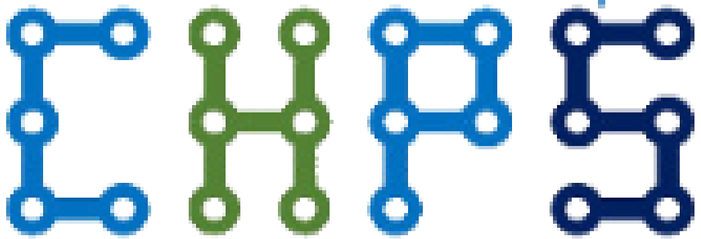
2024-2025

URCA

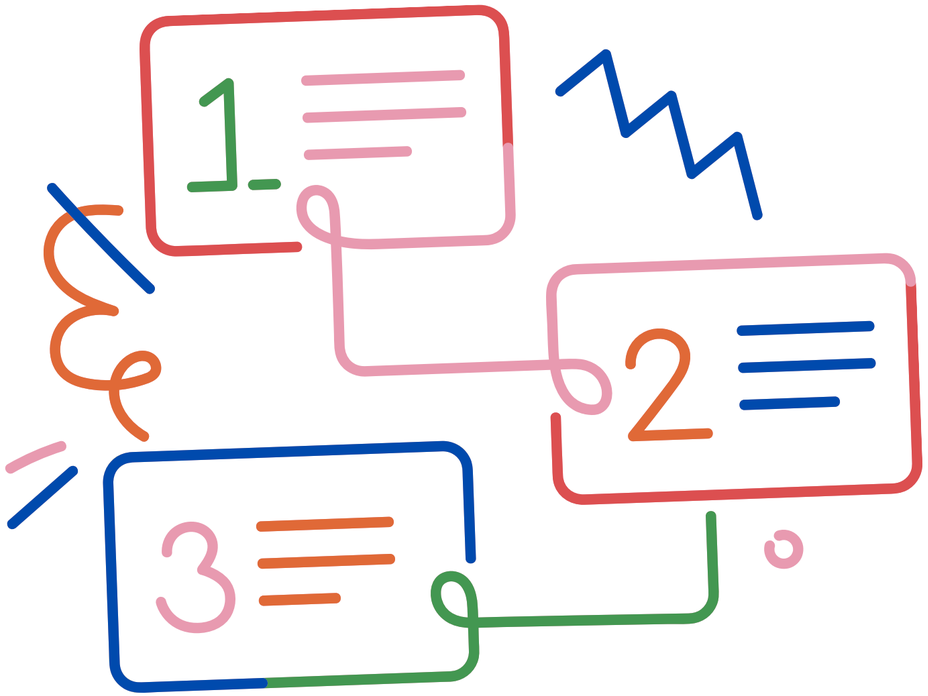
RayTracing CHPS0802

Master informatique



Présenté par Wejdane BOUCHHIOUA

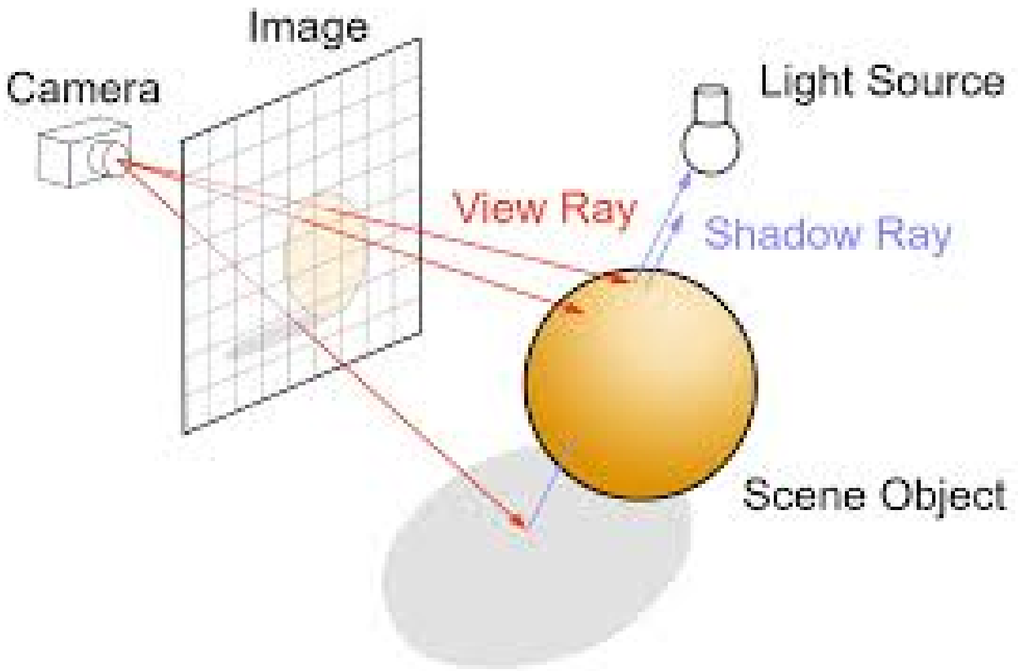
Sommaire



Ray Tracing?

