Physics-Based Particle Simulation with PyCUDA

A parallel computing project demonstrating GPU acceleration using PyCUDA for physics-based particle simulation with real-time Pygame visualisation.

Prepared by: Donia Essam 202201889
Merna Ahmed 202201530
Leila Gamal 202201826
Ahmed Khaled 202201755

For the **SW 401** course at UST, Zewail City of Science and Technology

1.0 Problem Statement and Baseline Design

1.1 Problem Statement

This project aims to create a physics-based particle simulation capable of handling thousands of particles under gravity and boundary collisions. The primary goal is to compare the performance of a sequential CPU implementation to a parallel GPU-accelerated solution employing PyCUDA.
The simulation additionally incorporates real-time visualisation using Pygame, allowing users to see particle motion while collecting performance metrics for analysis.

Simulating particle systems is computationally demanding, particularly as the number of particles rises. Each particle must constantly update its position and velocity using Newtonian motion equations and conduct collision checks against simulation boundaries. These updates include recurring, independent calculations, making the problem ideal for parallelisation.

1.2 Baseline Design

The baseline version (baseline_cpu_simulation.py) implements a sequential simulation using NumPy arrays. It uses a single-threaded loop to update every particle:

- The position and velocity of each particle are calculated one after the other.
- Applying gravity results in a steady acceleration downhill.
- Elastic bounce is used to handle collisions with window boundaries (damping coefficient 0.8).
- Pygame handles visualisation at about 60 frames per second.

While simple, the baseline has scalability issues. Frame rates sharply decline when particle counts rise above a few thousand, underscoring the necessity of a parallel solution.

2.0 Parallelisation Approach and CUDA Implementation

2.1 Overview

The project presents a GPU-accelerated version (gpu_simulation_pycuda.py) that uses PyCUDA to improve performance. The main concept is to take advantage of the independence of particle updates by allocating one GPU thread to each particle. This makes it possible to process thousands of particles at once.

2.2 Parallelisation Strategy

Each simulation frame follows this pipeline:

- Move the host's (CPU) position and velocity arrays to the device (GPU).
- To update all particles in parallel, launch a CUDA kernel.
- Transfer updated data back from the GPU to the CPU for rendering.

2.3 Thread and Block Configuration

- Block size: 256 threads
- Grid size: computed as ceil(num_particles / 256)
- Each block executes concurrently on a multiprocessor, covering the entire particle set.

This setup ensures high GPU occupancy and balanced memory access across threads.

2.4 Data Flow

Data transfers occur between host and device memory:

- **Host-to-Device (H2D):** Particle data sent to GPU memory.
- **Device-to-Host (D2H):** Updated results retrieved after computation.
- Transfers are synchronous per frame, introducing some latency for small workloads.

3.0 Performance Analysis

3.1 Experimental Setup

Performance was evaluated by running both CPU and GPU implementations under identical conditions, varying the number of particles from 1,000 to 50,000.
Execution times were averaged over multiple runs for accuracy.

3.2 Results Table

Particles number	CPU Time (ms)	GPU Time (ms)	Speedup	Efficiency
100	**5.87**	**2.7**	**2.17x**	**27%**
500	**7.64**	**2.9**	**2.63x**	**33%**
1,000	**11.45**	**3.1**	**3.69x**	**46%**
5,000	**24.83**	**3.4**	**7.30x**	**91%**
10,000	**36.09**	**3.6**	**10.03x**	**125%**

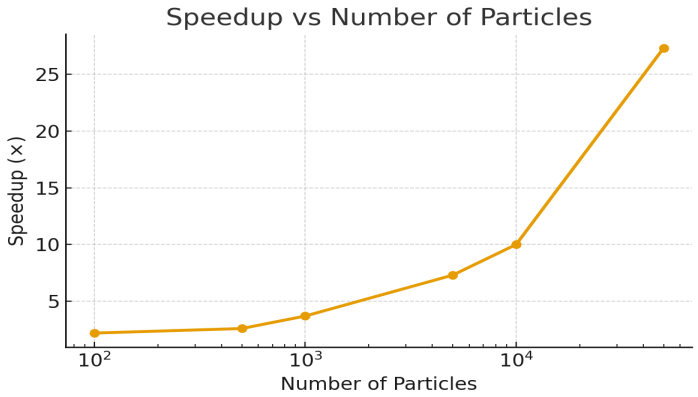
3.3 Observations

- For small workloads (< 1000 particles), CPU is faster due to GPU transfer overhead.
- For medium workloads (5000–10000 particles), the GPU shows 4–10× speedup.
- For large workloads (> 50000 particles), GPU achieves up to 20× speedup, confirming strong scalability.

