15-418 Project Final Report

Matthew Yu, Emily Ho

December 2025

1 Summary

We implemented a 3D reconstruction pipeline that takes in multiple 2D images and outputs a 3D representation of the image. We implemented a serial CPU version with Python, and then parallelized it on the GPU using CUDA. Overall, we were able to achieve **97.42x** speedup on the GPU version in the correspondence + triangulation pipeline.

2 Background

2.1 3D Reconstruction Pipeline

Reconstructing a 3D representation of 2D images can often be split up into multiple steps:

1. **Preprocessing: Feature extraction and matching**: The first stage of the pipeline includes running the SIFT algorithm to extract each individual pixel's important features for both the images, and then running an approximate KNN algorithm to create candidate matches between points from image 1 to image 2.

2. **Get correspondence via RANSAC**: The matches computed from the previous step are only candidates for actual correspondences between the two images, and thus a second step is needed to compute the actual point pairs (called inliers) that map from image 1 to image 2. We use RANSAC to run the eight point algorithm, which randomly samples any 8 point pairs and use them to find a fundamental matrix (F). Then, for each iteration, we compute the epipolar distance between the two point pairs using F and counts them as "inliers" only if it falls between a threshold. At the end of this step, the fundamental matrix yielding the most inliers is considered to be the one that best represents our images, and is returned as output.

3. **Triangulation to construct the 3D points**: With the filtered inlier correspondences and the relative camera poses (derived from the Fundamental matrix), we back-project rays from each camera center through the matched 2D pixels into 3D space. Since sensor noise prevents these rays from intersecting perfectly, we estimate the final 3D coordinate by solving a linear system to find the point that minimizes the distance to both rays.

For this project, we focused on parallelizing the second and third steps as they dominate the computational cost when scaling to large datasets. Although feature extraction occurs only once

per image, pairwise geometric verification scales quadratically ($O(N^2)$), creating a massive computational bottleneck that benefits significantly from the massive parallelism of CUDA for iterative RANSAC solvers and independent triangulation tasks. Additionally, since SIFT occurs only once per image, this could easily be preprocessed and precomputed in a pipeline. We wanted to focus on the parts that are more complicated to scale as the number of images increases.

2.2 Inputs, Outputs, and Data Structures

The primary data structures driving the simulation are dense arrays representing image coordinates and matrices.

- **Inputs:** Two arrays of size N, `pts1` and `pts2`, where each element is a tuple (u, v) representing a feature coordinate. In our C++ implementation, these are stored as arrays of `float2` (1D) structures to ensure memory alignment.
- **Intermediates:**
  - **Fundamental Matrix (F):** A 3×3 float matrix. During RANSAC, thousands of candidate matrices are generated.
  - **Inlier Mask:** An array of size N (booleans or ints) indicating whether the i-th point pair is an inlier that falls within the threshold given a specific F.
- **Outputs:** A final optimized 3×3 Fundamental Matrix and an array of size M (where $M \leq N$) containing `float3` vectors representing the reconstructed 3D points.

2.3 Workload Breakdown

The pipeline contains two distinct computational profiles that require different parallelization strategies:

2.3.1 RANSAC (Throughput-Bound / Compute-Bound)

RANSAC is an iterative algorithm used to find the best model in the presence of noise. The complexity is $O(K \cdot N)$, where K is the number of iterations and N is the number of matched points.

- **Key Operations for Each Iteration:**
  1. *Model Generation:* Solving a linear system ($Af = 0$) via Singular Value Decomposition (SVD) on a small 8×9 matrix.
  2. *Model Evaluation:* Computing the matrix-vector product $x'^T F x$ for all N points to check error thresholds.
- **Parallelism Potential:** The workload is highly parallel. Each iteration k is independent of iteration $k + 1$ (task parallelism). Furthermore, within the evaluation step, the error check for each point n is independent (data parallelism).
- **Dependencies:** The only dependency is a reduction at the end of execution to select the iteration that produced the maximum number of inliers.

2.3.2 Triangulation (Latency-Bound / Memory-Bound)

Once the fundamental matrix is found, we solve for the 3D coordinates.

- **Key Operations:** For every inlier pair, we construct a 4×4 linear system and solve for the null space using SVD.
- **Parallelism Potential:** Each of the M points can be triangulated completely independently (data parallelism).
- **Dependencies:** For each of the points, triangulation is called 4 times within findM2 to find the best reconstruction possible from the pair of points.