- Méthode de rendu 3D réaliste
- Simule le trajet des rayons lumineux
- Chaque pixel : un rayon émis depuis la caméra
- Intersection avec les objets

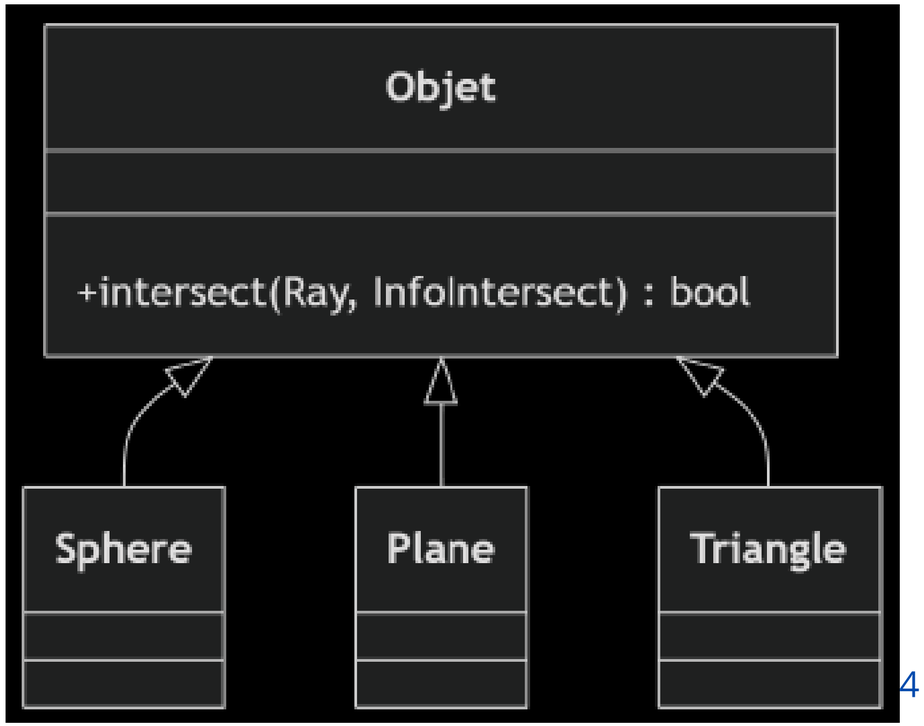
⇒ calcul de la couleur finale



Architecture C++ Séquentielle

Conception orientée objet :

- Objet (classe abstraite)
  Sphere, Triangle, Plane héritent et implémentent intersect()
- Material, Ray, Camera, Color
- Scene gère les objets et les lumières ray_color() récursive jusqu'à profondeur max.



Spécificités du code C++

- Usage de std::shared_ptr pour les matériaux et les objets
- ray_color() récursive avec contrôle de profondeur
- Export en format .ppm

❗ Limites CPU : rendu long, traitement séquentiel

Passage au GPU

Le concept du Ray Tracing est hautement parallélisable

- Chaque pixel est indépendant
- Chaque rayon peut être calculé en thread GPU

=> CUDA combine :

parallélisme massif

+

contrôle mémoire

+

calcul intensif

Etapes passage au GPU

1. Construction scène (CPU)
2. Transfert vers VRAM (GPU)
3. Exécution kernel render_kernel
   (1 thread = 1 pixel)
4. Récupération du résultat
5. Sauvegarde image .ppm

Défis passage au GPU

- ❌ Polymorphisme non supporté
- ❌ Récursivité coûteuse
- ❌ Rand() pas thread-safe
- ❌ STL (shared_ptr) et pointeurs dynamiques interdits

Solutions techniques en CUDA

- Polymorphisme transformé en Structs + Enum + Union

```
enum ObjectType { SPHERE, TRIANGLE,PLANE };

struct Object_GPU {
     ObjectType type;
     union { Sphere s; Triangle t; Plane p };
     int material_id;
};
```

- Récursivité transformée boucle itérative ray_color()
- Random → curandState par thread
- Gestion mémoire → cudaMalloc, cudaMemcpy, CudaBuffer<T>

Solutions techniques en CUDA

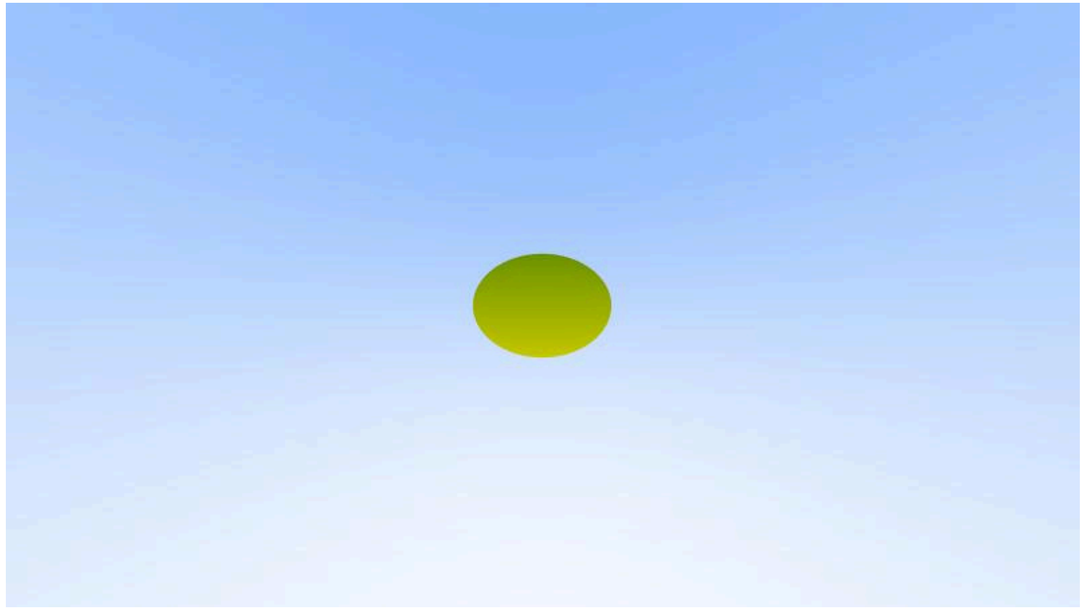
Organisation des threads:

```
dim3 blocks(16,16);
dim3 grid((width+15)/16, (height+15)/16);
render_kernel<<<grid, blocks>>>(...);
```

