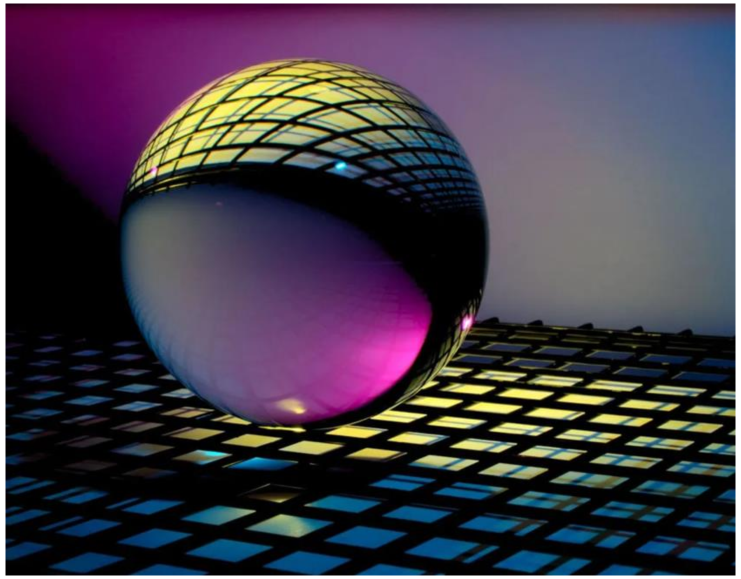
ITERATIVE CLOSEST POINT USING MPI

Presentation for CSE633: Parallel Computing [Spring 2025]

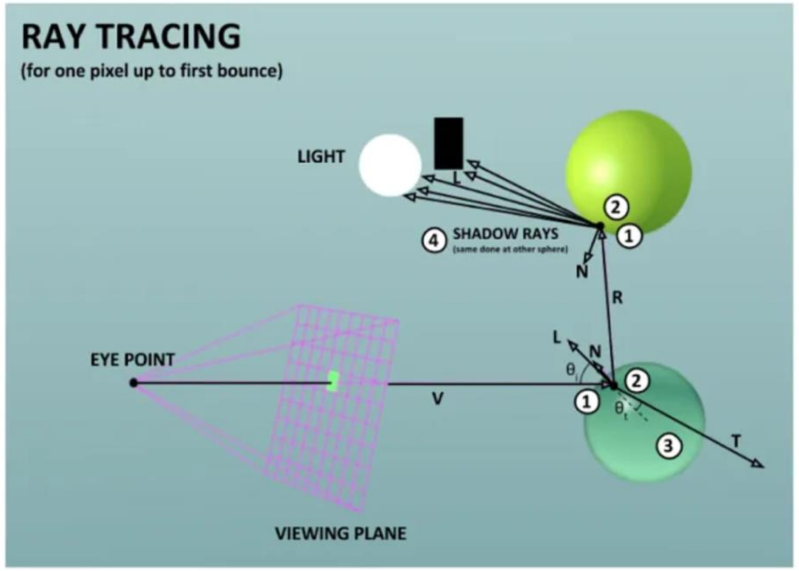
Utkarsh Kumar

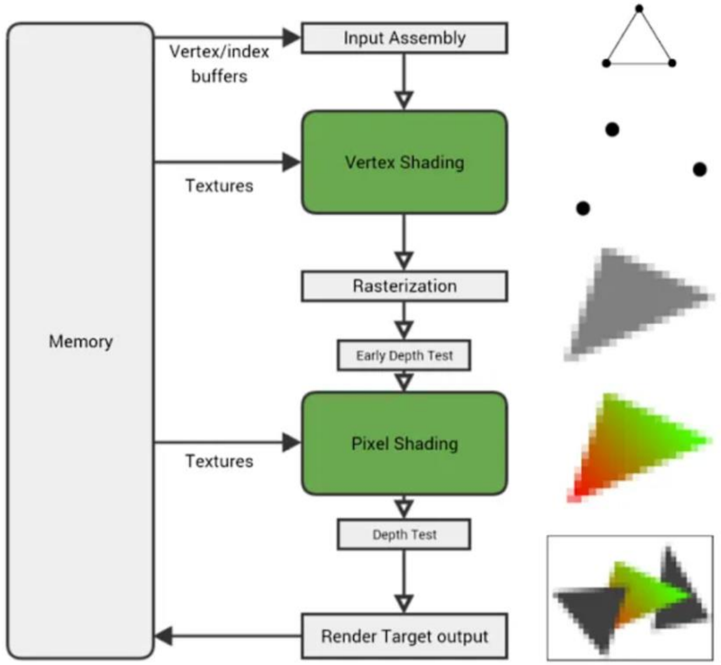
University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

What is ~~neural~~ classical rendering?



A typical spherical rendered picture is the hello world of computer graphics.





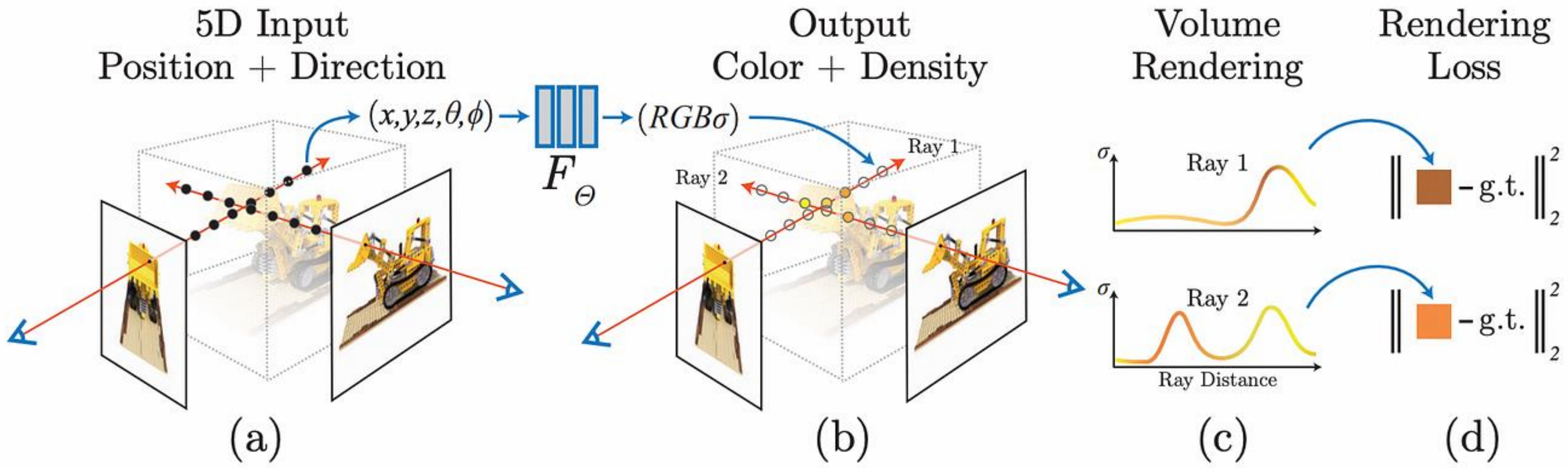
GPU Performance for Game Artists (80.lv/articles/gpu-performance-for-game-artists)
Ray tracing (graphics) (en.wikipedia.org/wiki/Ray_tracing_(graphics))

Why abandon classical rendering?

- Explicitly rendering millions of points
- Even more ray tracing for light paths
- It is not easy to determine redundancies and compress
- Most importantly – how to realistically imitate life?
    - How to use real life alignments for virtual/compute tasks
    - What does it mean to understand a scene?

