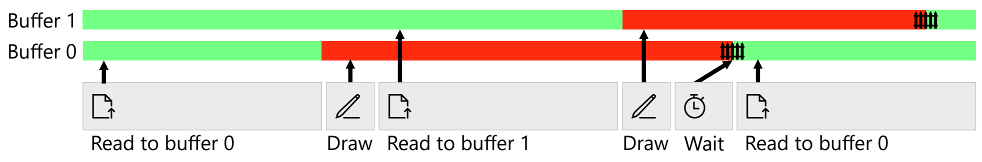


Fig. 1 Simplified illustration of streaming particle data to a persistently mapped buffer, in this case, for two segments. A buffer is marked red while the GPU is rendering on it and green while it is ready to accept new data from the CPU. After the draw call, a fence is injected into the command stream of the GPU, which the GPU will signal once it is done rendering the segment. When the CPU submits the second segment, the GPU uses both buffers simultaneously. Therefore, the next segment cannot be uploaded, and the CPU needs to wait for the GPU to finish work. This can be achieved by waiting for the fence to be signalled. In the actual implementation, the CPU does not wait for a specific buffer to become ready but for any of them.

to a CPU-accessible upload heap and subsequently issue a copy request that transfers the data from this upload heap to the resource being rendered. This not only incurs the cost for an additional copy, but it is also necessary to transition the resource from the copy target state to the shader resource state, which possibly stalls the pipeline. An alternative to this approach – the one we implemented because it better fits the way DirectStorage works – is to map the buffer on the upload heap persistently into the address space of the application and bind it as a shader resource simultaneously. This is an acceptable implementation as long as the application can guarantee that the region of the mapped buffer currently being rendered by any command list is not modified by the host code at the same time. For the CPU to know when it is safe to write new data, we split the resource holding the per-particle data into several segments. The actual implementation could place these segments either on a single heap or use multiple heaps, allowing for working around apparent heap size limits. Although we could not find official documentation on this, we found the maximum size of a heap being limited well below the available amount of physical VRAM, which is in line with reports by Gralka et al. (2023), who mentioned a vendor-dependent limit on contiguous allocations in OpenGL. Regardless of whether a segment was sub-allocated from a single heap or on an individual one, each command list rendered the particles of exactly one of them, and at the end of each list, we injected a fence that is being signalled once the GPU is done with that particular list (see Figure 1). Splitting the draw calls into multiple ones solely changes the number of primitives being processed and the resource descriptors that are bound. Therefore, we could use all of the rendering techniques implemented by Bruder et al. (2020), though we limited our measurements to the fastest one, which is instancing a quad as the proxy geometry of each particle.

At this point, there are multiple ways to fill the segments of the particle buffer, which all could take the place of `ReadFile` in algorithm 1: If all data fit into RAM, this can be as simple as a `memcpy`. Otherwise, the input file could be incrementally mapped into the address space of the process and copied from there, or it could be read using traditional POSIX-style I/O APIs or using a DirectStorage request from disk to memory. All of these methods effectively copy particle data to the location where a segment is mapped into the address space of the process. However, before they can do

Algorithm 1: Simplified process of streaming one frame using a traditional I/O API or copy from main memory.

```
Data: A file handle $H$ for the input data, the number of particles $P_T$, the
      number of particles per batch $P_B$, the number of batches $B$ that can
      be processed in parallel, an array $F$ of $B$ fences, an array $C$ of $B$
      command lists, a persistently mapped buffer $P_G$ for $B * P_B$ particles.
for $t \leftarrow 0$ to $\lceil P_T/P_B \rceil$   /* Iterate over all batches of particles. */
 do
    $b \leftarrow$ NextFreeBatch($F$)        /* Search signalled fence or wait. */
    $f \leftarrow F[b]$                            /* Get fence for batch. */
    $c \leftarrow C[b]$                  /* Get unused command list for batch. */
    QueueCommand ($c$, enable render target)       /* Set render target. */
    if $t = 0$ then QueueCommand ($c$, clear render target)
    ReadFile ($H$, &$P_G[b * P_B]$, $t * P_B$, $P_B$)     /* Read to mapped VRAM. */
    QueueCommand ($c$, set root signature) /* Configure shaders' input. */
    QueueCommand ($c$, set descriptors)         /* Set resource bindings. */
    QueueCommand ($c$, set primitive topology)    /* Set triangle strip. */
    QueueCommand ($c$, draw instanced)       /* Emit quad per particle. */
    if $t = \lceil P_T/P_B \rceil - 1$ then QueueCommand ($c$, disable render target)
    ExecuteCommandList ($c$)                /* Execute queued commands. */
    SignalWhenDone ($f, c$)          /* Signal the fence when rendered. */
    if $t = \lceil P_T/P_B \rceil - 1$ then Present (render target)
end
```

so, they must check whether the fence of the segment has already reached the value of the last draw call that was issued on the existing data at this location. If this is not the case, the application can have Direct3D signal a kernel event object once the GPU has reached the fence, which enables the host code to wait for the GPU to finish.

3.2 DirectStorage

While DirectStorage can be used as a replacement for traditional I/O APIs, its main benefit for the application case at hand is the tight integration with Direct3D 12. Most importantly, DirectStorage not only allows for issuing read requests from disk to main memory but it also accepts Direct3D resources as copy targets. In the best case, this operation can bypass any application buffer and copy directly from kernel memory and, if *Bypass I/O* is possible on an NVMe drive on Windows 11, additionally skip significant parts of the kernel I/O stack (Microsoft, 2023). Furthermore, this integration allows DirectStorage to upload compressed data and decompress it on the GPU. Decompression is a fully transparent feature for users of the API – it simply needs to be turned on, and the data on disk must be compressed in a compatible manner.