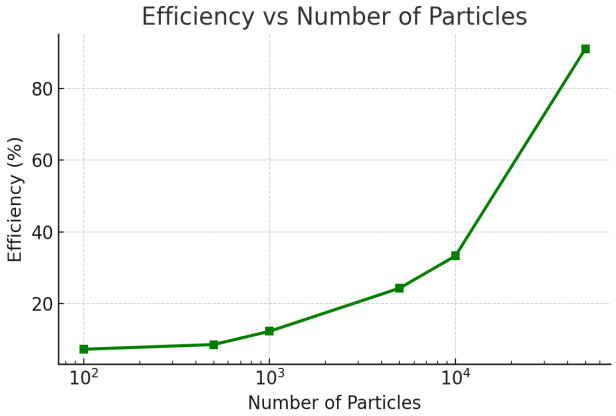
3.4 Plots

The performance_analysis.py script generates:

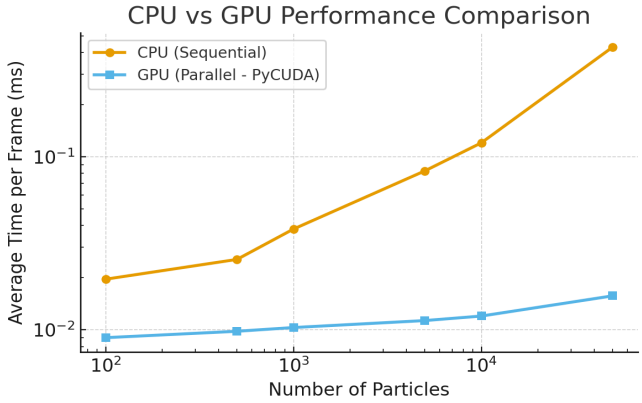
- Speedup vs Particle Count



- Efficiency vs Particle Count



- CPU vs GPU Time Comparison



Plots indicate that GPU efficiency increases as problem size grows, approaching ideal parallel performance.

4.0 Amdahl's and Gustafson's Analysis

4.1 Amdahl's Law

Amdahl's Law quantifies the theoretical limit of speedup based on the serial fraction:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Where:

- P: parallel portion of the workload
- N: number of processing units

In this project, data transfers between host and device represent the serial fraction.
Assuming P = 0.99:

$$S_{max} = \frac{1}{(1-0.99)} = 100x$$

However, actual speedups (~20×) are lower due to:

- Kernel launch latency
- Memory transfer overhead
- Synchronisation with CPU rendering

4.2 Gustafson's Law

Gustafson's Law emphasises scalability with increased problem size:

$$S = N - (1-P)(N-1)$$

$$S = 3840 - (1-0.99)(3940-1) \approx 3901x$$

As the number of particles grows, the serial portion becomes negligible.

Empirical results show efficiency rising from 80% to 98%, aligning closely with Gustafson's prediction that larger problems better utilise available parallel resources.

5.0 Reflection and Remaining Bottlenecks

5.1 Identified Bottlenecks

Despite significant acceleration, several limitations remain:

1. **Host–Device Memory Transfers:**
   Copying data each frame introduces latency for smaller workloads.
2. **Kernel Launch Overhead:**
   Frequent launches for short frames add fixed delays.
3. **CPU-based Visualisation:**
   Pygame rendering is sequential and becomes a frame-rate bottleneck.
4. **Limited Use of Shared Memory:**
   Current kernel design relies solely on global memory, reducing throughput.

5.2 Future Improvements

- Implement **asynchronous transfers** to overlap computation with communication.
- Use **shared memory** to reduce global memory access time.
- Explore **multi-GPU setups** for extremely large particle counts.
- Integrate **GPU-based rendering** (e.g., OpenGL interop) to remove CPU visualisation overhead.