2.3.3 Locality

In the RANSAC model evaluation phase, every thread within a warp (iteration) needs to read the entire set of input points (`pts1`, `pts2`). The 8 points chosen for the algorithm also gets accessed very often. This exhibits high temporal locality but presents a challenge for bandwidth if N is large.

In Triangulation, the access pattern is purely streaming (read point i, perform operations, write point i), offering poor temporal locality but perfect spatial locality.

2.3.4 SIMD Execution

1. **Highly Amendable**: Compute epipolar distance / inliers in RANSAC, Triangulation
   (a) Both of these steps utilize data parallelism and each thread is working on one point pair. Thus, it is highly amendable to SIMD execution and have high convergence in the warps.
2. **SIMD Divergence**: Sampling 8 points, Eight point algorithm, SVDs
   (a) All of these steps are done by 1 thread for the most part, which leads to divergence within the warp / block.

3 Approach

We decided to use CUDA to parallelize RANSAC and triangulation on the GPU within the GHC machines, since there are a lot of data parallelism that the pipeline could benefit from. Additionally, since RANSAC is a throughput bound problem, GPUs allow us to test many hypothesis at once, which not only increases the speed of the program but also the quality of the final output. For instance, on the CPU, the execution time scales up with the number of RANSAC iterations and number of points, whereas on the GPU it scales much slower.

We decided to use CUDA and C++ to create the parallel modules (source code under /code/src/). For the input/output handling of image files along with preprocessing prior to feeding into steps 2 and 3, we used Python. Due to the disk quota on the GHC machines, in order to visualize our results, we have a colab that takes in files of type .npz (compressed file format by numpy), which we generate as an optional output file within our 3D reconstruction pipeline on the GHC

machines.

The serial pipeline is adopted from CMU's Computer Vision Course assignment on reconstruction.

3.1 RANSAC

The RANSAC algorithm can be represented as follows:

```
# Pts1 and pts2 are 2D numpy arrays containing all the candidate correspondence points between the two images.
def ransac_algorithm(pts1, pts2, threshold, iters):
    best_F = [[0,0,0], [0,0,0], [0,0,0]]
    max_inliers = -1
    M = max(img_width, img_height)

    for _ in range(iters):
        sample_pts1, sample_pts2 = sample_8_points(pts1, pts2)

        curr_F = eightpoint_algorithm(sample_pts1, sample_pts2, M)

        errors = compute_symmetric_epipolar_distance(curr_F, sample_pts1, sample_pts2)

        num_inliers = count_inliers(errors, threshold)

        if num_inliers > max_inliers:
            max_inliers = num_inliers
            best_F = curr_F
```

For each iteration of the RANSAC algorithm, the only place that requires synchronization with global state is updating the max inlier and storing the best fundamental matrix. Thus, we recognized very early that we could parallelize each iteration of the RANSAC algorithm and do a reduction at the end to get the best fundamental matrix. Thus, the first version we wrote was one that was parallelized by the iteration, where each block would be responsible for 1 iteration.

3.1.1 Sample 8 Points

For generating random numbers, we used the cuRAND library, which requires us to initialize a cuRAND state before using it. Since each block runs 1 iteration and all are happening in parallel, we initialized a cuRAND state for each block to ensure each block have independent random states and can generate different numbers. Before the start of the actual RANSAC algorithm, we launched a separate kernel that initializes all these random states in parallel. One optimization we could make in the future as we scale up to support multiple images and run RANSAC algorithm in parallel for different pairs of images is that we only need to initialize the states once and it can be reused for each block, rather than needing to reinitialize at the start of each RANSAC run.

After the random states are initialized, we then enter the main kernel for the RANSAC algorithm. The first part of each iteration is to randomly sample 8 points. This step should be done once for each iteration (block), and synchronization is needed before moving forward with the algorithm. Since the 8 points cannot be repeated, and in CUDA only 1 random index could be generated at once, we decided to have 1 thread randomly generate 8 non repeating indexes and fetch

the coordinates at those indexes into shared memory. Once that is done, we synchronize across all threads in the block.

```
def sample_eight_points(all_pts1, all_pts2, s_pts1, s_pts2): # device function
    iter_num = block_id
    local_rand_state = global_states[iter_num] # global_states is populated after
    the kernel for initializing random states

    indices = []
    for i in range(8):
        while true:
            idx = generate_randint(0, len(pts1))

            # Check uniquness
            if idx not in indices:
                indices.append(idx)
                break

        # Fetch the points coordinates at the index into shared mem.
        s_pts1[i*2] = all_pts1[indices[i*2]]
        s_pts1[i*2+1] = all_pts1[indices[i*2+1]]
        s_pts2[i*2] = all_pts2[indices[i*2]]
        s_pts2[i*2+1] = all_pts2[indices[i*2+1]]
```

3.1.2 Eightpoint Algorithm

Now that each block (iteration) has chosen the 8 points, we run the eight point algorithm to find the fundamental matrix. The fundamental matrix F encapsulates the epipolar geometry between two views. It maps a point in one image to a line (the epipolar line) in the other image. If a point x in the first image matches a point x' in the second, x' is constrained to lie on the line defined by Fx.

