To Infinity and Beyond 🚀: a GPU-driven memory sharing pipeline to generate and process infinite synthetic data

DANIELE DELLA PIETRA, University of Trento, Italy
GINO LANZO HAHN, University of Trento, Italy
NICOLA GARAU, University of Trento, Italy and CNIT, Italy

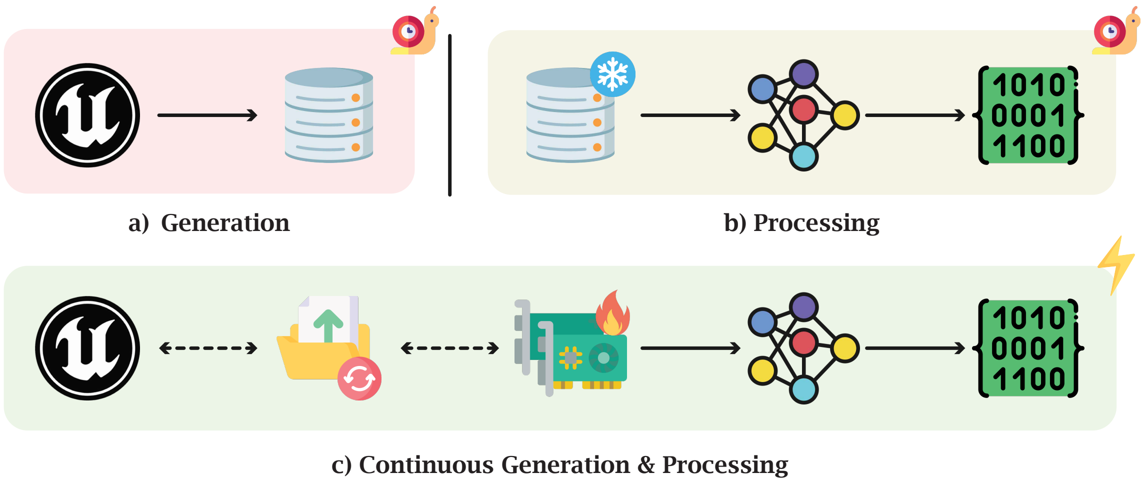

Fig. 1. Overview of a typical synthetic data generation pipeline (a) and data processing via an algorithm or neural network (b), versus our fast unified synthetic data generation and processing pipeline (c).

What if data generation, manipulation, and training could all happen entirely on the GPU, without ever touching the RAM or the CPU? In this work, we present a novel pipeline based on Unreal Engine 5, which allows us to generate, render, and process graphics data entirely on the GPU. By keeping the data stored in GPU memory throughout all the steps, we bypass the traditional bottlenecks related to CPU-GPU transfers, significantly accelerating data manipulation and enabling fast training of deep learning algorithms. Traditional storage systems impose latency and capacity limitations, which become increasingly problematic as data volume increases. Our method demonstrates substantial performance improvements on multiple benchmarks, offering a new paradigm for integrating game engines with data-driven applications.

ACM Reference Format:
Daniele Della Pietra, Gino Lanzo Hahn, and Nicola Garau. 2025. To Infinity and Beyond 🚀: a GPU-driven memory sharing pipeline to generate and process infinite synthetic data. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Posters (SIGGRAPH Posters '25), August 10-14, 2025*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3721250.3742983

Authors' addresses: Daniele Della Pietra, University of Trento, Italy, daniele.dellapietra@studenti.unitn.it; Gino Lanzo Hahn, University of Trento, Italy, gino.lanzohahn@studenti.unitn.it; Nicola Garau, University of Trento, Italy and CNIT, Italy, nicola.garau@unitn.it.

1 INTRODUCTION AND MOTIVATION

During the past few years, game engines have proven to be more than capable at generating and rendering high resolution, photorealistic scenes, with extensive customization options. For this reason, they are the ideal tools for generating synthetic datasets [1, 2] that can be used to speed up data-driven applications. However, the typical workflow—(1) build 3D scenes in a game engine, (2) render and write its data to disk, and finally (3) use that static data as input to an algorithm or neural network—suffers from repeated CPU–GPU round-trips and I/O stalls (Fig. 1a,b). The main bottleneck of this approach is usually the storage, which can contain a finite amount of static data, with a reduced access speed from the graphics hardware. Our contribution is twofold: (i) generating an infinite amount of data in real-time on the GPU, and (ii) consuming it without storing it anywhere, bypassing the delays due to storage access. A high-level description of this pipeline can be seen in Fig. 1.