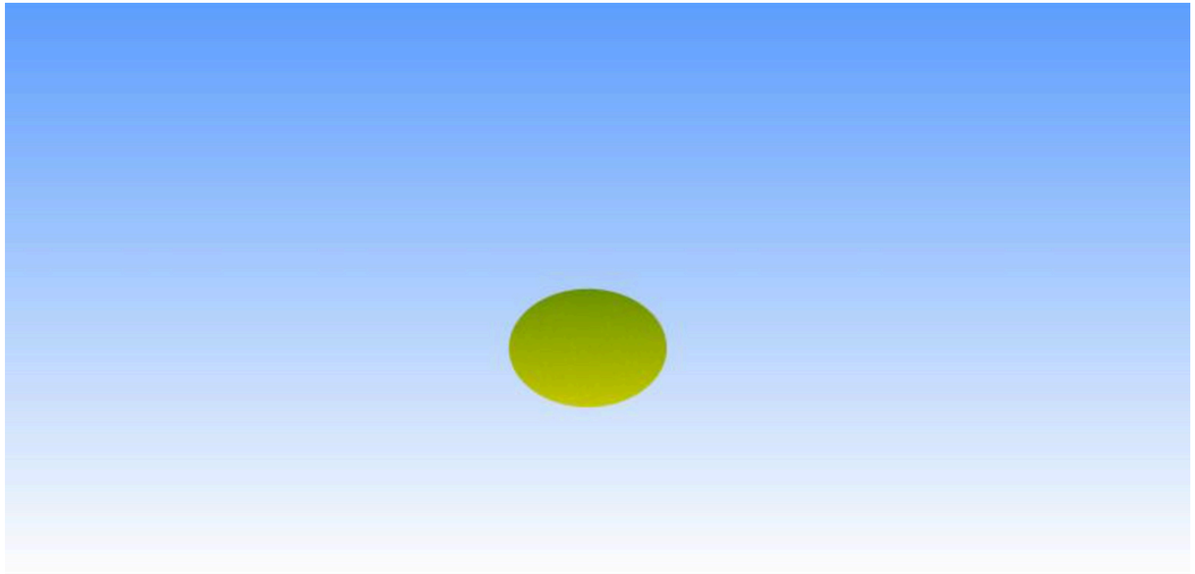
=> Chaque thread GPU = 1 pixel traité
=> Accès coalescés à la mémoire = meilleur débit