The algorithm consists of multiple steps:

1. **Normalize points**: Raw pixel values are often large, which can lead to instability as we square or multiply the values. Thus, we normalize by the dimension of the image to prevent overflow.

2. **Build the linear system**: The relationship $x'^T Fx = 0$ can be rewritten as a linear equation $Af = 0$. The code constructs this matrix A, where each row corresponds to one pair of matched points. It stacks the products of the coordinates to create the coefficients for the unknown values of F. The result is an 8 * 9 matrix, where each row represents 1 specific point pair, and the columns are for each values in the 3 * 3 fundamental matrix we are trying to solve.

3. **Solve for F (Least Squares)**: Since real data is noisy, we can't solve $Af = 0$ perfectly. Instead, we minimize the error $||Af||$ using SVD to find the vector f corresponding to the smallest singular value (the last row of Vh). This is the "best fit" solution and thus is the coordinates for the fundamental matrix.

4. **Enforce rank 2**: The math in Step 3 produces a generic 3×3 matrix that likely has Rank 3 (determinant $\neq 0$). However, a valid Fundamental Matrix must have Rank 2 (determinant $= 0$) to geometrically represent lines intersecting at an epipole. This step forces that property by doing another SVD on F and setting the smallest singular value to 0.

5. **Denormalization & Scaling**: This is to undo the effect of step 1 so that the fundamental matrix correctly represents the correspondence at the actual images' scale.

Before we can proceed onto the next step of the algorithm, the fundamental matrix must be computed for the iteration and synchronized across all the threads. Unfortunately, a lot of this is simple math that does not benefit from parallelization too much. For instance, the SVD for this step are for 3 * 3 and 8 * 9 matrices, which we concluded is faster if done in serial by one thread (Read the section on triangulation for more details). Thus, this part of the kernel function has a lot of serial portions, but it does not affect performance as a lot of these are simple calculations.

One place we did parallelize is a combination of steps 1 and 2 (done in 1 step). To compute each row of the A matrix (containing 9 elements), 4 memory loads are needed (x y coordinate for pts1, and for pts2). Thus, we did cooperative fetching and have the first 8 threads of each block load their corresponding point pairs, and compute and store the values into shared memory. We then synchronize.

For the cooperative loading, instead of assigning it to 72 threads where each thread is involved in 1 of the elements of the A matrix, we have 8 threads where each thread is responsible for a whole row. This is due to the fact that every single row requires the same 2 coordinates, and every single column actually has a different computation to do. If we were to do 72 threads, there'll be 9 threads reading the same exact data, but they'll all be computing different things. This leads to a high divergence where we'll end up with a serial execution for the 9 elements since we'll need many if statements to see which computation the thread should actually compute. Thus, there is no benefit in having more threads populate the A array.

The rest of the steps are done by thread 0 due to the lack of parallelism available (very small matrices), which is ok as Step 3 is where most of the computation happens and at a larger scale.

The pseudocode for this step is as follows:

```
def eightpoint(pts1, pts2, M): # pts1 and pts2 only contain the 8 points
    s_A = [] # assume 8 * 9 matrix in shared mem
    s_F = [] # assume 3 * 3 matrix in shared mem, for storing result fundamental
    matrix

    # Cooperatively construct matrix A (step 1 and 2).
    if (thread_id < 8):
        # Load the point from pts1 and pts2
        x1 = pts1[thread_id * 2] / M
        y1 = pts1[thread_id * 2 + 1] / M
        x2 = pts2[thread_id * 2] / M
        y2 = pts2[thread_id * 2 + 1] / M

        populate_row_for_A(s_A, thread_id, x1, y1, x2, y2)
    __syncthreads()

    # Rest are done in serial
    if thread_id == 0:
        V = svd(A)
        F = get_rank_two_matrix(V)
        F = unscale(F)
        s_F = F # store to shared memory
    __syncthreads()
```

3.1.3 Compute Epipolar Distance for Each Point

The goal of RANSAC is to explore many fundamental matrices and find the best fundamental matrix that maximizes the amount of inliers. Inliers are point pairs in which their symmetric epipolar distance is less than a given threshold. The last step is to compute the epipolar distances of each of the point pairs using the fundamental matrix, and determine the number of inliers for the iteration.