The rendering process is very similar to the previous streaming approach: The resource from which the GPU reads the particle data is segmented, and every unused segment is filled with new data and sent to the GPU to render it. The difference is

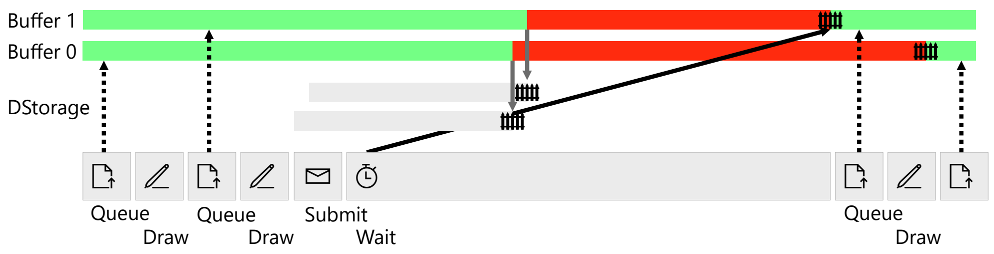


Fig. 2 Illustration of the streaming process when using DirectStorage: In contrast to the previous streaming implementation, the application no longer transfers particles directly to mapped GPU memory. Instead, it queues transfer requests and draw requests for the transferred data immediately after that. A fence that the GPU awaits before starting the draw makes sure that the rendering of a segment does not start before the transfer has been completed. As in the previous case, the application must handle the case when the GPU currently uses all buffers and prevent DirectStorage from overwriting this memory.

that the host code does not have to wait for the copy or I/O operation to complete any more because DirectStorage can signal fences, too (Figure 2). Hence, we have it signal a fence after every I/O request and have the command list wait for this fence before it starts rendering (algorithm 2). In other words, DirectStorage enables a "fire and forget" approach to queuing I/O and rendering without knowing the actual state of the I/O.

In DirectStorage, I/O requests are not fulfilled immediately, but they are queued, and all queued requests are processed as a batch once the queue is submitted. Microsoft recommends submitting larger batches simultaneously because submitting the queue is associated with some cost. Nevertheless, we have built our implementation to either immediately submit every request or multiple requests in batches, which allows for evaluating the overhead of frequent submissions. In the latter case, the submission is triggered by the host code detecting insufficient unused segments to submit more requests (see algorithm 2).

The GDeflate implementation is based on the method mentioned above of submitting batches of requests. As DirectStorage requires every request to be individually decompressable, the data must be preprocessed for this to work. Therefore, we perform a preprocessing step that splits the data set into chunks matching the size of a segment expected by the renderer and compress these individually. The compressed blocks are stored contiguously in a single file on disk while the application keeps an index of the offsets of each individual block as we need to provide the compressed and uncompressed extents when issuing a DirectStorage request. In a real-world application, this approach would require a custom file format that allows the software to read all of these indices before beginning to stream, thus removing all read-to-read dependencies during actual rendering.

Algorithm 2: Simplified process of streaming one frame using DirectStorage.

```
Data: A file handle $H$ for the input data, the number of particles $P_T$, the
      number of particles per batch $P_B$, the number of batches $B$ that can
      be processed in parallel, an array $F_G$ of $B$ fences, another array $F_I$ of
      $B$ fences, an array $C$ of $B$ command lists, a buffer $P_G$ for $B * P_B$
      particles, a DirectStorage I/O queue $Q$.
for $t \leftarrow 0$ to $\lceil P_T/P_B \rceil$   /* Iterate over all batches of particles. */
 do
    $b \leftarrow$ NextFreeBatch($F$)      /* Search signalled fence or wait. */
    $f_G \leftarrow F_G[b]$                /* Get fence for batch completion. */
    $f_I \leftarrow F_I[b]$                  /* Get fence for I/O completion. */
    $c \leftarrow C[b]$                /* Get unused command list for batch. */
    QueueCommand ($c$, enable render target)       /* Set render target. */
    if $t = 0$ then QueueCommand ($c$, clear render target)
    QueueRequest ($Q$, $H$, $t * P_B$, $P_G$, $b * P_B$, $P_B$)          /* Queue I/O. */
    QueueSignal ($Q$, $F_I$)     /* Make DStorage signal $F_I$ when done. */
    if $Q$ is full then SubmitQueue ($Q$)
    QueueCommand ($c$, wait $F_I$)     /* Make GPU to wait for DStorage. */
    QueueCommand ($c$, set root signature) /* Configure shaders' input. */
    QueueCommand ($c$, set descriptors)        /* Set resource bindings. */
    QueueCommand ($c$, set primitive topology)   /* Set triangle strip. */
    QueueCommand ($c$, draw instanced)      /* Emit quad per particle. */
    if $t = \lceil P_T/P_B \rceil - 1$ then QueueCommand ($c$, disable render target)
    ExecuteCommandList ($c$)             /* Execute queued commands. */
    SignalWhenDone ($f_G, c$)          /* Signal the fence when list was
     rendered. */
    if $t = \lceil P_T/P_B \rceil - 1$ then Present (render target)
end
```

4 Results

We performed a series of performance tests using synthetic data sets, which allowed us to control the properties of the data. As streaming data introduces overhead compared to simply rendering from VRAM, we also obtained reference measurements without DirectStorage, which allows for isolating the overhead introduced by streaming from RAM and the overhead introduced by streaming from disk using different methods.

4.1 Hardware

Our measurements were performed on two identically equipped PCs with an AMD Ryzen 9 5900X CPU and 64 GB RAM, one running Windows 10, the other running Windows 11. As GPUs, we used an NVIDIA RTX 4090 and an AMD Radeon RX 7900 XTX. The SSDs were Samsung 980 PRO NVMe drives and Samsung 850 PRO SATA drives. To get a baseline idea of the performance of the drives, we ran CrystalDiskMark 8.0.4 on each of them, one time using the default settings and another time using the

Table 1 Results of a read benchmark performed with CrystalDiskMark 8.0.4 on the SSDs used for our experiment using a 1 GB test file. The lower half of the table uses the settings recommended for NVMe drives.

Test configuration				Windows 10		Windows 11	
Access pattern	Block size	Queue depth	Threads	850 (SATA)	980 (NVMe)	850 (SATA)	980 (NVMe)
Sequential	1 MB	8	1	527 MB/s	6537 MB/s	531 MB/s	6486 MB/s
Sequential	1 MB	1	1	510 MB/s	4098 MB/s	528 MB/s	4156 MB/s
Random	4 KB	32	1	223 MB/s	581 MB/s	245 MB/s	633 MB/s
Random	4 KB	1	1	42 MB/s	84 MB/s	42 MB/s	87 MB/s
Sequential	1 MB	8	1	437 MB/s	6537 MB/s	533 MB/s	6205 MB/s
Sequential	128 KB	32	1	512 MB/s	4098 MB/s	536 MB/s	6567 MB/s
Random	4 KB	32	16	228 MB/s	3599 MB/s	244 MB/s	3888 MB/s
Random	4 KB	1	1	41 MB/s	84 MB/s	43 MB/s	87 MB/s

suggested NVMe settings. The results of these runs can be seen in Table 1. Differences between the standard and the suggested NVMe settings are inexistent with one notable exception: the NVMe drive massively benefits from issuing I/O requests from multiple threads in case of small random access patterns. Given these numbers, around 215 million particles per second are the upper limit for streaming, assuming the format we used in our test, which comprises three position components, a radius and four floating-point colour channels.