Résultats TEST1

CPU

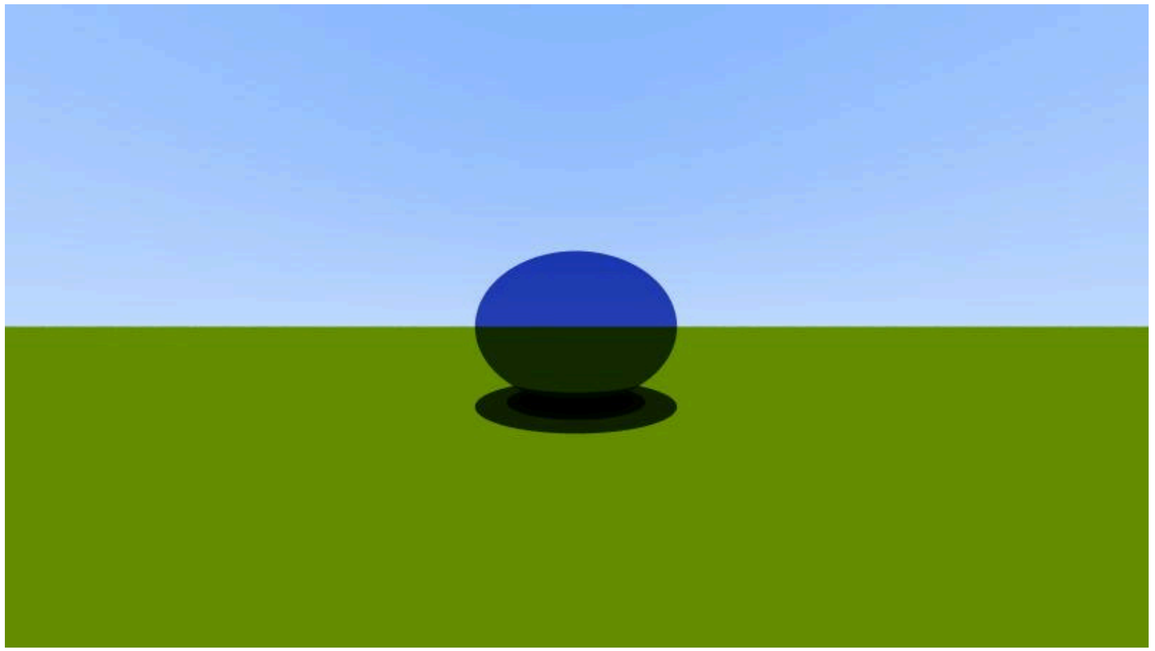


GPU

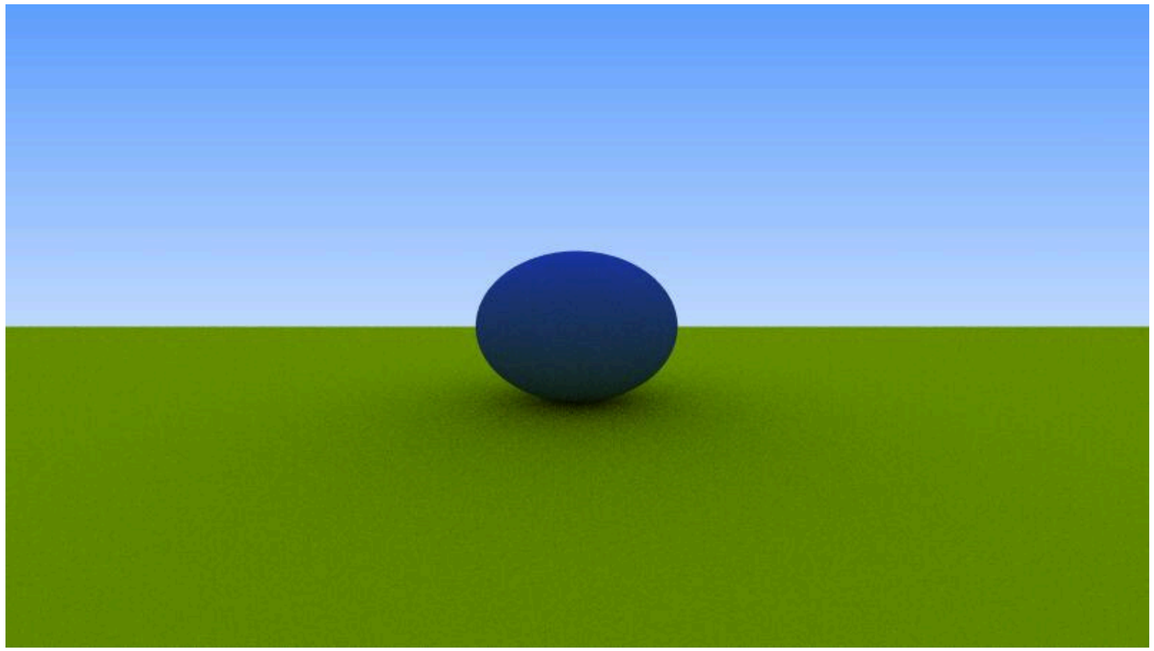


Résultats TEST 2

CPU

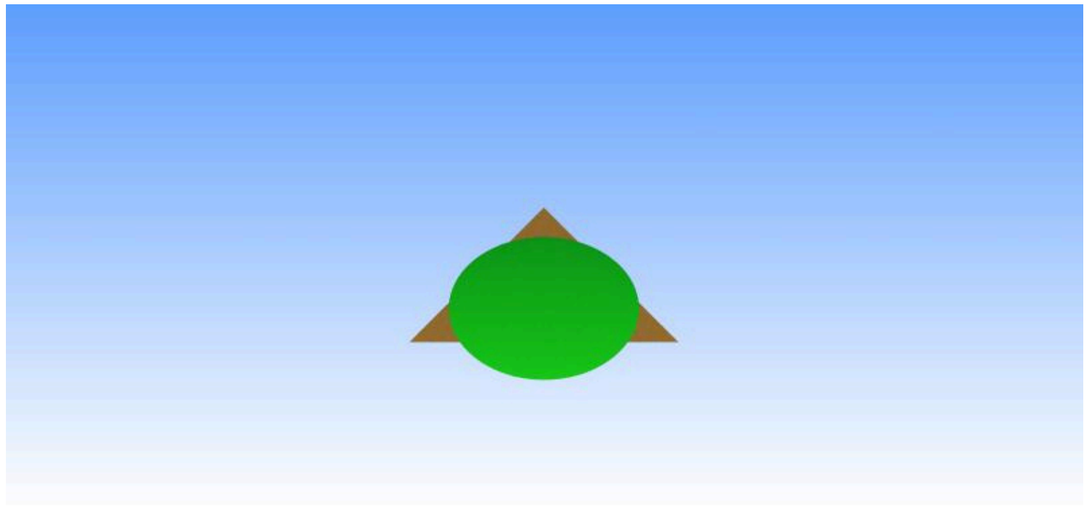


GPU

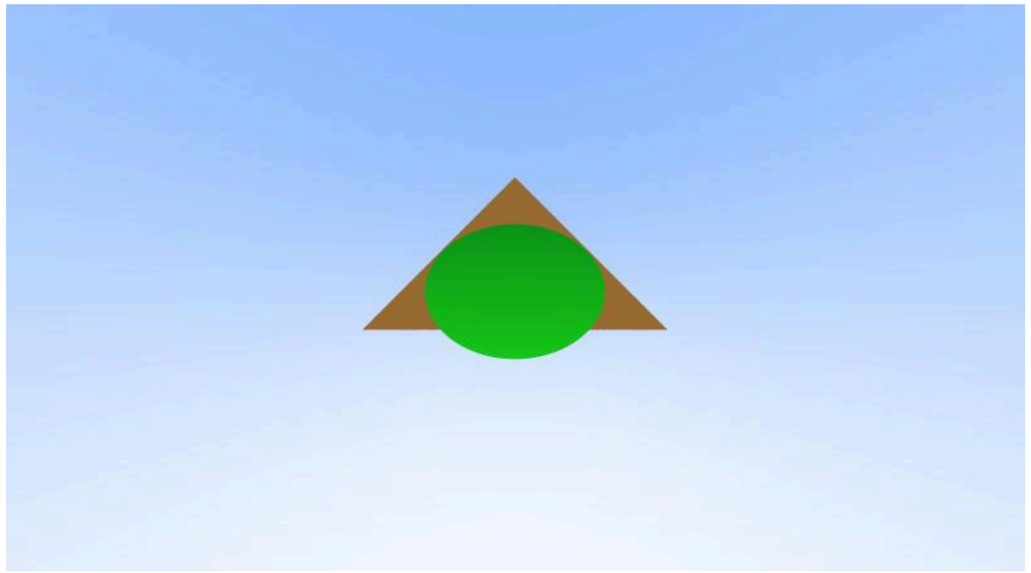


Résultats TEST 3

CPU