2 METHOD

From the Unreal Engine website[1]: *In recent years, Python has become the de facto language for production pipelines and interoperability between 3D applications, particularly in the media and entertainment industry. This is partially due to the wide range of applications that support it. As the complexity of production pipelines continues to soar, and the number of applications involved continues to grow, having a common scripting language makes it easier to create and maintain large-scale asset management systems.*

The natural question that follows is: how can we build a fast and efficient communication channel between the engine and an arbitrary Python script, to share any object in UE5, without storing it anywhere?

To address this task, we designed two solutions, both of which enable this sharing process in a synchronized, fast way, without relying on the physical storage at any point: **RAM Sharing** and **GPU Sharing**.

2.1 RAM Sharing

On the engine side, we develop a simple plugin that allocates a shared memory portion that lives in the RAM via a dedicated C++ library (**`UE::Learning::SharedMemory`**). On the Python side, we develop a library enabling a communication channel with UE5, that can read data synchronously from it, taking care of concurrent accesses via a flag system.

2.2 GPU Sharing

This solution is a bit more complex, but it provides multiple advantages over the RAM solution. To eliminate all CPU–GPU transfers and enable truly zero-copy data exchange, we implement a GPU-resident sharing mechanism based on CUDA Inter-Process Communication (IPC). At initialization, we define the desired buffer dimensions (e.g., number of vertices and indices, or texture width and height) and store them on a memory portion identified by a GUID. Under the hood, this routine (1) allocates a contiguous device buffer via **`cudaMalloc`**, (2) zero-initializes it, (3) creates a shared-memory region in which it writes both a **`cudaIpcMemHandle_t`** and a **`cudaIpcEventHandle_t`**, and (4) returns a struct that encapsulates typed pointers into the device allocation for ease of indexing.

Once the shareable resource has been created on the engine side, we acquire the native D3D12 handle by calling **`ID3D12Device::CreateSharedHandle`**. This handle is then imported into CUDA and mapped to a device pointer. Concurrently, an IPC event is exported via **`cudaIpcGetEventHandle`** so that consumers can synchronize on rendering

[1]https://dev.epicgames.com/documentation/en-us/unreal-engine/scripting-the-unreal-editor-using-python

Metric	**RAM**	**GPU**	**RAM**	**GPU**	**RAM**	**GPU**	**RAM**	**GPU**	**RAM**	**GPU**	**RAM**	**GPU**	**RAM**	**GPU**	**RAM**	**GPU**
Model	Suzanne		Spot		Teapot		Bunny		Nefertiti		Lucy		Armadillo		Dragon	
Vertices	507		2930		3644		35947		49971		49987		49990		125066	
Faces	968		5856		6320		69451		99938		99970		99976		249882	
Data copy (ms)	0.133	0.293	0.141	0.311	0.154	0.295	0.390	0.301	1.013	0.308	1.046	0.307	1.007	0.306	2.218	0.301
Rendering (ms)	0.175	0.197	0.179	0.203	0.182	0.195	0.288	0.197	0.600	0.202	0.595	0.204	0.570	0.201	1.231	0.198
Frame time (ms)	0.330	0.540	0.343	0.563	0.358	0.537	0.699	0.547	1.635	0.557	1.664	0.558	1.599	0.555	3.473	0.548
Avg. FPS	3028	1851	2915	1778	2794	1862	1430	1829	612	1795	601	1791	625	1803	288	1826

Table 1. Results of our method when transfering 3D meshes from UE5 to Python, and rendering them with PyOpenGL. We show results averaged over 1000 runs, using a shared portion of either the RAM, or the GPU VRAM. Despite obtaining the best results when fully utilizing the GPU, we show how strategically allocating a shared portion of the RAM can also lead to good performances when compared to loading data from local storage. However, our approach on the GPU shows remarkably constant performances across all the models, showcasing impressive scalability potential.

completion. During each game-thread tick, we lazily open the IPC memory handle and enqueue a render-thread command to perform a device-to-device copy. By recording the CUDA event, downstream processes (e.g., a Python consumer using `cudaIpcOpenMemHandle` and `cudaIpcOpenEventHandle`) can wait for completion and directly access the GPU buffer without any host-side staging. This design generalizes to meshes, textures, or arbitrary tensor data, underlying the capabilities of our infinite, on-the-fly data sharing and processing pipeline.

3 EXPERIMENTS AND RESULTS

We run all of our experiments on a desktop computer with a single NVIDIA RTX 4090, an AMD Ryzen 7950x and 64GB of RAM. We validate our pipeline on three representative tasks: mesh transfer, texture rendering, and end-to-end training throughput. In each experiment, we compare the variants (columns in Tables 1, 2, 3): **DataLoader**: Load a dataset from storage and create a Pytorch GPU tensor from it; **RAM:** Share a portion of system memory between UE5 and a Python process; **GPU:** Keep all buffers resident in GPU VRAM and perform every stage in-place.