4.2 Procedure

The measurements were highly automated, and the system performed the following for each configuration of varying factors. First, a data set comprising one million, ten million or 100 million random particles was created such that the overall volume of the particles filled approximately one third of the cubic bounding box. Each particle was represented by eight 32-bit floating-point numbers: three for the position in space, one for the radius and four for the colour. The data sets were created using the C++ STL random number generator (but with a fixed seed value) and a uniform distribution over `floats`. These data were written contiguously to disk, repeating all particles four times to simulate multiple frames the application must seek to. This approach mimics that in a real-world scenario, the file format would have some kind of index allowing the application to directly seek to individual frames after reading this index once. In the case of the benchmark using GDeflate decompression on the GPU, the data set was pre-split into the segment length to be tested, and the segments were compressed individually, retaining the offsets to seek into the file. Each benchmark then rendered the frames round-robin while performing the tasks outlined in algorithm 3: First, four frames were rendered that were excluded from the results, ensuring no changes in the pipeline state from previous benchmarks influenced the measurements. Furthermore, this step computed the number of iterations for the wall-time clock measurements. Measurements were then obtained in three passes. First, we determined how long the GPU worked on each frame. As the queries used for that cannot span multiple command lists, we measured each draw call individually and aggregated the numbers

Algorithm 3: The steps for a single run of the benchmarks.

```
Data: The number of pre-warming renderings $R_p$, the desired time $T_w$ to
      measure the wall time for, the number of iterations $R_g$ for on-GPU
      timings.
/* Preparation                                                        */
Allocate resources and setup pipeline
/* Phase 1: Pre-warm and estimate frames to teach $T_w$.              */
$p \leftarrow R_p$
$r \leftarrow$ true
while $r$ do
    $s \leftarrow$ Now()
    for $i \leftarrow 1$ to $p$ do
        Render frame
    end
    Wait for GPU
    $e \leftarrow$ Now()
    $q \leftarrow \max(1, \lceil T_w \cdot p/(e-s) \rceil)$
    $r \leftarrow q > p$  /* Repeat if estimated frames for $T_w$ not reached. */
    $p \leftarrow q$      /* $p$ is now best guess for required iterations. */
end
/* Phase 2: Measure on-GPU timings using queries.                     */
$t_g \leftarrow$ GpuQueryResult$[R_g]$
for $i \leftarrow 1$ to $R_g$ do
    Render frame instrumented with timing query
    $t_g[i] \leftarrow$ Result of query
end
Sort($t_g$)
$m \leftarrow t_g[R_g/2]$                          /* Median of GPU times. */
if $R_g \mod 2 = 0$ then $m \leftarrow 0.5(m + t_g[R_g/2 - 1])$
Output $(t_g[0], m, t_g[R_g - 1])$
/* Phase 3: Measure wall time using OS timers.                        */
$s \leftarrow$ Now()
for $i \leftarrow 1$ to $p$ do
    Render frame
end
Wait for GPU
$e \leftarrow$ Now()
Output $((e-s), p, (e-s)/p)$
/* Phase 3: Collect pipeline statistics.                              */
Issue statistics query
Render frame
Output (Results of statistics query)
```

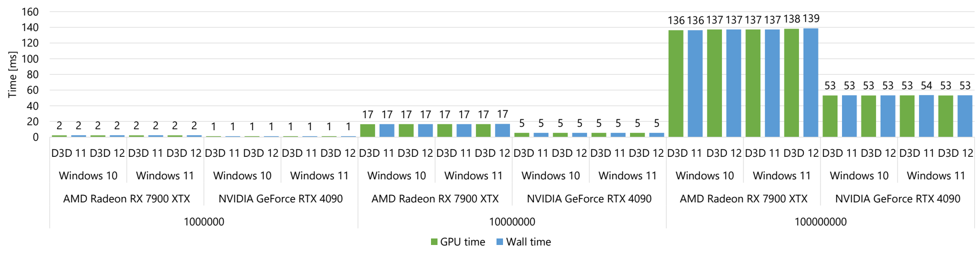


Fig. 3 Average frame times for rendering one million (31 MB per frame), ten million (305 MB per frame) and 100 million (3 GB per frame) particles from a buffer fully resident on the GPU. The operating system and the Direct3D API version barely influence the rendering performance. Furthermore, the overall wall-clock time per frame is clearly dominated by the time the GPU takes for rendering.

afterwards. In a second pass, the average frame time was measured on the CPU over the number of iterations determined in the first step. During this pass, the application also counted how often the host code had to wait for the GPU to finish in-flight draw calls, thus making at least one buffer available for transfer. The final step rendered only one frame to obtain pipeline statistics, i. e. the number of shader invocations, etc. Such statistics can be used to perform sanity checks on the data and judge the overdraw of the particles.

For all streaming implementations, there are mainly two factors that could be varied: the number of segments and the number of particles for each segment, which in turn determine the size of a single upload. DirectStorage has several additional parameters: the length of its internal queue and the size of its staging buffer. We did not modify the length of the queue as this primarily affects when the API performs a so-called auto submit, i. e. DirectStorage detects that the queue is running full and submits the queued requests without user intervention. We varied, however, the size of the staging buffer between 32 MB, 256 MB and 512 MB because of the findings of Ziman (2023). All of the test configurations were repeated for eight randomly chosen camera positions, ensuring the data set was visible. The random seed was fixed such that the same configurations were tested for each test case. For all test cases, the viewport was set to Full HD (1920 × 1080). Overall, we performed more than 30,000 individual measurements, which are available at https://doi.org/10.18419/darus-4017.