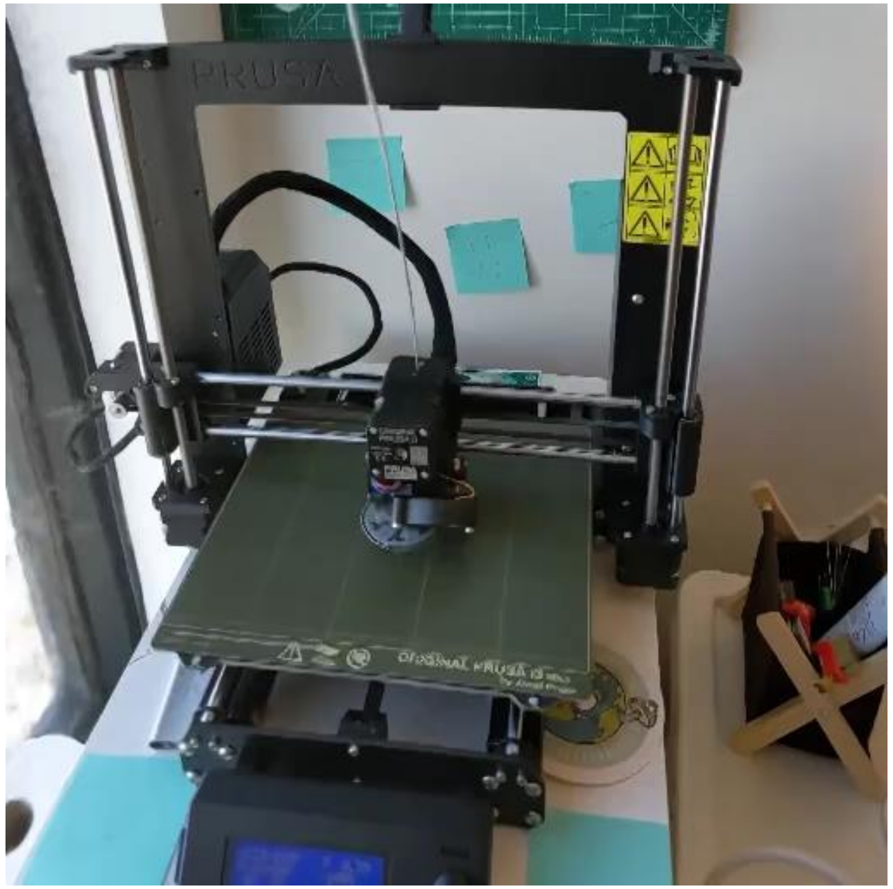
University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Enter neural rendering pipelines – NeRFs (Neural Radiance Fields)



Demo Static

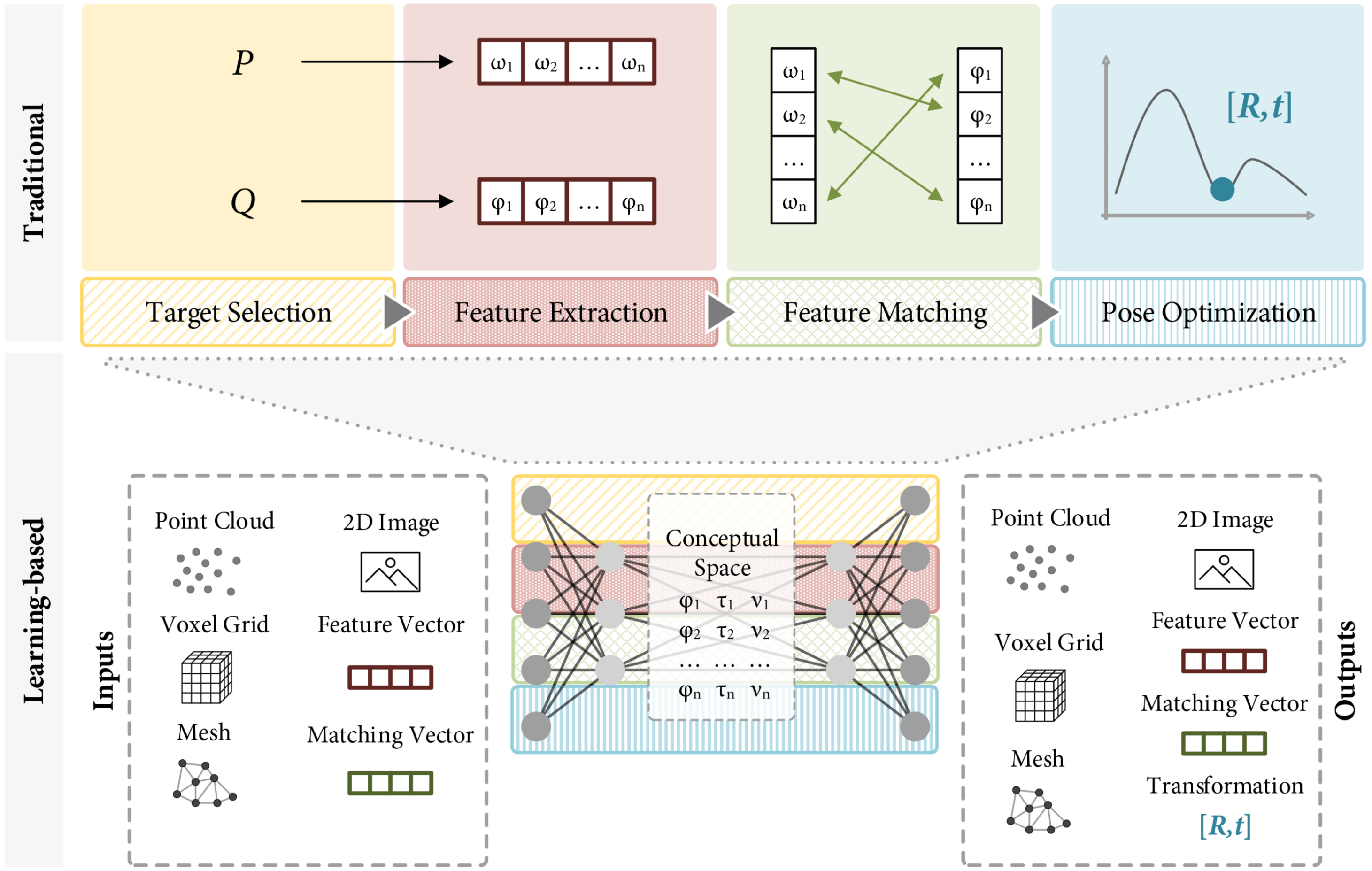
How good are the neural pipelines for dynamic stuffs?



Algorithm from https://www.albertpumarola.com/research/D-NeRF/index.html, re-emulated by Utkarsh

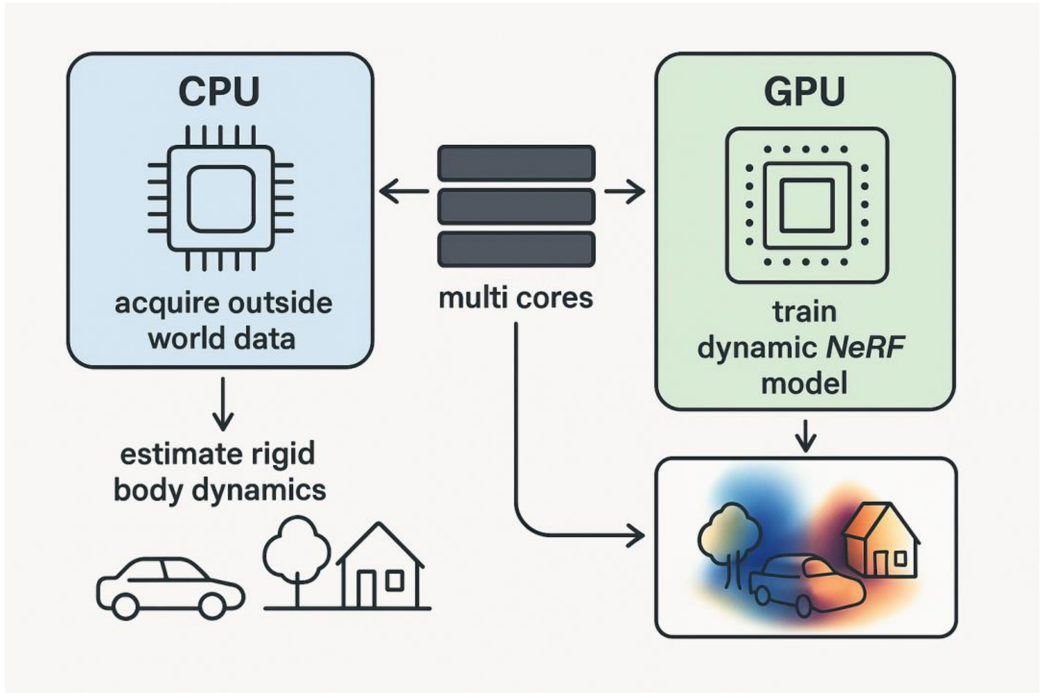
The alignment problem

- Does a neural network truly learn dynamics and motion?
  - Or does it only memorize the frames?
- The alignment problem – how to make neural networks truly understand ”motion”
- How to make it fast?
  - Most motion data is online, from a moving frame of multiple moving objects
- How do I keep acquisitions and CPU cores busy while GPU executes neural rendering pipeline