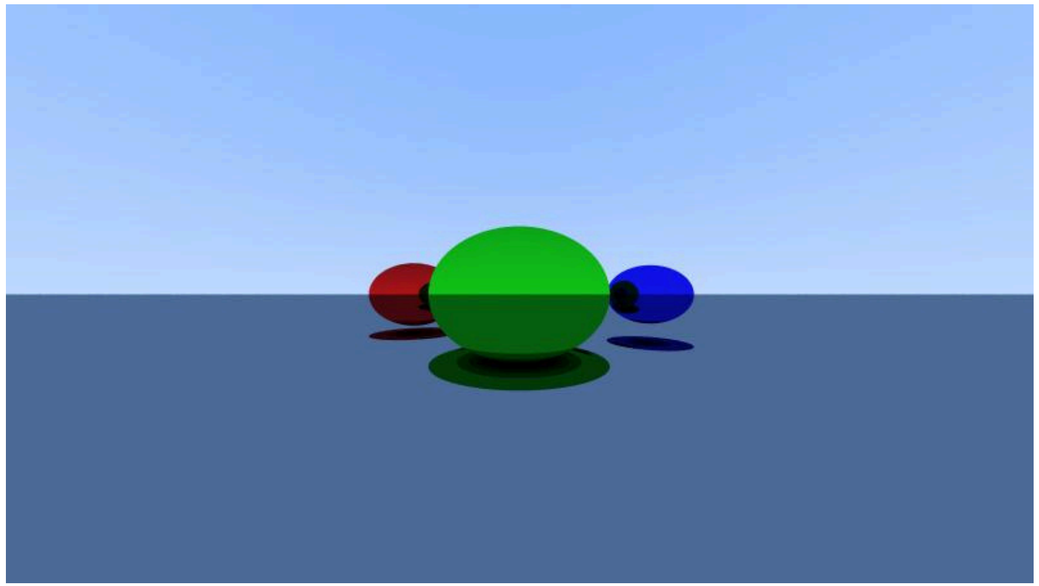


GPU

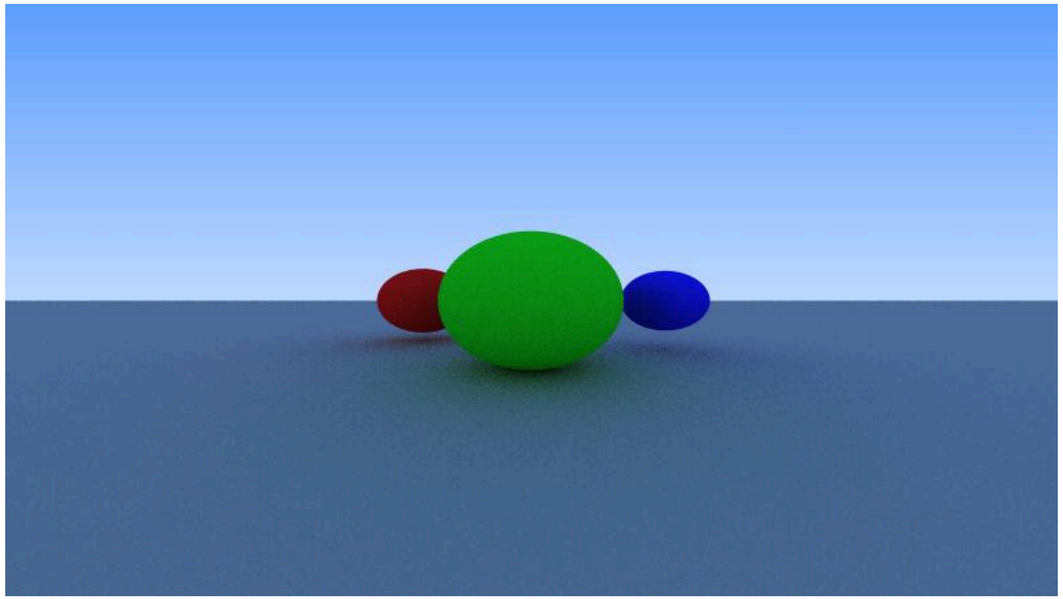


Résultats TEST 4

CPU

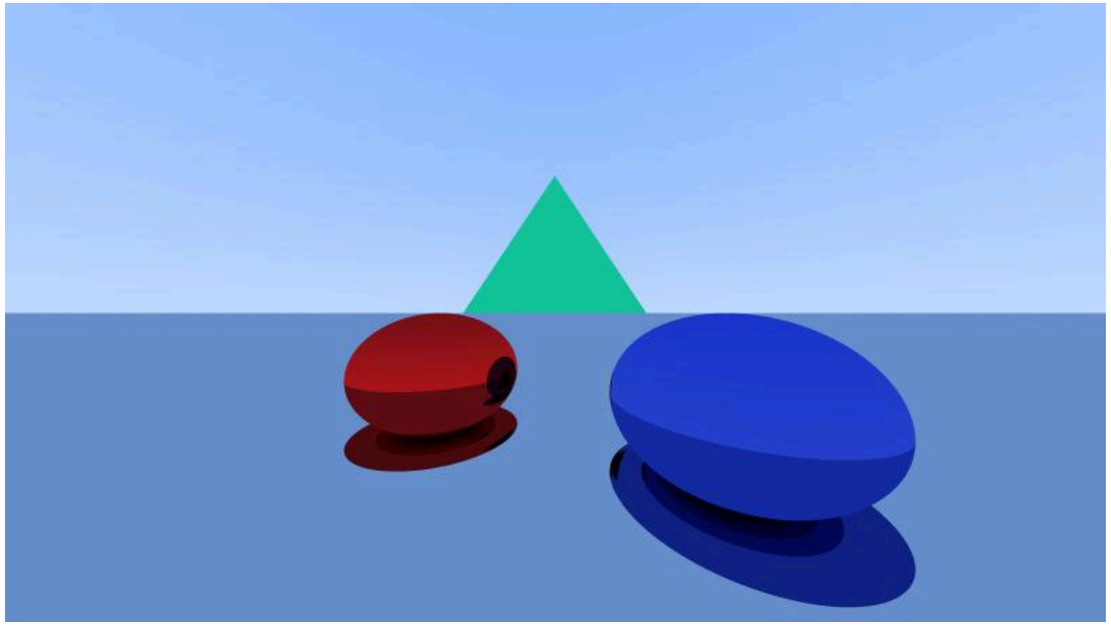


GPU

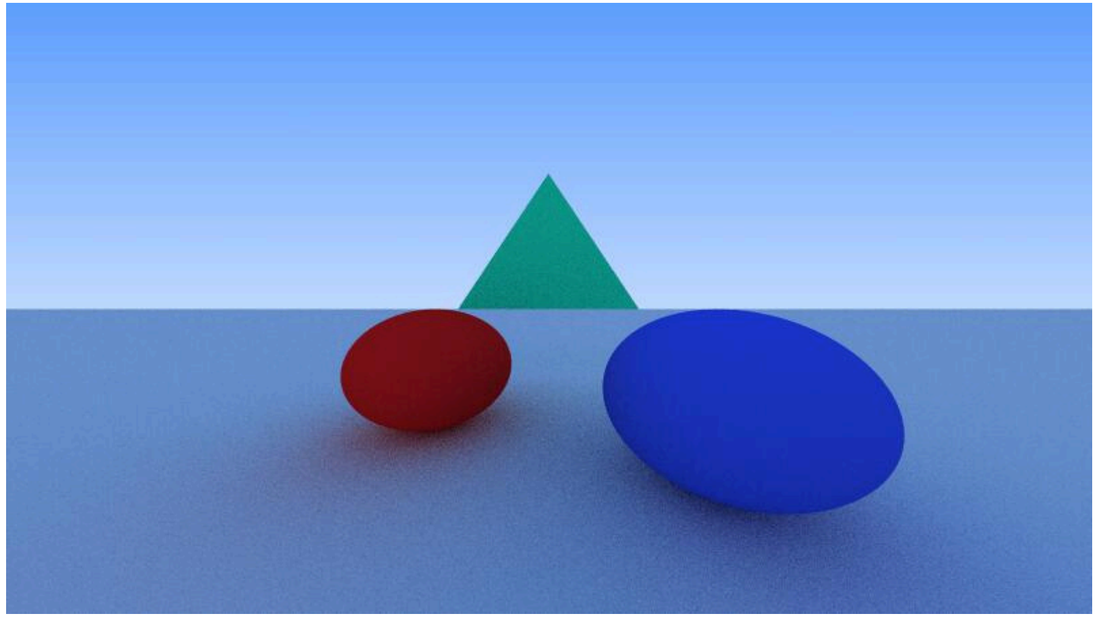


Résultats TEST 5

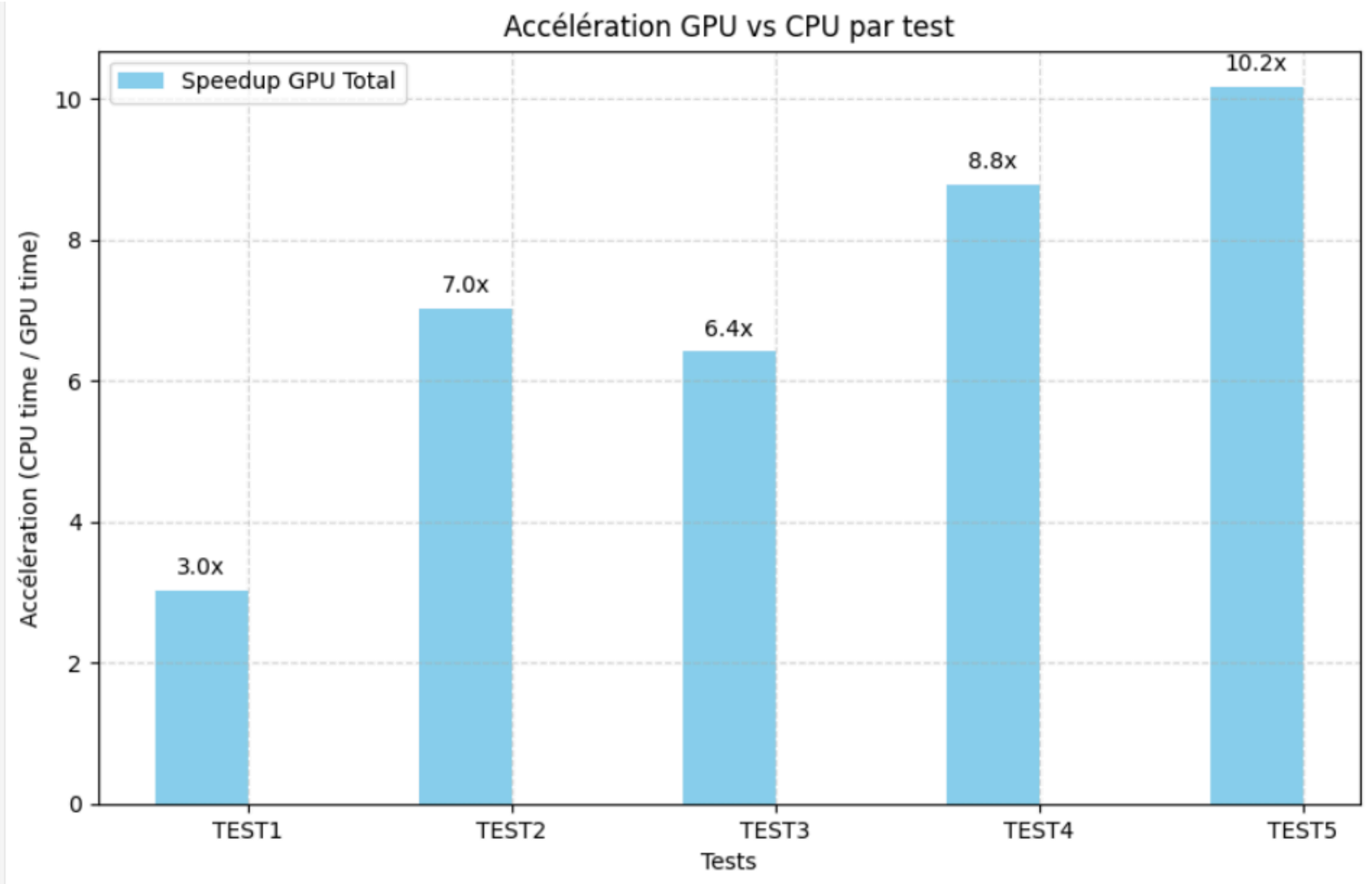
CPU



GPU