4.3 API and operating system

As a first step, we wanted to find out whether switching the renderer from Direct3D 11 to Direct3D 12 significantly impacted performance. There should be little to no API overhead for the base implementation issuing only a single draw call for all the particles. Performance should, therefore, be limited by what the hardware can handle, wherefore we hypothesised that there would be no significant difference between the implementations running on the same platform. To test this, we measured the rendering times of the Direct3D 11 implementation and the streaming-free port of the Direct3D 12 variant. Figure 3 shows the rendering times from the GPU timestamp queries and from the wall-clock timer on the CPU for different numbers of particles. The frame times for each condition are averages of all camera positions. Several of

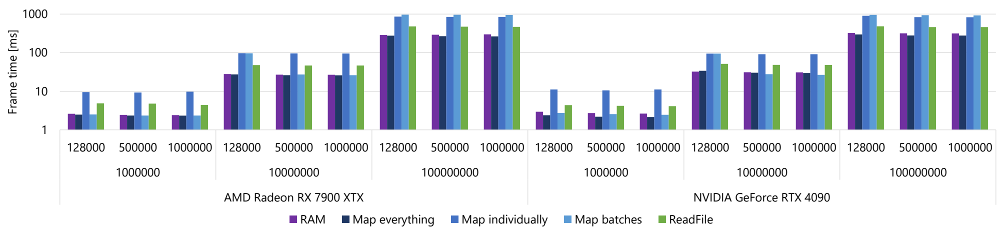


Fig. 4 Average wall-clock frame times for different "conventional" streaming methods when reading from a SATA drive. The factors on the x-axis are from top to bottom: the number of particles rendered for each batch (approximately 4 MB, 15 MB and 30 MB of raw data), the total number of particles per frame (approximately 31 MB, 305 MB and 3 GB of raw data per frame) and the GPU used. All numbers have been obtained on Windows 10 using 64 in-flight batches. The numbers for 256 parallel batches exhibit a similar structure overall, but the frame times are slightly higher except for `ReadFile`.

these were far away from the data set, leaving a lot of blank space in the frame, while others were in the data set such that all screen pixels were covered. The only apparent differences in Figure 3 are either caused by the choice of the GPU or the size of the data set. This effectively confirms the idea that the performance of this static rendering is limited by how many vertices and pixels the graphics card can push.

4.4 Conventional streaming

The first streaming method we tested was streaming from RAM. This is obviously only viable if the data set completely fits in the main memory, but the test gives us an idea of the overhead of streaming the data from RAM instead of holding them in VRAM. Next, we tested streaming using traditional APIs for file access. These included mapping the whole file into the process memory ("Map everything"), which is, again, not realistic for real-world large data. Therefore, we also implemented individually mapping each segment as needed ("Map individually") and mapping the number of segments that can be in flight simultaneously ("Map batches"). The latter two methods must always be possible, as otherwise, the system would not have sufficient main memory to run that configuration. Finally, we read each segment from disk as necessary using a blocking `ReadFile` call.

Figure 4 shows the wall-clock times for streaming one million, ten million and 100 million particles, respectively, using the methods above from the SATA drive. Streaming from RAM and mapping the whole file at once are, as expected, the fastest methods in all cases. They achieve around 30 fps even for the ten million particle data set on both cards – compared to almost 60 fps when rendering directly from VRAM on the AMD card and 200 fps for the NVIDIA GPU. The smallest data set can be rendered with around 100 fps even using the worst streaming method, which is mapping the whole file at once. Notably, even the blocking `ReadFile` achieves more than double the frame rate in this case and is also faster for the larger data sets. For the medium-sized data set, mapping batches is the best method actually transferring data from disk – except for the smallest batch size. For the largest data set, `ReadFile` becomes the best choice. However, frame rates around 2 fps and 3 fps even for the best method (streaming from RAM) are not interactive any more.

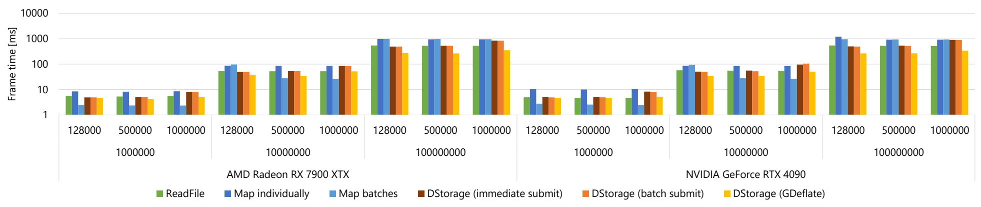


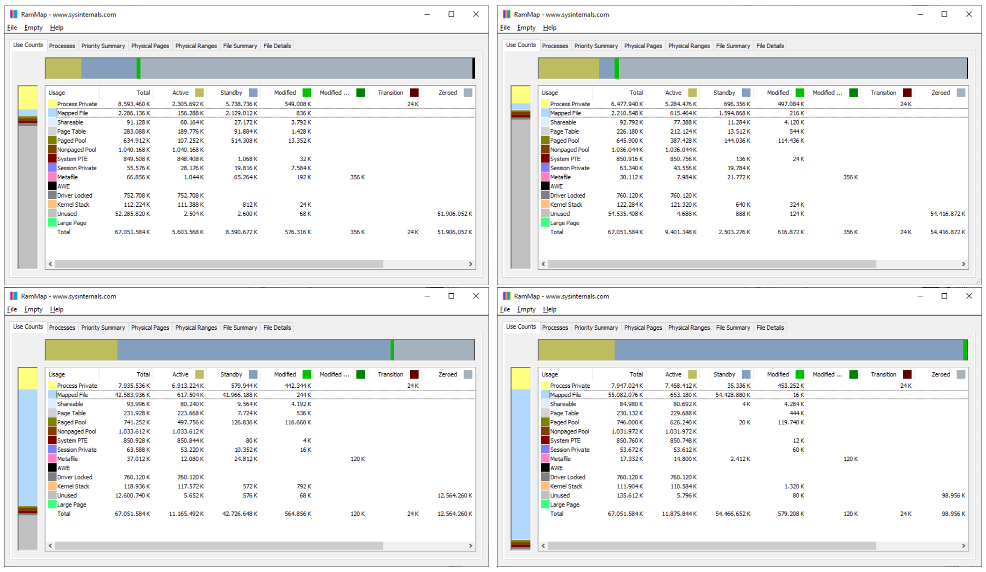
Fig. 5 Wall-clock frame times for different streaming methods when reading from an NVMe drive. The factors on the x-axis are from top to bottom: the number of particles read and rendered per batch, the total number of particles per frame and the GPU used. All numbers have been obtained on Windows 10 using a staging buffer of the Microsoft-recommended 32 MB and 64 in-flight batches.

