To Infinity and Beyond

A GPU-driven memory sharing pipeline to generate and process infinite synthetic data

DANIELE DELLA PIETRA[1], GINO LANZO HAHN[1], NICOLA GARAU[1,2]

[1]UNIVERSITY OF TRENTO, ITALY, [2]CNIT, ITALY

SIGGRAPH 2025
Vancouver+ 10-14 August

Abstract

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.

Motivation

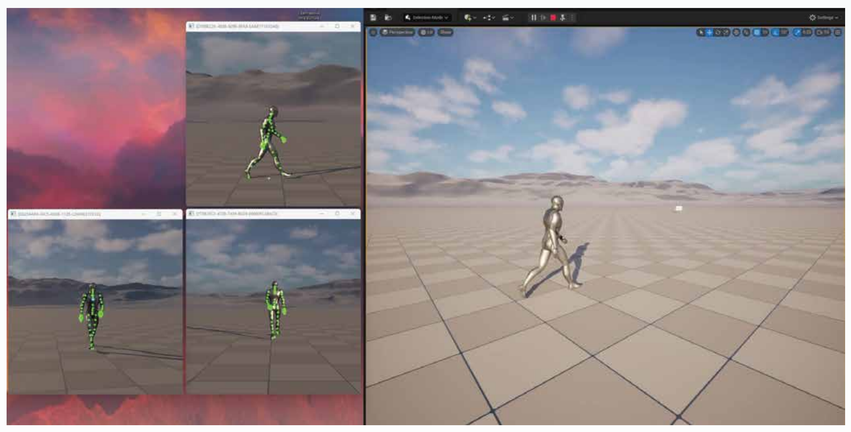
The typical workflow–(1) build 3D scenes in a game engine, (2) render and write its data to disk, and finally (3) use that static data as input to an algorithm or neural network–suffers from repeated CPU–GPU round-trips and I/O stalls.

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.

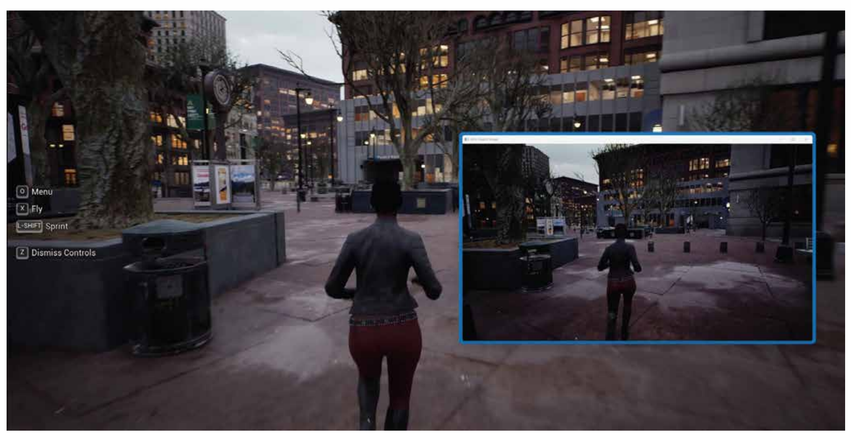
Our contribution is twofold: (i) generating an infinite amount of data in real-time on the GPU, and (ii) consuming it without storing it anywhere, bypassing the delays due to storage access.

Use Cases

Multi-camera sharing of images and character rig

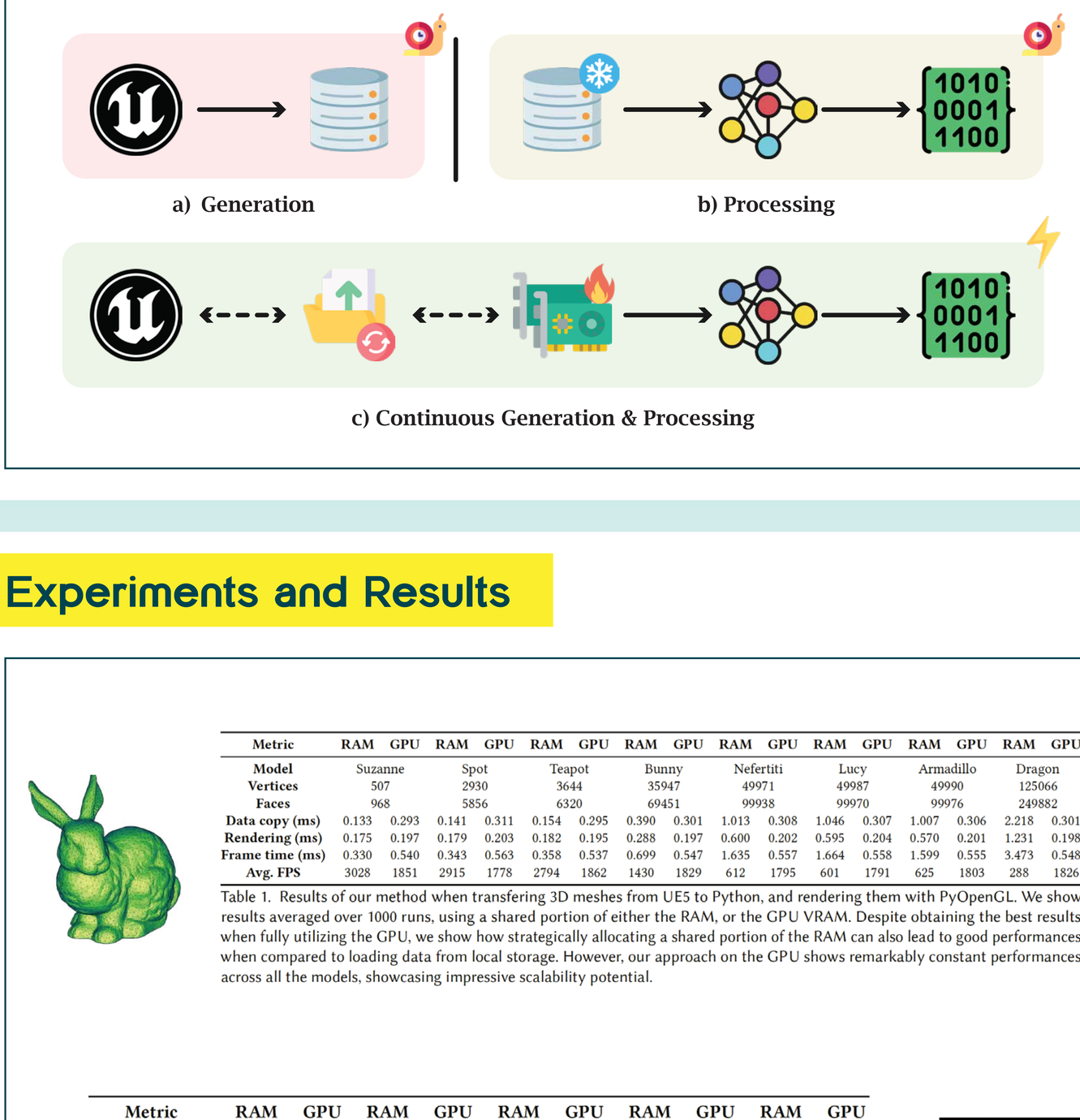


Real-time rendering and sharing of a complex scenario



Pipeline

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).



a) Generation

b) Processing

c) Continuous Generation & Processing

Experiments and Results

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.

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.

Acknowledgements

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

{daniele.dellapietra, gino.lanzohahn}@studenti.unitn.it, {nicola.garau}@unitn.it

SCAN ME