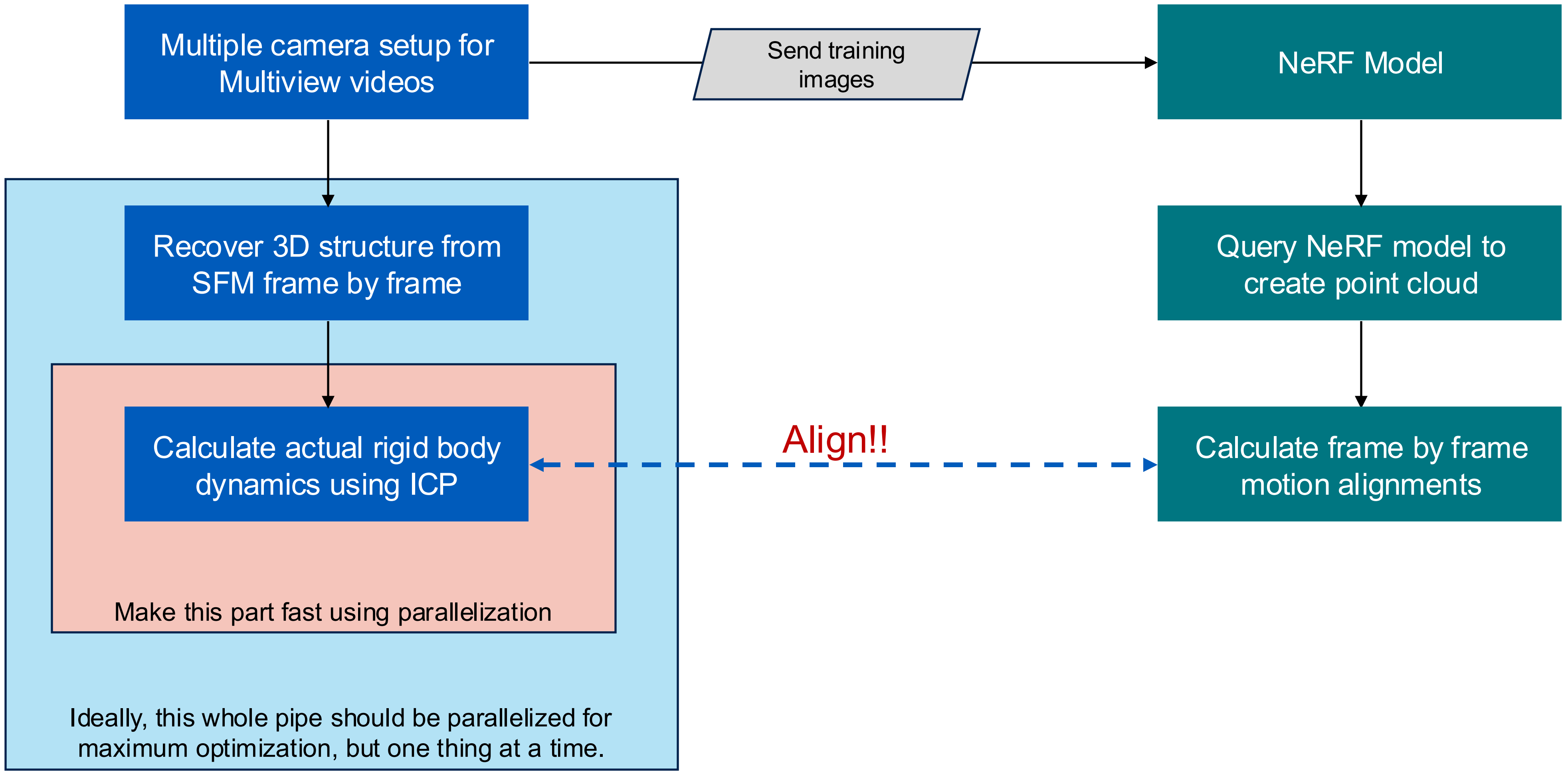
The alignment problem – reinforce motion during training

- Add actual recovered motion loss to optimization objective
- Fast rigid body motion estimations on the fly to calculate losses on

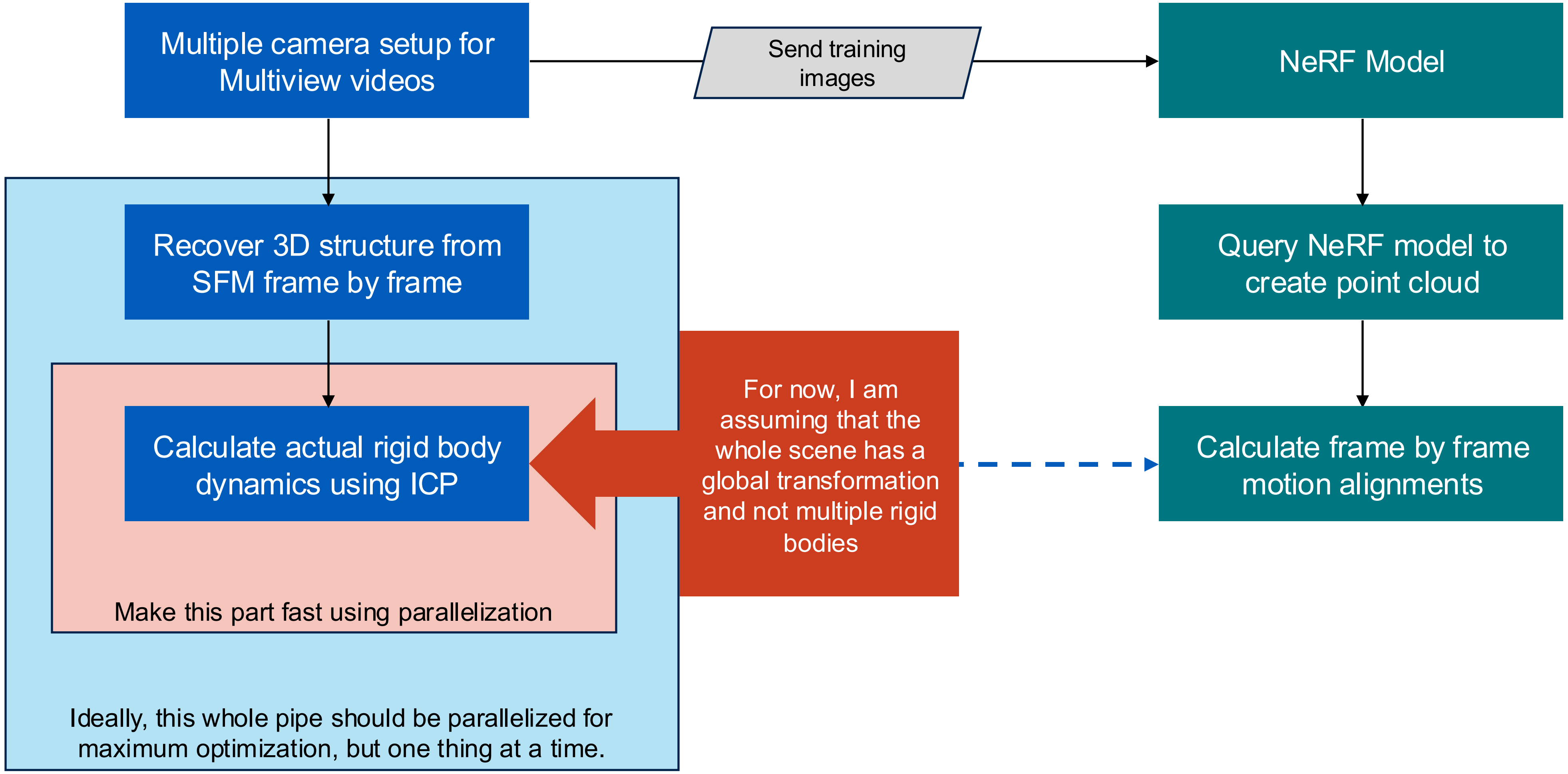

CPU
acquire outside world data
multi cores
GPU
train dynamic *NeRF* model
estimate rigid body dynamics