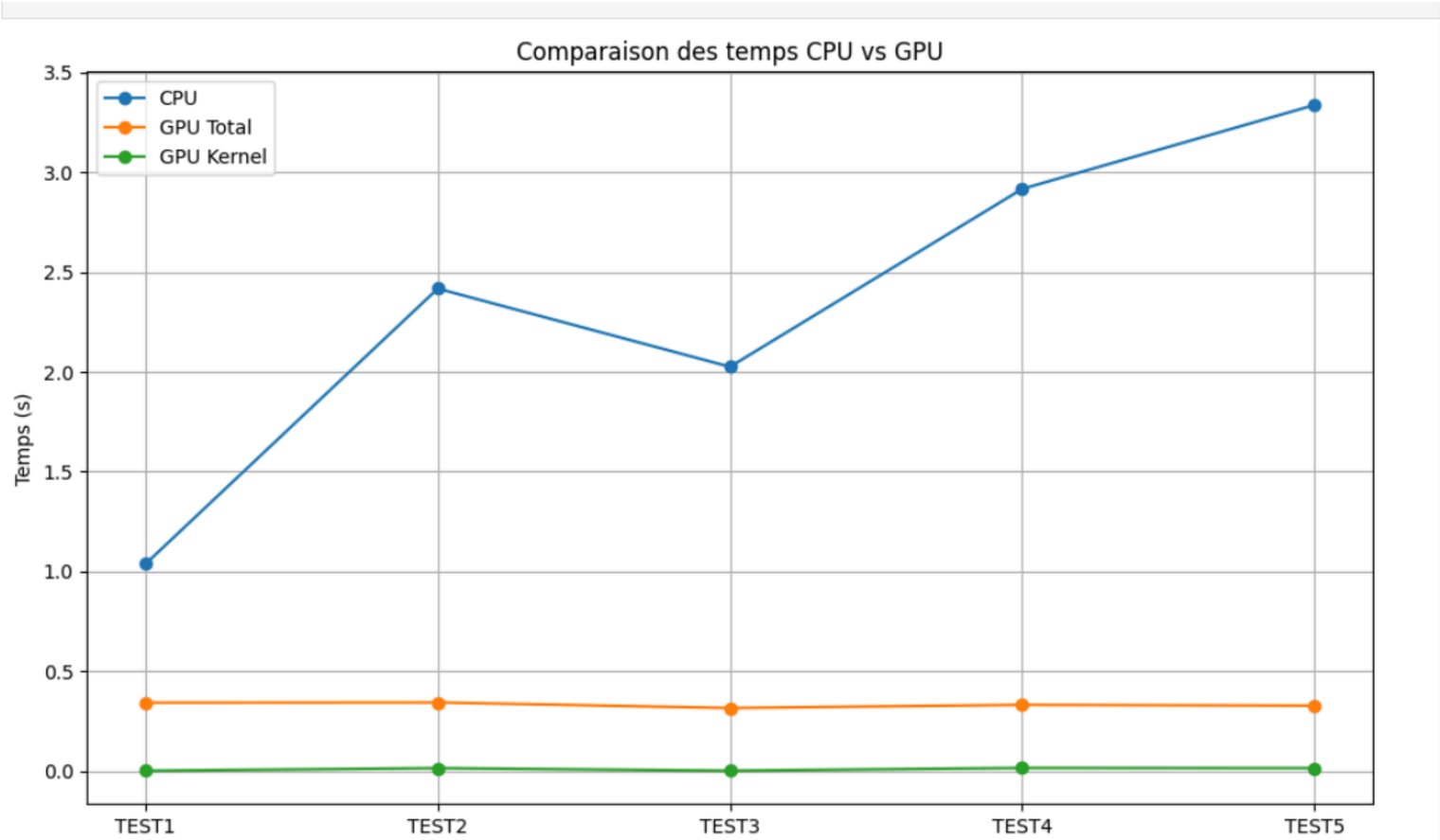


Analyse des résultats



Analyse des résultats

Nous pouvons remarquer que:
Plus les images sont plus complexe, le temps d'exécution est plus important.
En GPU, les tests sont environ 300 -340 ms pour les différentes images.

