COMS4040A & COMS7045A Project – 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 [2]. In many applications, particularly those involving heterogeneous data distributions, clustering of client updates prior to aggregation can further improve convergence and model quality [1]. 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.

Hardware and MPI Environment

- Experiments are run on the MSL HPC cluster using MPI across multiple nodes.
- Each MPI rank corresponds to one federated “client” process; rank 0 acts as the server/aggregator.

Data Preparation

- The CIFAR-10 dataset is loaded via a Python script.
- Four client shards (A–D) are created with non-IID class skews:
  - Client A favors labels 0,1,2 with 70% probability over 5 000 training and 1 000 test samples.
  - Client B favors 3,4,5 with 70% over 3 000 train / 500 test.
  - Client C favors 6,7 with 70% over 2 000 train / 400 test.
  - Client D favors 8,9 with 70% over 1 000 train / 200 test. Each shard is written to a binary file containing an integer pair (n, D) followed by row-major flattened feature vectors.

2.1 Federated K-Means Workflow

Server (rank 0) initializes K = 10 random centroids of dimension D = H·W·C = 32·32·3 = 3 072.

Centroids are broadcast to all client ranks.

Local Updates (Rounds 0–19):

Each worker (rank > 0) loads its local training shard and runs K-Means for 5 local iterations on a random minibatch of size 100.

During each minibatch iteration, each sample is assigned to its nearest centroid and its contribution (sum and count) is accumulated.

2.2 Server Aggregation:

After local updates, workers send their per-centroid sums and counts to the server via MPI_Reduce.

The server computes the global weighted average for each centroid, then broadcasts the updated centroids back to all workers.

After 20 communication rounds, the server saves the final centroid matrix to centroids.bin for later evaluation.

2.3 Evaluation Phase

A separate MPI program loads the saved centroids (centroids.bin) on rank 0 and broadcasts them.

Each worker loads its test shard and computes:

Local inertia (sum of squared distances to the nearest centroid)

Cluster counts (number of test samples assigned to each centroid)

Both metrics are reduced to the server, which reports:

Total sample count, global inertia, average distance ($\sqrt{inertia/total}$), and per-cluster counts.

2.4 Metrics and Logging

Convergence Behavior: Tracked by printing "[round r] done" on the server after each aggregation.

Final Centroid Preview: Server logs the first few centroid coordinates.

Evaluation Outputs: Overall inertia and distribution of test samples across clusters.

MPI Diagnostics: Each rank reports successful data loading and dimensionality.

This setup ensures a clear federated K-Means workflow—from non-IID data sharding through iterative, communication-efficient centroid updates to quantitative evaluation on held-out test shards.

2.5 Results and Discussion

Federated nodes (ranks)

Rank 1 (@mscluster45): 5 000 samples

Rank 2 (@mscluster46): 3 000 samples

Rank 3 (@mscluster47): 2 000 samples

Rank 4 (@mscluster48): 1 000 samples

Training

Total of 20 global rounds (0. . . 19), each round every rank completed its local update.

Final centroids written to centroids.bin.

2. Evaluation on 2 100 held-out samples Parameters

K = 10 clusters

Dimensionality D = 3 072

Overall quality

Inertia (sum of squared distances to nearest centroid): 963 171

Average distance per sample: 21.42

Cluster membership

C0: 117 samples (5.6

C1: 0

C2: 4 (0.2

C3: 1 979 (94.2

C4–C9: 0

3. Key Observations Severe imbalance in cluster usage

Only 3 of 10 centroids ended up owning any points; one cluster dominated (94% of samples).

Six centroids are “empty,” indicating they either never moved toward any data or lost membership entirely.

Cluster collapse / poor separation

The collapse into essentially one large cluster (C3) plus two tiny ones (C0, C2) suggests

suboptimal initialization (e.g. random seed unlucky),

K too large for the data’s intrinsic grouping, or

federated updates overwriting finer local structure.

High inertia and average distance

With most points packed into a single cluster, distances to that centroid inflate both inertia and the mean distance.

3 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.

4 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.

5 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 [6]. 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 [4, 5].

6 Methodology

6.1 Baseline GPU Port (`cuRaytracer-base`)

We transformed the OpenMP version into a one-thread-per-pixel CUDA kernel (`render_kernel`) shown in 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.

1. Experimental Setup Scene parameters

Resolution: 640 x 640

Samples per pixel (spp): 100

Max path-trace depth: 50 bounces

Camera: “look-from” at (13, 2, 3) looking at the origin, 20° FOV, aperture = 0.1, focus distance = 10

Data

A procedurally generated set of up to 64 spheres with randomized materials (Lambertian, Metal, Dielectric), created on the host via randomSpheres2(...).