Shamelessly created using diffusion image models, but you get the idea

A high-level overview


Multiple camera setup for Multiview videos
Send training images
NeRF Model
Recover 3D structure from SFM frame by frame
Query NeRF model to create point cloud
Calculate actual rigid body dynamics using ICP
Align!!
Calculate frame by frame motion alignments
Make this part fast using parallelization
Ideally, this whole pipe should be parallelized for maximum optimization, but one thing at a time.

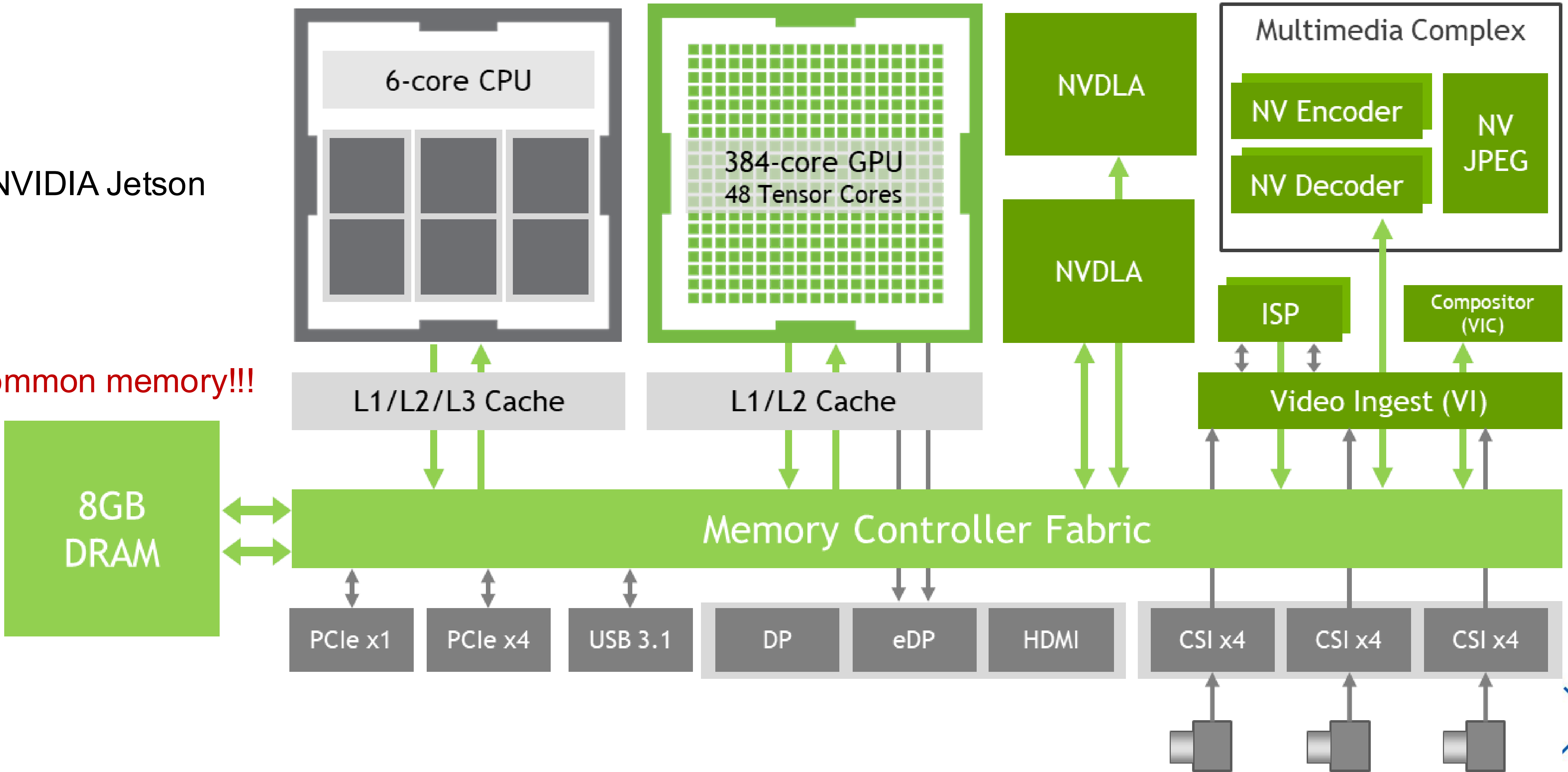
A high-level overview – make the problem simpler for now:


Multiple camera setup for Multiview videos
Send training images
NeRF Model
Recover 3D structure from SFM frame by frame
Query NeRF model to create point cloud
Calculate actual rigid body dynamics using ICP
For now, I am assuming that the whole scene has a global transformation and not multiple rigid bodies
Calculate frame by frame motion alignments
Make this part fast using parallelization
Ideally, this whole pipe should be parallelized for maximum optimization, but one thing at a time.

This kind of a setup is very suitable for modern edge devices

An NVIDIA Jetson

Common memory!!!



Iterative Closest Point (ICP) Algorithm

```
Require: Fixed point cloud $P$, moving point cloud $Q$, convergence threshold $\epsilon$, maximum iterations $N$
Ensure: Transformation $(R, t)$ aligning $Q$ to $P$
 1: $Q_0 \leftarrow Q$, $i \leftarrow 0$
 2: repeat                                          ▷ Find correspondences between $Q_i$ and $P$
 3:     for each point $q_j \in Q_i$, $j = 1, \dots, n$ do
 4:         $p_j \leftarrow \arg\min_{p \in P} \|p - q_j\|$
 5:     end for
 6:                                                 ▷ Compute centroids of the corresponding points
 7:     $\bar{p} \leftarrow \frac{1}{n} \sum_{j=1}^{n} p_j$
 8:     $\bar{q} \leftarrow \frac{1}{n} \sum_{j=1}^{n} q_j$
 9:                                                 ▷ Compute the cross-covariance matrix
10:     $H \leftarrow \sum_{j=1}^{n} (q_j - \bar{q})(p_j - \bar{p})^T$
11:                                                 ▷ Compute the SVD of $H$: $H = U \Sigma V^T$
12:     Compute $U$, $\Sigma$, and $V$ such that $H = U \Sigma V^T$
13:     $R \leftarrow V U^T$
14:     if $\det(R) < 0$ then
15:         Adjust $V$ by negating its last column: $V(:, n) \leftarrow -V(:, n)$
16:         $R \leftarrow V U^T$
17:     end if
18:                                                 ▷ Compute the translation
19:     $t \leftarrow \bar{p} - R \bar{q}$
20:                                                 ▷ Update the moving point cloud
21:     $Q_{i+1} \leftarrow \{R q + t \mid q \in Q_i\}$
22:     $i \leftarrow i + 1$
23: until Mean error $\frac{1}{n} \sum_{j=1}^{n} \|p_j - (R q_j + t)\| < \epsilon$ or $i \geq N$
24: return $(R, t)$
```