The conventional methods for streaming from disk are also included in Figure 5, which shows the frame times when streaming from NVMe. Surprisingly, `ReadFile` becomes slower on the faster drive, with the NVMe drive requiring between 111 % and 123 % of the frame time from the SATA drive. We surmise that the combination of our access pattern and chunk sizes poorly saturates the NVMe I/O queues. Mapping the batches individually as needed, in contrast, only requires between 86 % and 89 % of the time for the smaller two data sets on the AMD card and between 90 % and 95 % on the NVIDIA. For the 100 million particles, it becomes slower, too – in the worst case, which is the NVIDIA card with the smallest batch size, it requires 132 % of the time measured for the SATA drive. Mapping the segments in batches behaves effectively the same for both kinds of drives within a ±2 % range. In particular, for the two smaller data sets, the frame times of this technique are suspiciously close to what we could achieve by streaming from RAM. On closer investigation, we found that this is because this method *is* effectively streaming from RAM as the operating system does not immediately remove mapped pages from physical memory as the application unmaps a segment. Instead, it keeps them in standby memory, making it basically "free" to reuse a frame already rendered before, provided all frames fit into RAM. Figure 6 illustrates this for the four frames we used in our benchmark and two additional runs with 128 and 256 extra copies, respectively. As the size of the data set exceeds the memory available for backing the mapping, the frame times increase, as expected in these tests.

The fact that the performance differences between the cards from Figure 3 have largely disappeared in Figure 4 and Figure 5 indicates that the performance here is limited by the transfer capabilities of the PCIe bus (for RAM and mapping the whole file) or the disk (for the other methods).

4.5 DirectStorage

As mentioned, we have implemented three variants using DirectStorage: the first one submits the queue immediately after enqueuing each request ("naive"). This goes against any recommendation, as submitting a queue incurs an overhead, which is relatively more significant for smaller submissions. However, Figure 5 shows that the performance hit by this implementation on NVMe drives is small and becomes only noticeable as the size of each segment increases. For the smaller batch sizes of 128,000 and 500,000 particles, both techniques perform slightly better than `ReadFile`; when



Usage	Total	Active	Standby	Modified	Modified ...	Transition	Zeroed
Process Private	8.593.460 K	2.305.692 K	5.738.736 K	549.008 K		24 K	
Mapped File	2.286.136 K	156.288 K	2.129.012 K	836 K			
Shareable	91.128 K	60.164 K	27.172 K	3.792 K			
Page Table	283.088 K	189.776 K	91.884 K	1.428 K			
Paged Pool	634.912 K	107.252 K	514.308 K	13.352 K			
Nonpaged Pool	1.040.168 K	1.040.168 K					
System PTE	849.508 K	848.408 K	1.068 K	32 K			
Session Private	55.576 K	28.176 K	19.816 K	7.584 K			
Metafile	66.856 K	1.044 K	65.264 K	192 K	356 K		
AWE							
Driver Locked	752.708 K	752.708 K					
Kernel Stack	112.224 K	111.388 K	812 K	24 K			
Unused	52.285.820 K	2.504 K	2.600 K	68 K			51.906.052 K
Large Page							
Total	67.051.584 K	5.603.568 K	8.590.672 K	576.316 K	356 K	24 K	51.906.052 K

Usage	Total	Active	Standby	Modified	Modified ...	Transition	Zeroed
Process Private	6.477.940 K	5.284.476 K	696.356 K	497.084 K		24 K	
Mapped File	2.210.548 K	615.464 K	1.594.868 K	216 K			
Shareable	92.792 K	77.388 K	11.284 K	4.120 K			
Page Table	226.180 K	212.124 K	13.512 K	544 K			
Paged Pool	645.900 K	387.428 K	144.036 K	114.436 K			
Nonpaged Pool	1.036.044 K	1.036.044 K					
System PTE	850.916 K	850.756 K	136 K	24 K			
Session Private	63.340 K	43.556 K	19.784 K				
Metafile	30.112 K	7.984 K	21.772 K		356 K		
AWE							
Driver Locked	760.120 K	760.120 K					
Kernel Stack	122.284 K	121.320 K	640 K	324 K			
Unused	54.535.408 K	4.688 K	888 K	124 K			54.416.872 K
Large Page							
Total	67.051.584 K	9.401.348 K	2.503.276 K	616.872 K	356 K	24 K	54.416.872 K

Usage	Total	Active	Standby	Modified	Modified ...	Transition	Zeroed
Process Private	7.935.536 K	6.913.224 K	579.944 K	442.344 K		24 K	
Mapped File	42.583.936 K	617.504 K	41.966.188 K	244 K			
Shareable	93.996 K	80.240 K	9.564 K	4.192 K			
Page Table	231.928 K	223.668 K	7.724 K	536 K			
Paged Pool	741.252 K	497.756 K	126.836 K	116.660 K			
Nonpaged Pool	1.033.612 K	1.033.612 K					
System PTE	850.928 K	850.844 K	80 K	4 K			
Session Private	63.588 K	53.220 K	10.352 K	16 K			
Metafile	37.012 K	12.080 K	24.812 K		120 K		
AWE							
Driver Locked	760.120 K	760.120 K					
Kernel Stack	118.936 K	117.572 K	572 K	792 K			
Unused	12.600.740 K	5.652 K	576 K	68 K			12.564.260 K
Large Page							
Total	67.051.584 K	11.165.492 K	42.726.648 K	564.856 K	120 K	24 K	12.564.260 K

Usage	Total	Active	Standby	Modified	Modified ...	Transition	Zeroed
Process Private	7.947.024 K	7.458.412 K	35.336 K	453.252 K		24 K	
Mapped File	55.082.076 K	653.180 K	54.428.880 K	16 K			
Shareable	84.980 K	80.692 K	4 K	4.284 K			
Page Table	230.132 K	229.688 K		444 K			
Paged Pool	746.000 K	626.240 K	20 K	119.740 K			
Nonpaged Pool	1.031.972 K	1.031.972 K					
System PTE	850.760 K	850.748 K		12 K			
Session Private	53.672 K	53.612 K		60 K			
Metafile	17.332 K	14.800 K	2.412 K		120 K		
AWE							
Driver Locked	760.120 K	760.120 K					
Kernel Stack	111.904 K	110.584 K		1.320 K			
Unused	135.612 K	5.796 K		80 K			98.956 K
Large Page							
Total	67.051.584 K	11.875.844 K	54.466.652 K	579.208 K	120 K	24 K	98.956 K