3.1 Mesh transfer

We first measure the cost of transferring 3D meshes from Unreal Engine 5 into Python and rendering them via PyOpenGL. Table 1 reports data-copy time (UE5 to Python, with no additional processing), rendering time (in Python, using GL), overall frame time, and average FPS over 1000 runs on eight common meshes taken from the Stanford 3D Scanning Repository [3] ranging from the low-poly *Suzanne* (507 vertices, 968 faces) to the high-poly *Dragon* (125 066 vertices, 249 882 faces).

Our results in Table 1 demonstrate that an end-to-end GPU approach offers dramatic speedups (up to 12× on the largest mesh) and constant performance as scene complexity grows.

3.2 Render target

In this scenario, we render RGBA textures at increasing resolutions (128 px–2048 px) in UE5 and share them with a Python process over either RAM or GPU. The shared images are then visualized using OpenCV, measuring the running time over 1000 runs. The results can be seen in Table 2.

Our GPU-resident pipeline removes the resolution-dependent bottleneck in both data transfer and image-processing stages. In particular, the GPU copy remains near 0.4–0.5 ms even at 2048 px, while the RAM variant shows a far bigger variance (13 ms for a 2048 px image).

Metric	RAM	GPU	RAM	GPU	RAM	GPU	RAM	GPU	RAM	GPU
Resolution	128	128	256	256	512	512	1024	1024	2048	2048
Data copy (ms)	0.035	0.362	0.259	0.410	0.770	0.429	2.809	0.491	13.144	0.427
Rendering (ms)	10.840	10.883	10.703	10.835	11.047	10.817	15.194	10.889	43.819	23.588
Total time (ms)	10.882	11.249	10.969	11.249	11.826	11.250	18.012	11.783	56.976	24.019
Avg. FPS	92	89	91	89	85	89	56	85	18	42

Table 2. Results of our method when transfering RGBA rendered images at various resolutions from UE5 to Python, averaged on 1000 runs. After the data copy, the images are rendered once again using OpenCV. In order to do that, the images need to be copied back to CPU memory, thus the increase in the **Rendering** row. Once again, our method running on the GPU shows impressive constant performances in the **Data copy** row, compared to the shared RAM memory method.

Metric	DataLoader	GPU	DataLoader	GPU	DataLoader	GPU	DataLoader	GPU	DataLoader	GPU
Resolution	128	128	256	256	512	512	1024	1024	2048	2048
Frame time	0.4 ms	3 ms	1 ms	3 ms	5 ms	3 ms	14 ms	3 ms	50 ms	3 ms
Resnet-50	79.16	57.61	76.63	63.34	61.71	61.60	24.53	31.06	5.80	7.89

Table 3. Results of our method when training on a synthetic, real-time-generated dataset of renderings at various resolutions, versus training on the same subset of pre-rendered images in Pytorch. In this table, **Frame time** is the time spent grabbing a single batch (including UE5's rendering time for our GPU scenario) and storing it in a Pytorch tensor, while **ResNet-50** describes the overall iterations per second, including the network processing time.

3.3 Training on a dataset

An interesting application of our method is the one of creating a generator of high quality infinite data on-the-fly for training deep neural networks in real-time. The motivation behind this approach is that we can bypass the reliance on storage and provide a constant stream of data during training, effectively acting like an infinite batch generator. The main challenge involved in this scenario is to make it real-time and to provide a similar throughput compared to the classic data loaders. We compare our GPU pipeline that loads images into PyTorch tensors with a standard Pytorch DataLoader and train a ResNet50 network on the fly. An interesting result that can be seen in Table 3 is that the frame time for our GPU pipeline remains almost constant at different resolutions, differently from the standard DataLoader.

4 ADDITIONAL MATERIAL

Please look at the Supplementary Material for further results and plots. The code will be available upon publication.

5 ACKNOWLEDGEMENTS

Funded by the European Union - Next Generation EU, Mission 4 Component 2 - CUP E63C22000970007.

REFERENCES

[1] Thomas Pollok, Lorenz Junglas, Boitumelo Ruf, and Arne Schumann. Unrealgt: Using unreal engine to generate ground truth datasets. In *Advances in Visual Computing : 14th International Symposium on Visual Computing, ISVC 2019, Lake Tahoe, NV, USA, October 7–9, 2019, Proceedings, Part I. Ed.: George Bebis*, page 670–682. Springer, 2019.

[2] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[3] Stanford University Computer Graphics Laboratory. The stanford 3d scanning repository. https://graphics.stanford.edu/data/3Dscanrep/, December 2010. Accessed: 2025-04-24.