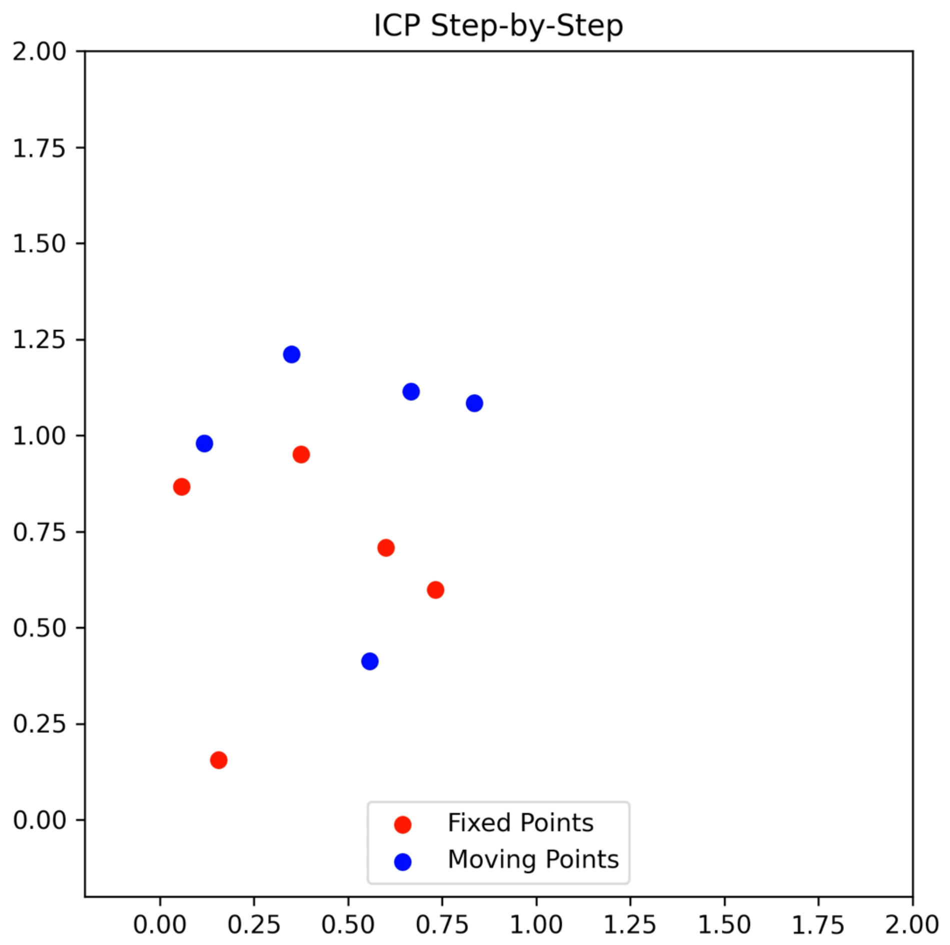
Iterative Closest Point (ICP) Algorithm

```
Require: Fixed point cloud $P$, moving point cloud $Q$, convergence threshold $\epsilon$, maximum iterations $N$
Ensure: Transformation $(R, t)$ aligning $Q$ to $P$
 1: $Q_0 \leftarrow Q$, $i \leftarrow 0$
 2: repeat                                                    ▷ Find correspondences between $Q_i$ and $P$
 3:     for each point $q_j \in Q_i$, $j = 1, \ldots, n$ do
 4:         $p_j \leftarrow \arg\min_{p \in P} \|p - q_j\|$
 5:     end for
 6:                                                           ▷ Compute centroids of the corresponding points
 7:     $\bar{p} \leftarrow \frac{1}{n} \sum_{j=1}^{n} p_j$
 8:     $\bar{q} \leftarrow \frac{1}{n} \sum_{j=1}^{n} q_j$
 9:                                                           ▷ Compute the cross-covariance matrix
10:     $H \leftarrow \sum_{j=1}^{n} (q_j - \bar{q})(p_j - \bar{p})^T$
11:                                                           ▷ Compute the SVD of $H$: $H = U \Sigma V^T$
12:     Compute $U$, $\Sigma$, and $V$ such that $H = U \Sigma V^T$
13:     $R \leftarrow V U^T$
14:     if $\det(R) < 0$ then
15:         Adjust $V$ by negating its last column: $V(:, n) \leftarrow -V(:, n)$
16:         $R \leftarrow V U^T$
17:     end if
18:                                                           ▷ Compute the translation
19:     $t \leftarrow \bar{p} - R \bar{q}$
20:                                                           ▷ Update the moving point cloud
21:     $Q_{i+1} \leftarrow \{R q + t \mid q \in Q_i\}$
22:     $i \leftarrow i + 1$
23: until Mean error $\frac{1}{n} \sum_{j=1}^{n} \|p_j - (R q_j + t)\| < \epsilon$ or $i \geq N$
24: return $(R, t)$
```

What is that one fundamental problem?

Ans – We don’t know the point correspondences!
If we knew this was a piece of cake. But we don’t know where the new points are, where did they move to?

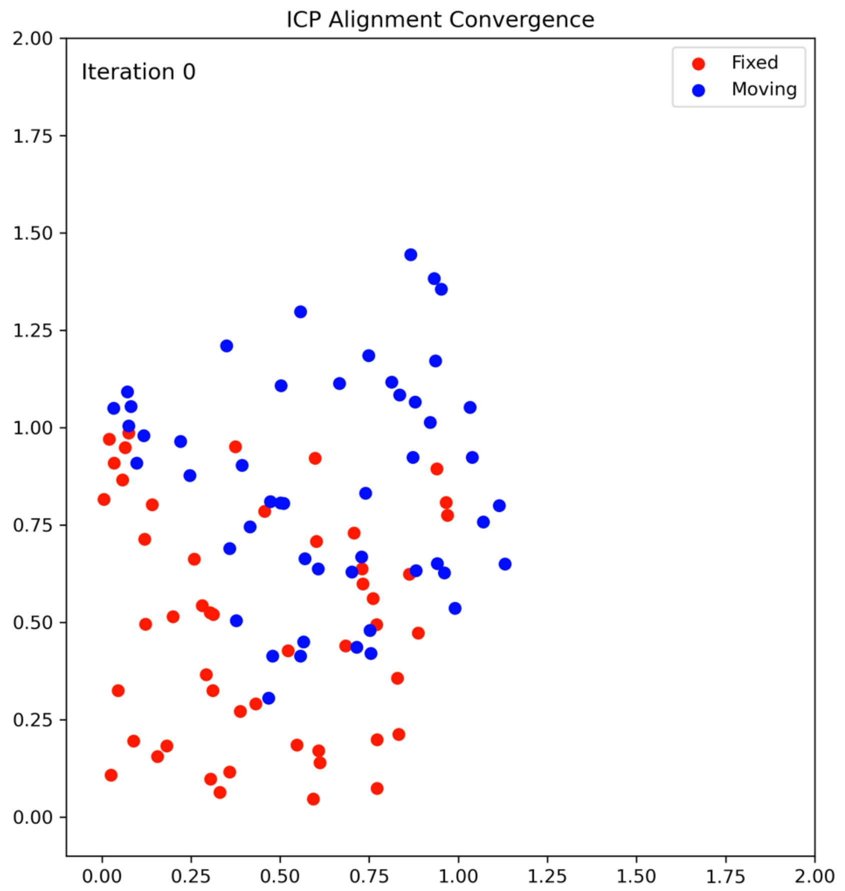
Iterative Closest Point (ICP) Algorithm – one step