Fig. 6 Memory usage while running the benchmark performing batch-mapping of the input file as captured by *RamMap*. The first snapshot shows the idle state of the system after cleaning all working sets. The second one in the first row shows the state while running the ten million particle data set with four copies of the frame, as we did for our test runs. This file comprises approximately 1.2 GB, which are completely resident in the standby memory after the first iteration (light blue bar in the visualisation on the left). The third snapshot on the left of the bottom row shows the same benchmark when repeating the frame 128 times. Again, this file, which is approximately 40 GB, is fully backed by standby RAM (larger light blue bar on the left). For the last snapshot, the frame was repeated 256 times. This file is approximately 80 GB and, as such, cannot be fully backed by RAM on a system with 64 GB RAM. The memory occupied by mapped files is capped at around 60 GB in this constellation (the light blue bar on the left effectively occupies all the previously available memory).

rendering a million spheres at once, both perform worse. However, for the 100 million particle data set rendered using the RTX 4090, we found a combination of parameters (with a batch size of one million) where the presumably better implementation using batches performs worse than the naive one. This combination is, however, a bad one in general as the naive approach takes 177 % of the render time of `ReadFile`, and the batched approach even needs 195 %.

The second, more streamlined, variant ("batches") submits the queue whenever it cannot queue new storage requests, i. e. every time the configured number of parallel segments was requested from disk. As already mentioned, this technique performs very similarly to the naive one – on the NVIDIA card, sometimes a bit better with a difference of no more than a single-digit percent number.

Except for one case (streaming the smallest data set in one batch to the GeForce), the final implementation, which loads GDeflate-compressed data and has DirectStorage decompress it on the GPU, outperforms all other methods. On average, over all cases, including the one that performs worse, the GDeflate implementation requires only 57 % (48 % in the best case and 111 % in the worst case) of the time that `ReadFile` needs for the same benchmark configuration. This number is almost directly in line

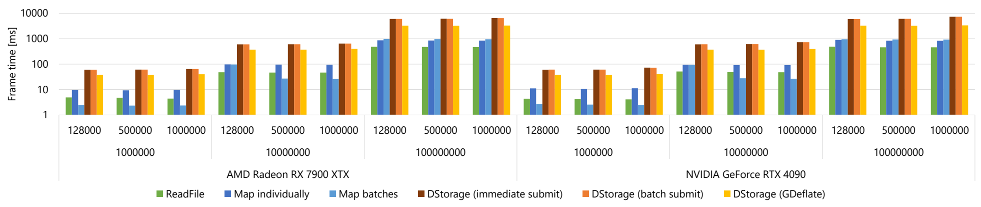


Fig. 7 Wall-clock frame times for different streaming methods when reading from a SATA drive. The factors on the x-axis are from top to bottom: the number of particles read and rendered per batch, the total number of particles per frame and the GPU used. As in Figure 5, all numbers have been obtained on Windows 10 using a staging buffer 32 MB and 64 in-flight batches.

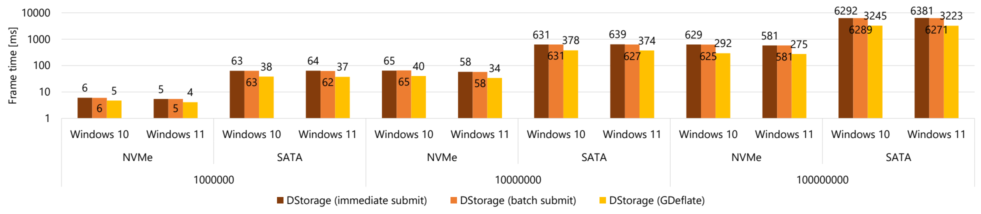


Fig. 8 Comparison of the averaged frame times for a staging buffer of 256 MB and 64 segments of 500,000 particles between Windows 10 and Windows 11. The achievable render times effectively do not differ.

with the compression ratio of GDeflate between 50 % and 60 %, depending on the size of the segments that are compressed individually.

Although 263 ms per frame (3.8 fps) are not interactive, DirectStorage achieves a massive improvement for 100 million particles (AMD rendering batches of 500,000, NVIDIA batches of 128,000) over any other streaming method. GDeflate streaming is effectively on par with streaming uncompressed particles from RAM (287 ms for batches of 128,000 on the AMD card, 315 ms on NVIDIA with batches of one million). The best "conventional" method performing actual I/O is `ReadFile`, which requires 420 ms to read from the NVMe drive in its best case, which is slightly faster than uncompressed DirectStorage but significantly slower than GDeflate.

Figure 7 shows the same results as Figure 5, but for the SATA drive. DirectStorage performs consistently worse than any other method in this case. In summary, GDeflate takes between six and ten times the time of `ReadFile` and the uncompressed variants between twelve and 18 times.

Finally, we were interested in whether DirectStorage would perform better on Windows 11 than on Windows 10. The comparison in Figure 8 shows that this is not the case, although we have verified that the NVMe drive we streamed the data from is eligible for Bypass I/O. As for the mixed results with the larger data sets, we cannot say whether this is due to an unfortunate choice of streaming parameters or whether there is no performance difference to expect.

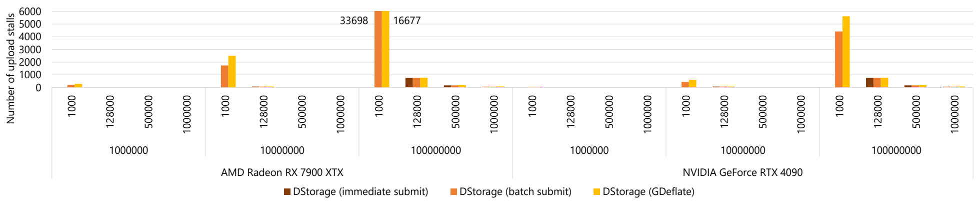


Fig. 9 The average number of cases when DirectStorage requests could not be made because the GPU was still working on all buffers. The factors on the x-axis are from top to bottom: the number of particles read and rendered per batch, the total number of particles per frame and the GPU used. The measurements were made on Windows 10 from the NVMe drive using 64 batches and a 32 MB staging buffer.