Hardware and timing

The host launches a single CUDA kernel and times it using cudaEventRecord/cudaEventElapsedTime.

Block size: 8×8 threads; grid size computed to cover the full image.

The output buffer is allocated once on the device (cudaMalloc) and copied back at the end to produce a PPM image.

2. Code Adjustments for GPU Acceleration a) Data transfer and initialization Scene upload

Host std::vector<Sphere> → std::vector<SphereGPU> (a POD struct)

Copy to device:

Base version uses cudaMemcpy into a global device pointer (ctx.d_spheres).

Shared-memory version copies into __constant__ SphereGPU d_spheres[...], then at kernel launch each block threads copy from d_spheres into fast shared memory.

Environment map (HDR skybox)

On host, load HDR with stbi_loadf → pack into a float4 array.

Upload via cudaMallocArray + cudaMemcpy2DToArray → create a cudaTextureObject_t with wrap filtering and normalized coords.

Pass that texture object into the kernel so each ray miss can sample a realistic sky.

b) Kernel entry point Annotated __global__ void render_kernel(...).

Maps (blockIdx, threadIdx) to (x,y) pixel; early-exit if outside image bounds.

Each thread maintains its own uint32_t rng seed based on pixel index, used by rng_next() for per-sample jitter.

c) Ray-tracing loop in device code Base vs. shared

Base: calls a device function ray_color(...) that scatters rays through the scene array in global memory.

Shared: first warp of each block copies the constant-memory sphere list into an extern __shared__ SphereGPU s_sph[] array for all other threads to reuse (dram→SMEM).

Ground plane

Checked analytically: if ray points downward, compute tPlane, then checkerboard via floorf(x)+floorf(z) parity.

Sphere hits

Loops over up to ns spheres and tests with hit_sphere(); maintains the nearest hit in a local HitRecord.

Material shading

Lambertian, metal, dielectric handled entirely on-device (no host callbacks).

Sky sampling

If no hit, either sample the HDR texture with tex2D<float4>() or fall back to a simple vertical gradient.

d) Memory and performance optimizations Constant memory for small, read-only sphere list.

Shared memory preload in the "shared" variant to cut global-memory traffic.

cubin launch parameters: uses rsqrtf() and fused math (e.g. sqrtf()) to speed up gamma correction.

Texture memory for environment map—leveraging the read-only cache and hardware filtering.

e) Host-side orchestration prepare_world(): builds the C scene and returns camera, sphere list, framebuffer pointer.

render_init_cuda(...): allocates device buffers, copies sphere data.

main():

Optionally loads HDR argv[2], calls uploadEnvMap() and uploadScene().

Records start event.

Launches render_kernel«<grid,block,shmem»>(...).

cudaEventSynchronize, cudaEventElapsedTime → prints "took XX ms".

Copies back the image and writes out via writeToPPM().

In short, the original ray-tracing loop and scene setup were uplifted into CUDA by:

Splitting host and device responsibilities (scene build vs. ray march).

Copying all geometry and HDR data into GPU memory (global/constant/texture).

Converting recursive/iterative ray logic into device functions called inside a single monolithic __global__ kernel.

Optimizing memory access via constant and shared memory, and using the texture cache for environment sampling.

Parallelizing per-pixel work over thousands of CUDA threads, each carrying its own RNG state and color accumulator.

Let me know if you'd like deeper detail on any specific part of that pipeline!

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.

6.2 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.

6.3 Optimisation 2 – Shared-Memory Ray Queues

Inspired by [5], 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.

6.4 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.

6.5 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.

7 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.

8 Evaluation Results & Discussion

8.1 Performance

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.

8.2 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.

8.3 Image Quality

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

8.4 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 [3].

References

[1] Vishek Ghosh et al. “An Efficient Framework for Clustered Federated Learning”. In: 33 (2020), pp. 19586–19597.

[2] Brendan McMahan et al. “Communication-Efficient Learning of Deep Networks from Decentralized Data”. In: (2017), pp. 1273–1282.

[3] NVIDIA. *NVIDIA OptiX 7 Programming Guide*. https://raytracing-docs.nvidia.com/. 2024.

[4] NVIDIA. *Using Shared Memory in CUDA C/C++*. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/. 2013.

[5] Thomas Pitkin. “GPU Ray Tracing with CUDA”. MA thesis. Eastern Washington University, 2014. URL: https://dc.ewu.edu/cgi/viewcontent.cgi?article=2092&context=theses.

[6] Peter Shirley. *Ray Tracing in One Weekend*. https://raytracing.github.io/books/RayTracingInOneWeekend.html. 2016.