This is massively parallelizable as each point pair could be computed completely independent from other point pairs. Thus, we utilized data parallelism to do the math needed for each point pair to get their distance. Since each block consist of 256 threads and we might have more than 256 number of point pairs, each thread may be responsible for multiple point pairs. We ensured coalesced memory access by making sure the threads in each warp are accessing sequential memory from each other.

The array containing all points were converted directly from numpy (with shape N * 2), meaning that the arrays contain [pt1_0_x, pt1_0_y, pt1_1_x, pt1_1_y, ...]. Originally, we were loading the x values into registers, then ys. However, we realized this is actually not coalesced as each thread would be skipping 1 value in the middle (the other coordinate). To mitigate this, we casted the float array into float2 and do 1 load for both x and y. This ensures that each thread is reading the bytes for both x and y coordinates in 1 load, meaning the memory access across all threads are now sequential. This improved our speedup as the number of points got large.

In the CPU implementation, after computing all of the epipolar distances into an array, it uses numpy operations to get a mask of all the point pairs that has epipolra distances smaller than the threshold. Then it does a sum on the mask to get the number of inliers, and use it to see if this iteration is better than the current best iteration. For the GPU version, since the iterations are occurring simultaneously, computing and storing the mask would entail needing to do it for every iteration, which takes (N * num_iters) memory and is unfeasible for large number of points or large number of iterations. Thus, this step does not worry about the mask at all and rather just computes the number of inliers using the fundamental matrix.

For each thread, since it might compute multiple epipolar distances, they each keep a local count of the number of inliers. At the very end once all the threads are done, we perform a reduction to find the total number of inliers for this block (iteration), and store that into a global array.

The pseudocode for this step is as follows:

```
# device func
def compute_epipolar_distances(pts1, pts2, num_points): # pts1 and pts2 contain
ALL points
    s_inlier_cnt = 0 # In shared memory.

    local_inlier_count = 0
    for i in range(thread_id, num_points, block_dim, threshold):
        float2 pt1 = pts1[i * 2] # Read both x and y in 1 load as float2
        float2 pt2 = pts2[i * 2]
        epipolar_dist = get_epipolar_dist(pt1, pt2)
        if epipolar_dist < threshold:
            local_inlier_count += 1

```

```
13          reduceAdd(s_inlier_cnt, local_inlier_count)
```

3.1.4 End of Kernel: Find Best Iteration

Outside of the kernel function, after cudaDeviceSynchronize() to ensure all blocks are done computing and have stored their inlier count into the global array, we then scan through the array to find the index with the max number of inliers, and use the index to get the fundamental matrix to return as the end of the RANSAC algorithm.

To keep the one iteration per block approach, we cannot do this step until after all the blocks have finished and returned. In other words, we cannot fuse this with the rest of the kernel function. Two approaches we considered were iterating through the array of inlier counts serially and finding the argmax, or perform a parallel reduction operation and compute the mask too. We tried both and actually found that the parallel reduction is only optimal at high number of points (10k or above), due to the overhead of launching another kernel function and synchronizing. Thus, since the images typically have less than 500 corresponding points, it is not worth to launch a separate kernel function and perform synchronization, so we went with the serial approach.

3.1.5 Observations and Shortcomings

One shortcoming of this approach is that the GPU is underutilized during step 1 and 2, as thread 1 is doing the majority of the work. Even in the parallel parts of step 2, only 8 threads participate in the cooperative fetching. This leads to severe under utilization of our GPU resources during steps 1 and 2, which is not ideal. We then looked into ways to increase the utilization of the GPU, and found that there's a lot of warp primitives and intrinsics that we could utilize.

Unfortunately, there is not much we can do regarding parallelizing the serial portions of step 1 and 2 due to the nature of the task. However, by having each warp represent one iteration instead of 1 block, we are decreasing the amount of idle threads significantly and thus the overall performance improves. Each warp on the GPU consist of 32 threads, meaning that our thread utilization for step 1 and 2 would increase by 8 (Since our block size is 256).

3.2 Improvements and Optimizations: Final Implementation

Our final implementation uses the 1 warp per iteration approach instead of the original blocked approach, where all threads within a warp work towards calculating the number of inliers after the fundamental matrix is generated and stored into shared memory by thread 0. The general structure of the code is the same as the blocked version mentioned above, with the following changes to the design:

1. **Identification**
    (a) **Block Version**: threadId represents the identity of each thread/worker within the iteration. Iteration represented by blockId. Increment by blockDim when dividing work within the iteration.