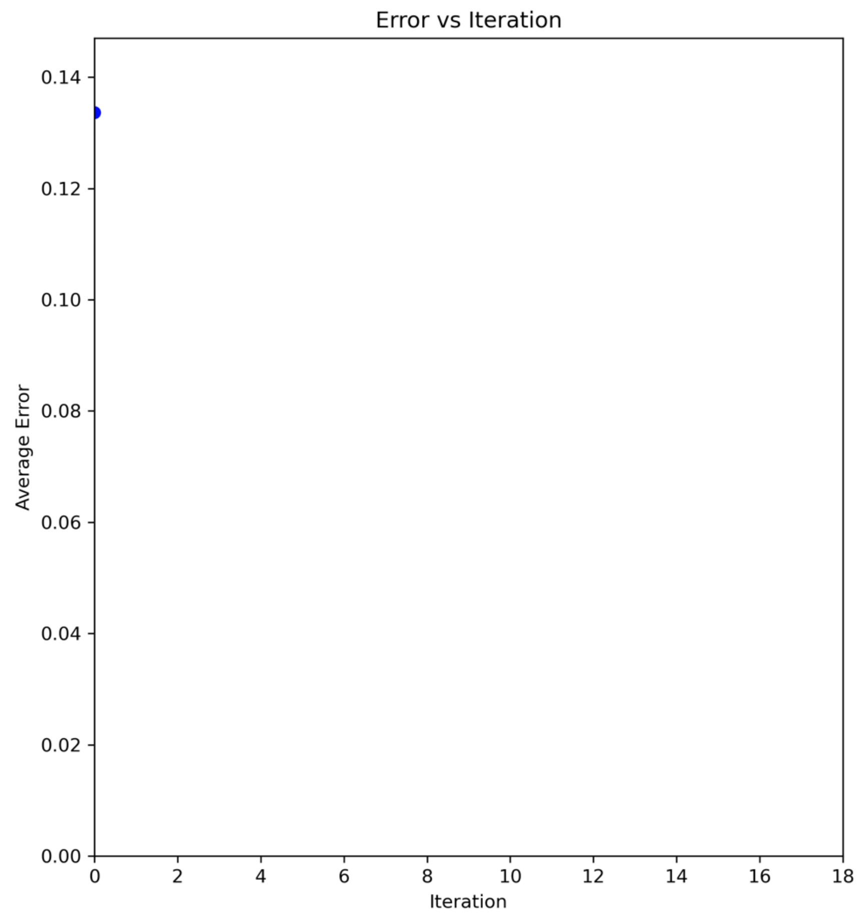


ICP Pseudocode

```
1. For each point $q \in Q$, find $p = \arg\min_{p \in P} \|p - q\|$.
2. Compute centroids: $\bar{q} = \frac{1}{n}\sum q$, $\bar{p} = \frac{1}{n}\sum p$.
3. Compute cross-covariance: $H = \sum (q - \bar{q})(p - \bar{p})^T$.
4. Compute SVD: $H = U \Sigma V^T$.
5. Compute rotation: $R = V U^T$ (adjust if $\det(R) < 0$).
6. Compute translation: $t = \bar{p} - R\bar{q}$.
7. Update moving cloud: $Q \leftarrow RQ + t$.
```

Iterative Closest Point (ICP) Algorithm – one frame solution





Vanilla serial implementation

Step 1 – Calculate the nearest neighbors for each point on the moving cloud

```
for (int i = 0; i < n; i++) {
        double min_dist = 1e9;
        int min_idx = 0;
        for (int j = 0; j < n; j++) {
            double d = distance(moving[i], fixed[j]);
            if (d < min_dist) {
                min_dist = d;
                min_idx = j;
            }
        }
        correspondence[i] = min_idx;
    }
```

Vanilla serial implementation

Step 2 – Compute centroids for the two clouds

```
double sum_mx = 0, sum_my = 0;
   double sum_px = 0, sum_py = 0;
   for (int i = 0; i < n; i++) {
       sum_mx += moving[i].x;
       sum_my += moving[i].y;
       int idx = correspondence[i];
       sum_px += fixed[idx].x;
       sum_py += fixed[idx].y;
   }
   double centroid_mx = sum_mx / n;
   double centroid_my = sum_my / n;
   double centroid_px = sum_px / n;
   double centroid_py = sum_py / n;
```

Vanilla serial implementation

Step 3 – Compute cross covariance terms

```
double Sxx = 0, Sxy = 0;
   for (int i = 0; i < n; i++) {
       double qx = moving[i].x - centroid_mx;
       double qy = moving[i].y - centroid_my;
       int idx = correspondence[i];
       double px = fixed[idx].x - centroid_px;
       double py = fixed[idx].y - centroid_py;
       Sxx += qx * px + qy * py;
       Sxy += qx * py - qy * px;
```

Vanilla serial implementation

Step 4 – Compute optimal rotation angle

```
double theta = atan2(Sxy, Sxx);
double cos_theta = cos(theta);
double sin_theta = sin(theta);
```

Step 5 – Compute centroid translation

```
 double tx = centroid_px - (cos_theta *
centroid_mx - sin_theta * centroid_my);
    double ty = centroid_py - (sin_theta *
centroid_mx + cos_theta * centroid_my);
```

Step 6 – Update all points

```
for (int i = 0; i < n; i++) {
       double x = moving[i].x;
       double y = moving[i].y;
       moving[i].x = cos_theta * x - sin_theta * y + tx;
       moving[i].y = sin_theta * x + cos_theta * y + ty;
```

Vanilla serial implementation

Step 4 – Compute optimal rotation angle

```
double theta = atan2(Sxy, Sxx);
double cos_theta = cos(theta);
double sin_theta = sin(theta);
```

Step 5 – Compute centroid translation

```
 double tx = centroid_px - (cos_theta *
centroid_mx - sin_theta * centroid_my);
     double ty = centroid_py - (sin_theta *
centroid_mx + cos_theta * centroid_my);
```

Step 6 – Update all points

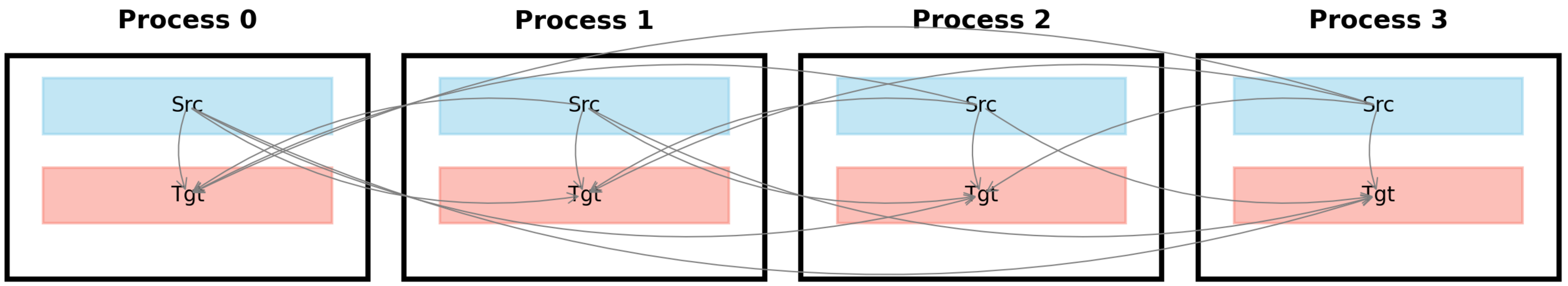
```
for (int i = 0; i < n; i++) {
       double x = moving[i].x;
       double y = moving[i].y;
       moving[i].x = cos_theta * x - sin_theta * y + tx;
       moving[i].y = sin_theta * x + cos_theta * y + ty;
```

Parallelization Scheme:

To implement:

- Give every processor an equal share of both target and source mesh, randomly distributed
  - Calculate local optimizations
  - Broadcast centroid and covariances to all, receive from all
  - After all send & receive, update local values
  - Update all points locally