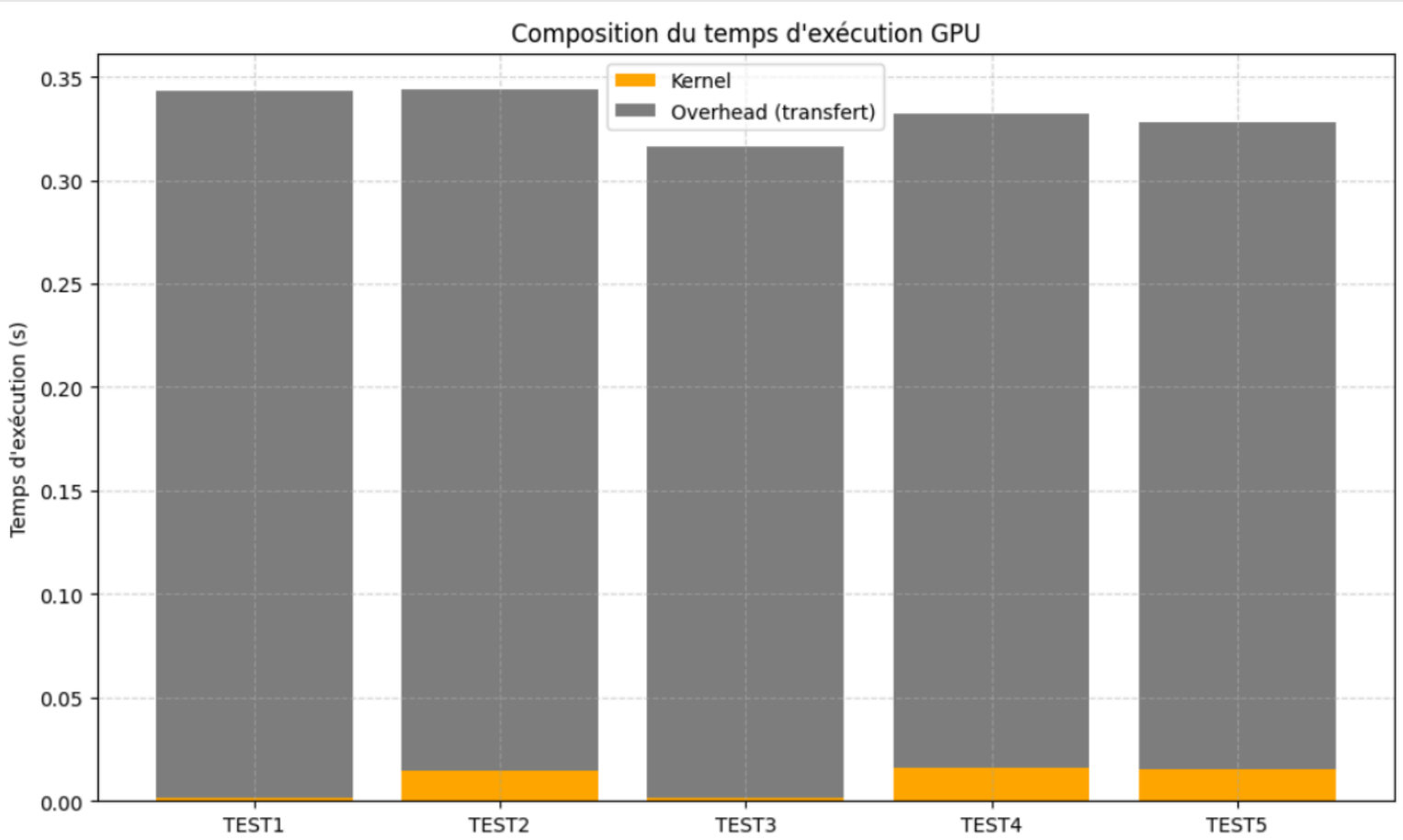


Analyse de résultats

Nous avonc un overhead GPU (transfert) est élevé

=> cela signifie que le temps passé en dehors du calcul GPU

=>principalement les transferts de données entre le CPU (host) et le GPU (device) — est important par rapport au temps de calcul réel (kernel)



Conclusion

✓Ray Tracer fonctionnel sur CPU et GPU

✓ Migration maîtrisée malgré contraintes CUDA

✓ Accélération notable (7× plus rapide)

✓ Architecture propre, modulaire et extensible

Merci pour votre attention !

Wejdane BOUCHHIOUA