(b) **Warp Version**: laneId within the warp represents the identity of each thread/worker within the iteration. Iteration represented by blockId * 8 + warp_id. Increment by warp size when dividing work within the iteration.

2. **Memory**

   (a) **Block Version**: Since shared memory is for the whole iteration, we used it to store the A matrix in step 2, the 8 points for eightpoint algorithm, and the inlier counts.

   (b) **Warp Version**: Now each block actually is responsible for 8 iterations. Thus, we'll need to store 8 As and 8 sets of 8 point pairs for the algorithm. The amount of shared memory needed increased by 8, and we use the warp ID to determine which portion of shared memory each warp should access.

3. **Synchronization**

   (a) **Block Version**: Synchronize across all threads within a block (__syncthreads()).

   (b) **Warp Version**: Synchronization across all threads within a warp (__syncwarp()). This leads to much less synchronization overhead and increases resource utilization as well since the threads don't need to wait for as long before continuing to execute.

4. **Inlier Count Reduction**

   (a) **Block Version**: We used CUDA's built in atomicAdd() to add the local count into the shared copy atomically.

   (b) **Warp Version**: There were 2 approaches we considered. The first approach involve storing an array of 8 integers, 1 for each warp to track their inlier counts and we would use atomicAdd to have each lane within the warp increment their corresponding count. The other approach is to use CUDA's intrinsic __shfl_down_sync, which we ended up going with.

Explanation of the Updated Inlier Count Reduction The shuffle intrinsic is the most optimal way to perform a reduction within a warp due to many reasons, and is significantly faster than the shared memory approach.

```
// Version 1: Atomically Reduction with Shared Memory
s_inlier_cnt[8] = {0,0,0,0,0,0,0,0};
atomicAdd(&s_inlier_cnt[WARP_ID], local_inlier_cnt);

// Version 2: Shuffle Reduction
for (int offset = WARP_SIZE / 2; offset > 0; offset /= 2) {
  local_inlier_cnt += __shfl_down_sync(0xFFFFFFFF, local_inlier_cnt, offset);
}
```

The __shfl_down_sync intrinsic allows the thread to access data that is offset ahead of us and perform computations with it. For instance, if lane ID is 0 and offset is 4, it reads laneID 3's value for local_inlier_count and adds it to our own. The loop performs a parallel tree-based reduction to sum values across the warp without using shared memory. By iterating with offsets of 16, 8, 4, 2, and 1, each thread adds a value from a neighbor "offset" positions away, effectively collapsing the partial sums in $O(\log N)$ steps until Thread 0 holds the total count for the entire warp.

The benefits of version 2 above version 1 are as follows:

1. **Data Storage / Access**: For version 1 with atomic reduction, everything is stored in shared memory. On the other hand, the shuffle reduction performs all the operation entirely in registers which means much faster access.

2. **Contention**: Atomic operations means serializing that specific operation. In other words, version 1 entail each lanes within a warp would take turns updating their respective inlier count. Version 2's shuffle reduction occurs in parallel for each time step (ex: thread 0 reads/adds thread 1's value to itself as thread 2 reads/adds thread 3's value to itself).

3. **Time Complexity**: Version 1 will take O(N) time where N is the number of lanes in a warp (32 on the GHC machines), whereas Version 2 takes O(logN) time because that's the amount of steps needed for reduction.

3.3 Performance

3.3.1 Scaling Up Number of Iterations

Table 1: RANSAC Performance: Iteration Scaling (Fixed $N = 5000$)

Iterations	**Py (ms)**	**Block (ms)**	**Warp (ms)**	**Speedup (vs Py)**	**Speedup (vs Blk)**
1,000	251.19	5.83	3.03	82.8×	1.9×
5,000	1,256.14	27.73	12.18	103.1×	2.3×
10,000	2,508.40	54.93	23.64	106.1×	2.3×
50,000	12,511.20	273.38	92.28	135.6×	3.0×
100,000	25,004.19	479.88	184.50	135.5×	2.6×
200,000	50,142.96	941.38	368.16	136.2×	2.6×
400,000	100,261.19	1,782.73	735.32	136.4×	2.4×

For the speedup comparison between the GPU warp version and the CPU, the speedup originally increases, but once we hit 50,000 iterations the speedup actually started becoming the same regardless of the amount of iterations. At low iterations, the GPU is not fully saturated, so the fixed overheads of CUDA like kernel launch overhead occupy a larger percentage of the total runtime. Since not all the SMs in the GPU are fully utilized, we aren't able to hide the overhead and amoritize its cost.