5 Discussion

While the port from Direct3D 11 to Direct3D 12 mostly behaved as we had expected, namely that there would be no difference in performance, the results for DirectStorage are mixed. In cases when DirectStorage performed better than `ReadFile` as the reference, the improvement was roughly in the same range as reported for games (Serandour and Ono, 2022). Only GDeflate was consistently faster than `ReadFile`, although more at the lower end of the improvements achieved by Ziman (2023). However, this only holds for the NVMe drive in our benchmark. On the SATA drive, we could not only achieve no speed up at all, but DirectStorage even performed significantly worse than basically any other technique. This is different from the results of Serandour and Ono (2022), and it is difficult to pinpoint whether this is a principal problem of the sequential access pattern interwoven with rendering or our specific choice of parameters – not at least because there are no details available for the implementation of *Forspoken*, which would allow finding similarities or differences between the implementations. It is, however, likely that the game would load the assets before starting to use them and probably also have more parallel in-flight read requests than we have.

When investigating the depth of the I/O queue of our benchmark in the *PIX* performance analyser, it was far less saturated than what Ziman (2023) showed. This leads to a fundamental problem illustrated in Figure 9, which shows how often our benchmark could not immediately submit a read request because the GPU still used all the buffers. This mostly happens when the batches are small, which hints at a massive overhead for submitting work to the GPU compared to what the GPU then actually does – filling the disk queues causes the GPU to be less efficient. Therefore, similar to what Gómez-Luna et al. (2012) found for CUDA streams, it is difficult to find a balance that equally saturates the rendering and copy queues of the GPU for perfect overlap. We believe that an even broader study of parameters is required for DirectStorage than the few thousand measurements we performed. However, a different approach than systematically testing combinations of parameters would be required for that, which already took days of fully automated benchmarks for the data at hand. The current approach puts certain a priori assumptions into the test, which makes it unlikely to find the black swan with a limited number of samples (Taleb, 2007). Given the large range in render times introduced by the size of the data sets and the comparably low noise added by the remaining parameters, our data were not even remotely normally distributed, much like the data of Wiedemann and Kranzlmüller (2019), who reported even aggregation of their data did not lead to passing the Shapiro-Wilk test. Therefore, randomly sampling the parameter space using per-parameter distributions should paint a more reliable picture of it and possibly hit unexpectedly good and bad combinations. However, it is equally possible that there are no significantly better parametrisations for our current approach, and the algorithm itself needs to be improved, for instance, by removing the one-to-one mapping of read requests to draw calls.

The unexpectedly good performance of memory mapping is a reminder that modern operating systems already perform a lot of optimisation under the hood. Therefore, DirectStorage is much like Direct3D 12 in that it *enables* programmers to tailor their code to leverage modern hardware, but at the same time also *requires* them to do so. In contrast, existing solutions provide decent performance with less complexity. Hence, it should be carefully considered if investing in low-level work is justifiable for a specific application case.

Finally, one might wonder if our findings might be transferable to other platforms. While the close integration of DirectStorage and Direct3D 12 via shared resources, fences and queues has no direct equivalent on Linux, it should be possible to build a similarly overlapping streaming implementation using the relatively new `io_uring` API – which should be beneficial on NVMe drives similar as DirectStorage is – and fences in OpenGL or, even better, Vulkan. Considering the previously stated complex relations between parameters, making predictions for such an implementation based on our measurements would be highly speculative – this must be evaluated empirically. Likewise, GDS transferring directly from NVMe to GPU memory is an even better solution than DirectStorage with its kernel-space copy, but only for CUDA resources. We cannot predict from our results whether the performance impact of sharing these resources with OpenGL or Vulkan would outweigh the benefits of the direct transfer – this must be measured, too.

6 Conclusion

We have implemented several variants of streaming particle data sets directly from disk to the GPU using DirectStorage. The variants include a very straightforward one that effectively just replaces a conventional I/O request with a DirectStorage one, one that batches multiple I/O requests, which better suits the deep queues of modern NVMe drives, and a final one that uses the built-in GDeflate feature that allows DirectStorage to uncompress data not until they are already on the GPU. To isolate which potential performance gains and losses can be attributed to DirectStorage, we also implemented different conventional streaming methods ranging from streaming from RAM to reading with POSIX-style APIs. The results were mixed and showed that it is imperative to choose the right parameters, which also depend on each other in some cases. In particular, when exploiting the compression feature, DirectStorage can be an improvement over other I/O methods. However, if the wrong parameters are chosen, there is not only no performance gain, but the effect can be detrimental, with DirectStorage performing an order of magnitude worse than `ReadFile`. Therefore, we plan on reworking our testing framework to sample the parameter space more randomly, as previously mentioned, hopefully leading to a better understanding of the interdependencies between parameters. We also want to investigate ways to decouple I/O and render queues to better saturate both at the same time.

Acknowledgements. This work was partially funded by Deutsche Forschungsgemeinschaft (DFG) as part of SFB/Transregio 161 (project ID 251654672).

Declarations

Not applicable

References

Bentley, J.L.: Multidimensional Binary Search Trees Used for Associative Searching. Commun. ACM **18**(9), 509–517 (1975) https://doi.org/10.1145/361002.361007

Bruder, V., Müller, C., Frey, S., Ertl, T.: On Evaluating Runtime Performance of Interactive Visualizations. IEEE Trans. Vis. Comput. Graph. **26**(9), 2848–2862 (2020) https://doi.org/10.1109/TVCG.2019.2898435

Childs, H., Brugger, E., Whitlock, B., Meredith, J., Ahern, S., Pugmire, D., Biagas, K., Miller, M., Harrison, C., Weber, G., Krishnan, H., Fogal, T., Sanderson, A., Garth, C., Bethel, E.W., Camp, D., Rubel, O., Durant, M., Favre, J., Navratil, P.: VisIt: An end-user tool for visualizing and analyzing very large data. High Perf. Vis.-Enabling Extreme-Scale Scient. Insight, 357–372 (2012)

Clark, J.H.: Hierarchical Geometric Models for Visible Surface Algorithms. Commun. ACM **19**(10), 547–554 (1976) https://doi.org/10.1145/360349.360354