$\left.\right\}$ Iterate until convergence



MPI_Scatter: Partition Source & Target
MPI_Bcast: Exchange Target Partitions

Parallelization scheme

- Each process updates its local source points using the current transformation and computes its initial nearest neighbor matches (local optimals)
- One-by-one, each process broadcasts its local target partition (due to memory constraints), so all processes can search that partition and update their best candidates for every local source point
- After all broadcasts, every process has effectively determined the global nearest neighbor for each of its local source points
- All processes then collectively reduce their local accumulations (sums and covariance matrices) via MPI_Allreduce, ensuring that every processor has the global optimal parameters
- Finally, each process independently computes the updated transformation and applies it to its local source points for the next iteration

```
for $iter = 1$ to $maxIter$ do
    for each local source point $s_i \in S_p$ do
        Transform: $s_i \leftarrow R\, s_i + t$;
    end
    for each process $q = 0, \ldots, P-1$ do
        Process $q$ broadcasts its local target partition $T_q$ via MPI_Bcast;
        for each local source point $s_i \in S_p$ do
            Search in received $T_q$ for nearest neighbor candidate $t_q^*$;
            Update local best candidate for $s_i$ if
                $\|s_i - t_q^*\|^2 < \text{current best distance}$
        end
    end
    for each local source point $s_i \in S_p$ do
        Accumulate local sums:
        $S_{\text{sum}}^{(p)} \mathrel{+}= s_i, \quad T_{\text{sum}}^{(p)} \mathrel{+}= \text{best\_neighbor}(s_i)$, and
        Covariance: $H_p \mathrel{+}= s_i\, \text{best\_neighbor}(s_i)^T$;
    end
    // Distributed reduction: All processes share their results
    Use MPI_Allreduce to compute global sums:
        $\mu_S = \frac{1}{N} \sum_{p=0}^{P-1} S_{\text{sum}}^{(p)}, \quad \mu_T = \frac{1}{N} \sum_{p=0}^{P-1} T_{\text{sum}}^{(p)}, \quad H = \sum_{p=0}^{P-1} H_p$
    Adjust $H \leftarrow H - N\, \mu_S\, \mu_T^T$;
    Each process computes optimal $(R, t)$ from $H$ (via SVD/Horn's method) locally;
    if convergence criteria met then
        break
    end
end
```

Cost analysis

Assume that the serial ICP computation requires

$$T_{\text{serial}} = \alpha\, N\, M,$$

where N is the number of source points, M is the number of target points, and α is the cost per distance computation.

In the fully distributed scheme with P processes, each process holds

$$N_{\text{local}} = \frac{N}{P} \quad \text{and} \quad M_{\text{local}} = \frac{M}{P}.$$

Each process performs a nearest-neighbor search over all target partitions by broadcasting one partition at a time. The computation cost per process is

$$T_{\text{comp}} = \alpha\, N_{\text{local}}\, M = \frac{\alpha\, N\, M}{P}.$$

Each broadcast of a target partition incurs a cost β; since there are P broadcasts per iteration, the communication cost is

$$T_{\text{comm}} = P\, \beta.$$

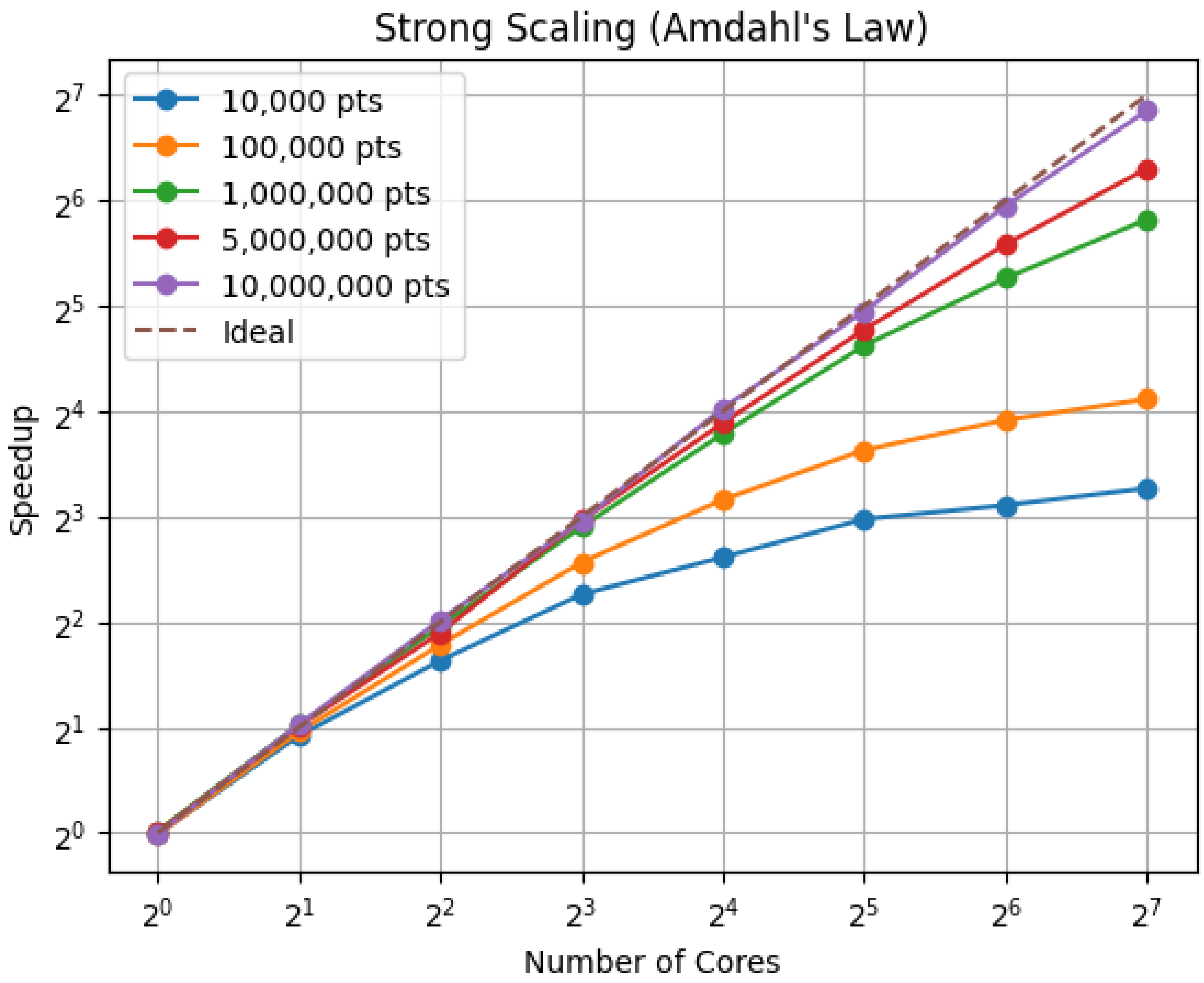
Thus, the total parallel time per iteration is approximately