However, once we hit 50,000 iterations, increasing the number of iterations no longer improve the speedup. This meant that at 50,000 iterations is when all the SMs are all used, and we are now compute bound. At such high iteration is when the serial parts of the pipeline (SVDs) dominate the runtime and the pipeline is saturated, meaning the fraction of the serial work is a higher percentage of the overall runtime.

For the speedup comparison between the block and warp version of RANSAC, it peaked at 50,000 iteration and actually started decreasing afterwards. Initially up until 50,000 iterations, it made sense for the warp version to perform better than the block version as we are better utilizing the GPU (less idle time), and the shuffle reduction across 32 threads had better performance than

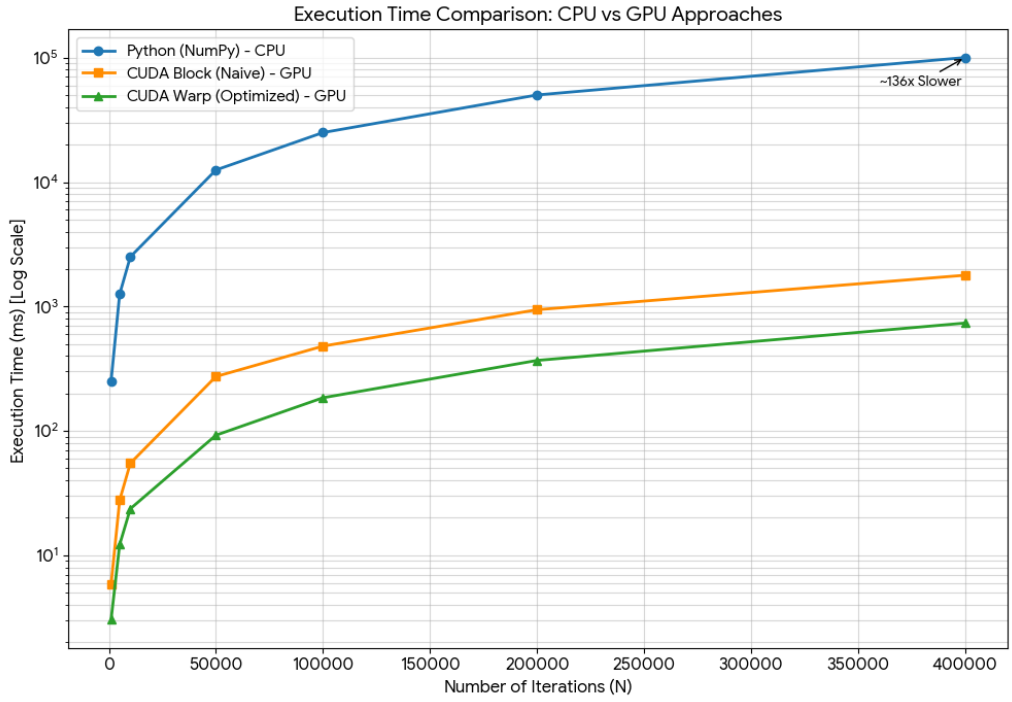

Figure 1: RANSAC Execution time for the 3 approaches on different RANSAC iteration counts

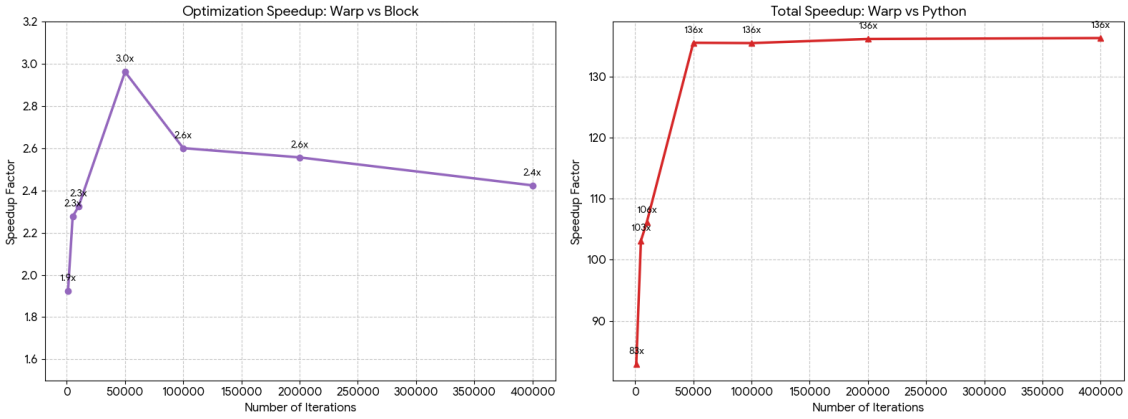

Figure 2: RANSAC Speedup between GPU warp vs. GPU block and CPU vs. GPU warp

the serial atomic add across all 256 threads in the block version.