Fraedrich, R., Schneider, J., Westermann, R.: Exploring the Millennium Run – Scalable Rendering of Large-Scale Cosmological Datasets. IEEE Trans. Vis. Comput. Graph. **15**(6), 1251–1258 (2009) https://doi.org/10.1109/TVCG.2009.142

Grottel, S., Krone, M., Müller, C., Reina, G., Ertl, T.: MegaMol—A Prototyping Framework for Particle-Based Visualization. IEEE Tran. Vis. Comput. Graph. **21**(02), 201–214 (2015) https://doi.org/10.1109/TVCG.2014.2350479

Gómez-Luna, J., González-Linares, J.M., Benavides, J.I., Guil, N.: Performance models for asynchronous data transfers on consumer Graphics Processing Units. J. Parallel Distrib. Comput **72**(9), 1117–1126 (2012) https://doi.org/10.1016/j.jpdc.2011.07.011

Grottel, S., Reina, G., Dachsbacher, C., Ertl, T.: Coherent Culling and Shading for Large Molecular Dynamics Visualization. Comput. Graph. Forum **29**(3), 953–962 (2010) https://doi.org/10.1111/j.1467-8659.2009.01698.x

Grottel, S., Reina, G., Ertl, T.: Optimized data transfer for time-dependent, GPU-based glyphs. In: Proc. Pac. Vis. Symp., pp. 65–72 (2009). https://doi.org/10.1109/PACIFICVIS.2009.4906839

Gralka, P., Reina, G., Ertl, T.: Efficient Sphere Rendering Revisited. In: Proc. Symp. Parallel Graph. Vis. (2023). https://doi.org/10.2312/pgv.20231083

Grottel, S.: Point-based Visualization of Molecular Dynamics Data Sets. PhD thesis, Universität Stuttgart (2012). https://doi.org/10.18419/opus-6385

Gumhold, S.: Splatting Illuminated Ellipsoids with Depth Correction. In: Proc. VMV, pp. 245–252 (2003)

Gralka, P., Wald, I., Geringer, S., Reina, G., Ertl, T.: Spatial Partitioning Strategies for Memory-Efficient Ray Tracing of Particles. In: Proc. LDAV, pp. 42–52 (2020). https://doi.org/10.1109/LDAV51489.2020.00012

Hoang, D., Bhatia, H., Lindstrom, P., Pascucci, V.: High-Quality and Low-Memory-Footprint Progressive Decoding of Large-Scale Particle Data. In: Proc. LDAV, pp. 32–42 (2021). https://doi.org/10.1109/LDAV53230.2021.00011

Humphrey, W., Dalke, A., Schulten, K.: VMD: Visual molecular dynamics. J. Mol. Graph. **14**(1), 33–38 (1996) https://doi.org/10.1016/0263-7855(96)00018-5

Inupakutika, D., Davis, B., Yang, Q., Kim, D., Akopian, D.: Quantifying Performance Gains of GPUDirect Storage. In: Proc. Int'l Conf. Netw., Arch. Storage, pp. 1–9 (2022). https://doi.org/10.1109/NAS55553.2022.9925516

Kumar, S., Petruzza, S., Usher, W., Pascucci, V.: Spatially-Aware Parallel I/O for Particle Data. In: Proc. ICPP (2019). https://doi.org/10.1145/3337821.3337875

Müller, C., Braun, M., Ertl, T.: Optimised Molecular Graphics on the HoloLens. In: Proc. IEEE VR, pp. 97–102 (2019). https://doi.org/10.1109/VR.2019.8798111

Microsoft: DirectStorage on Windows Samples. Online, last accessed 2023-12-01 (2022). https://github.com/microsoft/DirectStorage

Microsoft: BypassIO for filter drivers. Online, last accessed 2023-12-01 (2023). https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/bypassio

NVIDIA: NVIDIA Turing GPU Architecture. Technical report (2017). Available online, last accessed 2023-12-01. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

Parker, S.G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., *et al.*: Optix: a general purpose ray tracing engine. ACM Trans. Graph. **29**(4), 1–13 (2010)

Samet, H.: Implementing ray tracing with octrees and neighbor finding. Computers & Graphics **13**(4), 445–460 (1989) https://doi.org/10.1016/0097-8493(89)90006-X

Serandour, A., Ono, T.: Breaking Down the World of Athia: The technology of Forspoken. Talk at GDC 2022, available online, last accessed 2023-12-01 (2022). https://gpuopen.com/gdc-presentations/2022/GDC_Breaking_Down_The_World_Of_Athia.pdf

Taleb, N.N.: The Black Swan: The Impact of the Highly Improbable, 1st edn. Random House, New York (2007)

Thompson, A., Newburn, C.: GPUDirect Storage: A Direct Path Between Storage and GPU Memory. Online, last accessed 2023-12-01 (2019). https://developer.nvidia.com/blog/gpudirect-storage/

Usher, W., Huang, X., Petruzza, S., Kumar, S., Slattery, S.R., Reeve, S.T., Wang, F., Johnson, C.R., Pascucci, V.: Adaptive Spatially Aware I/O for Multiresolution Particle Data Layouts. In: Proc. IPDPS, pp. 547–556 (2021). https://doi.org/10.1109/IPDPS49936.2021.00063

Uralsky, Y.: Accelerating Load Times for DirectX Games and Apps with GDeflate for DirectStorage. Online, last accessed 2023-12-01 (2022). https://developer.nvidia.com/blog/accelerating-load-times-for-directx-games-and-apps-with-gdeflate-for-directstorage/

Wiedemann, M., Kranzlmüller, D.: Statistical Analysis of Parallel Data Uploading using OpenGL. In: Proc. Symp. Parallel Graph. Vis. (2019). https://doi.org/10.2312/pgv.20191114

Wald, I., Knoll, A., Johnson, G.P., Usher, W., Pascucci, V., Papka, M.E.: CPU ray tracing large particle data with balanced P-k-d trees. In: Proc. VIS, pp. 57–64 (2015). https://doi.org/10.1109/SciVis.2015.7429492

Yeung, A.: DirectStorage is coming to PC. Online, last accessed 2023-12-01 (2020). https://devblogs.microsoft.com/directx/directstorage-is-coming-to-pc/

Ziman, D.: DirectStorage – Optimizing load-time and streaming. Talk at GDC 2023, available online, last accessed 2023-12-01 (2023). https://gpuopen.com/gdc-presentations/2023/GDC-2023-DirectStorage-optimizing-load-time-and-streaming.pdf