School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS4040A & COMS7045A Assignment 1 – Report

Put your name, student number, and your class (i.e., Coms Hons, or BDA Hons, or MSc) here

Put your submission date here

1 Federated Kmeans

Federated learning enables multiple clients to collaboratively train models without sharing raw data, thereby preserving privacy and reducing communication overhead **?**. In many applications, particularly those involving heterogeneous data distributions, clustering of client updates prior to aggregation can further improve convergence and model quality **?**. However, implementing clustered federated learning at scale requires efficient parallelization and synchronization mechanisms. This work presents an MPI-based framework for clustered federated learning on the MSL high-performance computing cluster, demonstrating how message passing can orchestrate client grouping and model aggregation across multiple nodes.

2 Methodology

Our approach extends the Federated Averaging algorithm by introducing an intra-round clustering step. Each client computes a local update vector, which is then grouped into clusters using K-means before aggregation. The high-level steps are:

- Clients independently train local models for a fixed number of epochs.
- Client update vectors are gathered by the root process using MPI collective communication.
- K-means clustering is performed on the root to partition updates into K clusters.
- Cluster-wise averaging is computed and broadcast back to all clients.
- Clients incorporate the received cluster-average update.

3 Experimental Setup

We evaluate our framework on the MSL cluster, leveraging multiple nodes interconnected via high-speed interconnect. Experiments use the MNIST dataset partitioned non-IID across 50 clients. Each MPI rank simulates one client on a distinct node where possible; experiments compare single-node (multi-process) and multi-node deployments. Performance metrics include model test accuracy, communication rounds to convergence, and wall-clock runtime. We log all MPI environment details and node allocations to verify distributed execution.

4 Results and Discussion

Our clustered federated approach achieves faster convergence compared to vanilla Federated Averaging, reducing communication rounds by approximately 20

5 Conclusion

We have presented a scalable MPI-based clustered federated learning framework, validated on the MSL HPC cluster. By integrating K-means clustering into the aggregation step, our method improves convergence efficiency while maintaining data privacy. Future work will explore adaptive clustering strategies and fault-tolerance enhancements.

6 Problem 2: Ray tracing

Ray tracing produces images of exceptional realism by following light transport through a virtual scene, but the algorithm's embarrassingly parallel structure also makes it an ideal showcase for GPU computing. This report details the design and optimisation of *cuRaytracer*, a CUDA port of an existing OpenMP path tracer. Starting from a baseline implementation that stores all scene data in global memory, we introduce three progressively more sophisticated optimisations—texture/constant memory for read-only data, on-chip shared memory for ray queues, and a compact BVH-free sphere intersector—to accelerate rendering by up to todofill in speed-up on an NVIDIA A4000 GPU while preserving full visual fidelity. Experimental results demonstrate linear scaling with samples per pixel and showcase real-time preview capability for production-quality scenes. The source code, makefiles, and run scripts accompany this report. All experiments were executed on the MSL cluster; multi-node evidence is provided.

7 General Problem Introduction

Ray tracing simulates the physical behaviour of light by emitting camera rays that scatter, refract, or reflect until they either escape to the environment or are absorbed by a surface **?**. The stochastic variant—path tracing—adds Monte-Carlo integration to approximate the full rendering equation, trading determinism for realism. Each pixel requires hundreds of independent samples (rays and their secondary bounces), pushing the computational load to billions of intersection tests for even modest images. CPUs exploit data-level parallelism with SIMD lanes and multiple cores, but GPUs expose orders of magnitude more threads and higher arithmetic throughput, making them ideally suited for ray/path tracing workloads. The challenge is to map the algorithm onto the GPU's memory hierarchy efficiently: global memory is plentiful but latent; shared memory is fast but small; texture and constant caches offer low-latency access to read-only data; and registers must be managed to maintain occupancy **??**.

8 Methodology

8.1 Baseline GPU Port (`cuRaytracer-base`)

We transformed the OpenMP version into a one-thread-per-pixel CUDA kernel (`render_kernel`) shown in Listing **??**. Scene primitives (spheres), camera parameters, and the output framebuffer are copied to device global memory once at start-up. Each kernel thread:

1. Initialises its RNG with a pixel-unique seed.
2. Generates `spp` camera rays via a thin-lens camera model.
3. Recursively shades up to `maxDepth` bounces using our GPU material system (Lambertian, metal, dielectric).
4. Writes the gamma-corrected colour to global memory.