Once we pass the 50,000 iteration point, the improvement of warp in comparison to block actually starts to decrease due to the fact that the serial computation is now the bottleneck of our program, and we no longer see additional benefit from higher utilization of our GPUs (we are at the max utilization for both), and the shuffle reduction cannot improve performance even more. This, the speedup ratio starts to stabilize.

3.3.2 Scaling Up Number of Points

Table 2: RANSAC Performance: Point Scaling (Fixed Iterations = 5000)

Points (N)	**Py (ms)**	**Block (ms)**	**Warp (ms)**	**Speedup (vs Py)**	**Speedup (vs Blk)**
1,000	730.92	21.46	9.48	77.1×	2.3×
5,000	1,249.06	27.62	12.09	103.3×	2.3×
10,000	1,810.74	28.27	12.52	144.6×	2.3×
50,000	6,482.52	33.47	16.61	390.4×	2.0×
100,000	13,085.37	39.94	22.49	581.8×	1.8×
200,000	32,711.59	71.43	45.43	720.0×	1.6×
400,000	76,257.51	116.55	82.85	920.4×	1.4×

There is massive speedup as we increase the number of points from the CPU implementation due to the high amounts of parallelism available. As the number of points scale, the amount of epipolar distance computation increases (which is parallel), which decreases the percentage of the serial parts of our pipeline (steps 1 and 2). Thus, due to Amdahl's law, this means the serial portions have much less effect on limiting speedup. Especially at high number of points, GPU checks many more points in parallel and thus it can absorb the extra computation much better than the CPU. At high number points, doubling the amount of points means more than double the runtime for CPU, but actually less than double for GPU versions.

An interesting trend between the block and warp speedup is that the warp version actually becomes less better as the number of points scale. This is due to the fact that as the number of points increase, we switch from being compute bound to now memory bound. At low number of points, the cost is dominated by the reduction, so the warp performs significantly better than the block version. However, at high number of points, the runtime is dominated by the computation of epipolar distances and reading the data from global memory. For both of these portions, the two versions operate exactly the same (reads the same amount of data, performs the same computations), hence why the speedup between the warp and block version starts to become less and less apparent.

3.3.3 Performance Analysis on Scaling Up

The trends as we increase the number of points vary greatly from the trends as we increase the number of iterations (analyzed above). Based on the graphs, our algorithm scale much better to high number of points compared to high number of iterations, which is ideal as RANSAC often doesn't need an extremely high number of iterations to converge, but it is beneficial for RANSAC