$$T_{\text{parallel}} = \frac{\alpha\, N\, M}{P} + P\, \beta.$$

The speedup $S(P)$ is given by

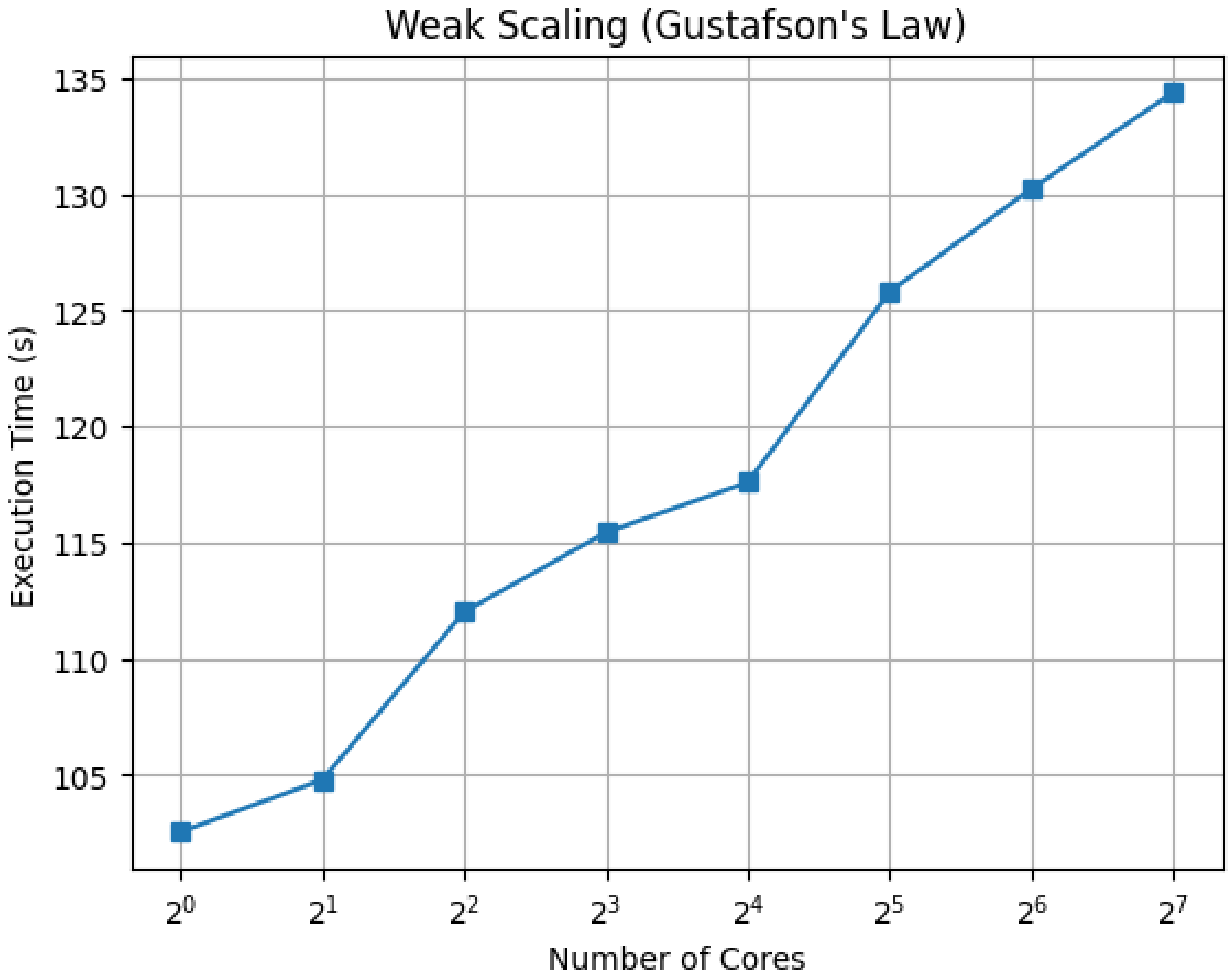
$$S(P) = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{\alpha\, N\, M}{\frac{\alpha\, N\, M}{P} + \beta\, P} = \frac{P}{1 + \frac{\beta\, P^2}{\alpha\, N\, M}}.$$

Strong Scaling



Weak Scaling

Please note: Time (Y) axis scale is in 10^3 seconds



Optimization: Non-Blocking Communications

- At the beginning of every iteration, the most time-consuming step is to build a k-d tree for a nearest neighbor correspondence
- When updating values over communication, we can use the idle cores to start building k-d tree for the next step

1. Local Optimals

```
corr = find_correspondences(source, tgt_tree, local_pts);

compute src_sum[], tgt_sum[], cov[][] from corr
```

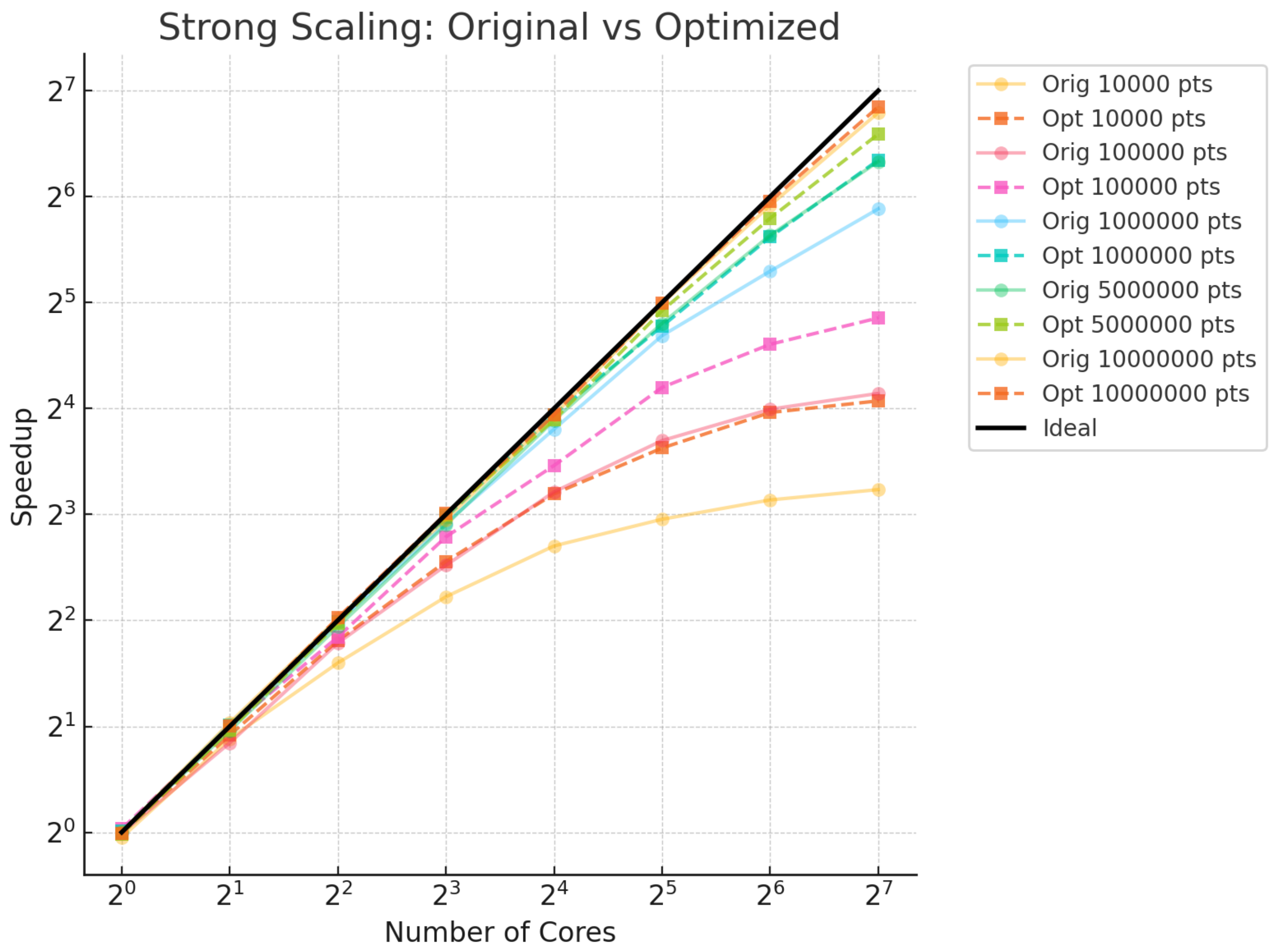
2. Non-Blocking Broadcast

```
MPI_Iallreduce(src_sum, src_sum, 3, …, &reqs[0]);
MPI_Iallreduce(tgt_sum, tgt_sum, 3, …, &reqs[1]);
for (r = 0; r < 3; ++r)
  MPI_Iallreduce(cov[r], cov[r], 3, …, &reqs[2+r]);

free_kdtree(src_tree);
src_tree = build_kdtree(source, local_pts);
```

Build for next iteration

Strong Scaling: Non-Blocking



Conclusions:

- Non-blocking is a very trivial optimization, case needs to be studied in more depth
- Write fused kernels: Make sure cores are busy while network transfers huge data chunks in parallel
- Mesh is being loaded in continuous chunks, but randomly – load mesh in a more ordered way
- Hierarchical implementation – but does it satisfy no Monte Carlo?
- Read more papers, study more parallel algorithms!

Thank You!