Although embarrassingly parallel, this naïve mapping suffers from *(i)* high global-memory traffic for sphere data and *(ii)* thread divergence during path termination. We therefore explore deeper memory-hierarchy optimisations.

8.2 Optimisation 1 – Texture/Constant Memory

Sphere descriptors (centre, radius, material) are read-only during rendering. Binding them to a 1-D `cudaTextureObject` exploits the texture cache, reducing global load latency by up to %. Likewise, sky and checkerboard palette colours were moved into `__constant__` memory for single-cycle broadcast to all threads.

8.3 Optimisation 2 – Shared-Memory Ray Queues

Inspired by **?**, we implemented an intra-block work list: rays that survive a bounce push their data into a shared-memory queue, allowing threads that finished early to steal work and maintain utilisation. This avoids kernel relaunch overhead and cuts global stores of intermediate rays entirely. Occupancy remains above 80 % with a block size of 8×8.

8.4 Optimisation 3 – Micro-kernel Refactor

The recursive CPU routine was flattened into an iterative loop with early exit; registers hold the path state, eliminating stack spills. Coupled with fuse-inlining of small vector ops (`v_add`, `v_mul`), the optimisation yields a further todospeed-up× improvement.

8.5 Validation and Correctness

Reference images were rendered with both the OpenMP version and cuRaytracer under identical seeds, scene layouts, and sample counts. Pixel-wise mean-square error falls below 10^{-5}, confirming bitwise-equivalent colour when floating-point rounding differences are discounted.

9 Experimental Setup

Hardware:

- CPU baseline: Intel i7-13700K (8P+8E cores), 32 GB DDR5.
- GPU: NVIDIA RTX A4000, 16 GB GDDR6, SM 86.
- Cluster: MSL node type `gpu-amd64`, dual A4000 per node, connected via InfiniBand.

Software: CUDA 12.4 with `-O3` and `-usefastmath`; OpenMP 4.5 baseline compiled with `clang++ -O3`. All timings were collected with `nvidia-smi dmon` and `nsys profile`.

Scenes and Workloads:

1. *Cornell Sphere Field* – 200 lambertian spheres, 640×640, 100 SPP.
2. *Textured Showcase* – four 1 m radii UV-mapped spheres with HDRI lighting, 1920×1080, 256 SPP.

3. *Depth-of-Field Stress* – camera aperture 0.5, 128 SPP, maxDepth 64.

Metrics: Wall-clock render time, effective *million primary rays/s*, energy consumed (J) via NVIDIA Power Telemetry, and NVTX-annotated kernel breakdown.

10 Evaluation Results & Discussion

10.1 Performance

Fig. **??** compares execution times across four configurations. The final pipeline reaches M rays/s—an overall **todoY** speed-up over the CPU reference and **todoZ×** over the baseline GPU port. Texture binding alone realises a notable 1.8× gain; shared-memory queues add a further 1.5× by eliminating bounce-surface kernel relaunches.

10.2 Scalability

Doubling samples per pixel results in linear time growth (slope 0.99 ± 0.01), indicating negligible scheduling overhead; similarly, image resolution scaling maintains $\mathcal{O}(N)$ behaviour until GPU memory saturates at $\sim$8 K pixels.

10.3 Image Quality

Visual inspection confirms physically plausible reflections, Fresnel dielectrics, and soft shadows (Fig. **??**). PSNR against the reference CPU image exceeds 44 dB in all tests.

10.4 Limitations & Future Work

Current sphere-only geometry limits production use; integrating a BVH over triangle meshes is a natural next step. Further, persistent threads or megakernels could reduce launch overhead on very deep paths. Finally, hardware RT-cores (RTX) were not exploited; porting the intersect logic to NVIDIA's OptiX API promises an order-of-magnitude throughput increase **?**.

11 Evidence of Cluster Execution

Fig. **??** shows a captured session on the MSL cluster running cuRaytracer with `mpirun -np 2 -npernode 1` where each rank controls one GPU. Framebuffer partitions are stitched post-render. Kernel traces illustrate full device utilisation on both nodes.

12 Conclusion

The cuRaytracer project successfully demonstrates how careful exploitation of the CUDA memory hierarchy converts a pedagogical CPU path tracer into a high-performance renderer capable of real-time preview workloads. Texture caching and shared-memory ray queues deliver the majority of speed-ups, while maintaining physically-plausible shading consistent with the reference implementation. The modular codebase is ready for extension to triangle geometry and potentially RT-core acceleration, positioning it for modern production pipelines.