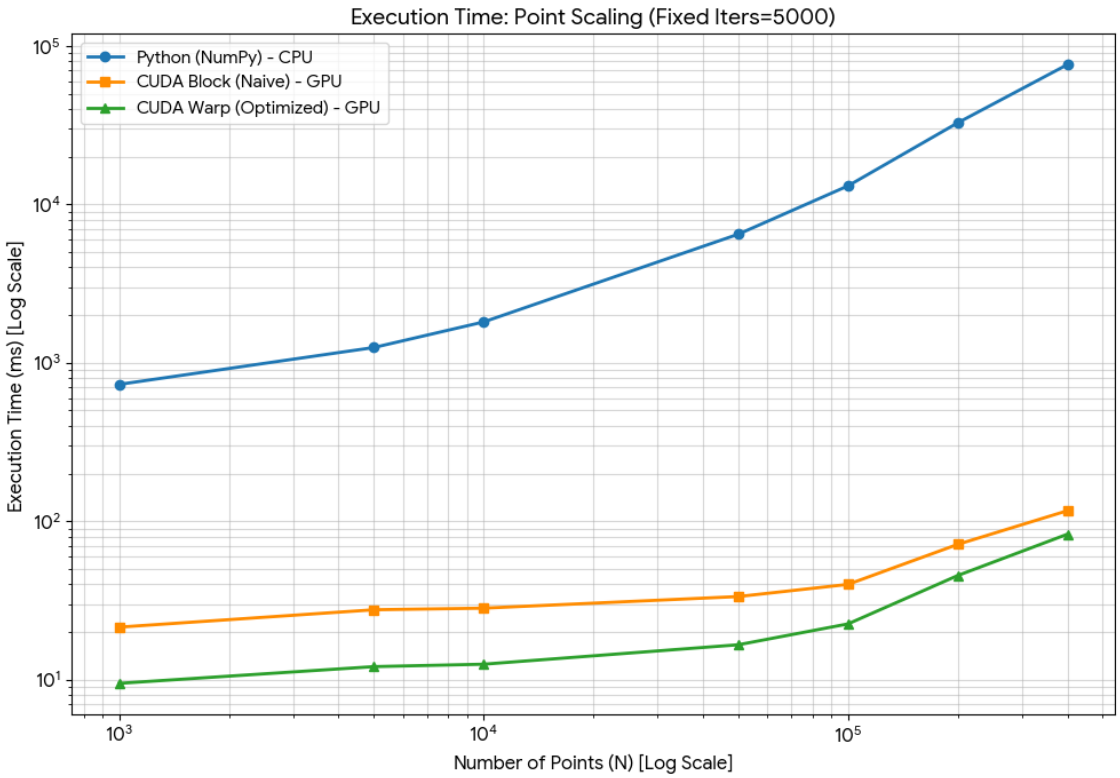

Figure 3: RANSAC Execution time for the 3 approaches on different amounts of points

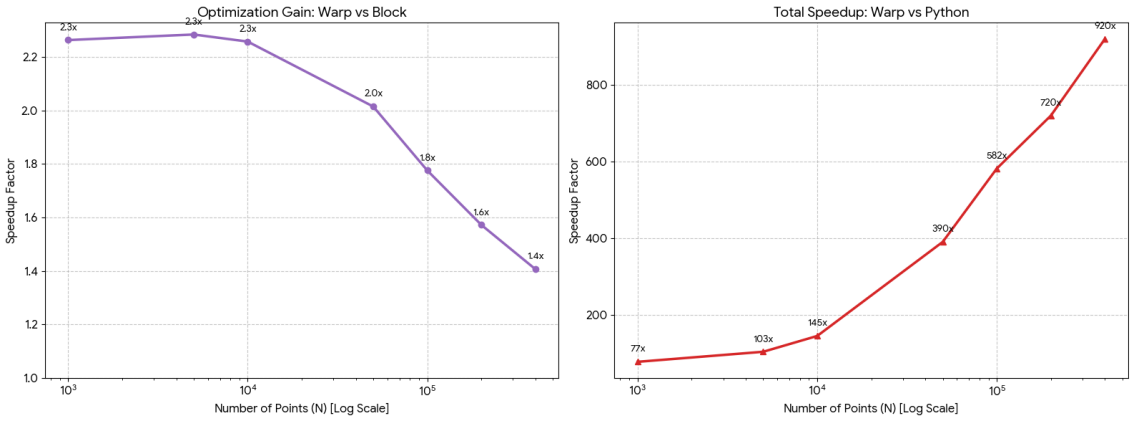

Figure 4: RANSAC Speedup between GPU warp vs. GPU block and CPU vs. GPU warp

to be able to handle many points at once for complex or high dimensional images/videos.

Increasing the number of iteration meant increasing the importance of steps 1 and 2 of the pipeline, which is serial, hence why the speedup stabilized at a much smaller number. On the other hand, increasing the number of points meant increasing the importnace of step 3, which is why the speedup increases significantly as we increase the number of points, even at high numbers like 400k.

3.4 Triangulation

To reconstruct 3D geometry from 2D correspondences, we solve for the 3D point $\mathbf{X} = (x, y, z)$ that minimizes the reprojection error across two views. For a pair of camera matrices $P_1, P_2 \in \mathbb{R}^{3\times4}$ and corresponding 2D points $\mathbf{x}_1 = (x_1, y_1), \mathbf{x}_2 = (x_2, y_2)$, this problem is formulated as a homogeneous system of linear equations $A\mathbf{X} = \mathbf{0}$.

Each correspondence yields two constraints per camera, resulting in a system where $A \in \mathbb{R}^{4\times4}$:

$$A = \begin{bmatrix} u_1\mathbf{p}_1^{3\top} - \mathbf{p}_1^{1\top} \\ v_1\mathbf{p}_1^{3\top} - \mathbf{p}_1^{2\top} \\ u_2\mathbf{p}_2^{3\top} - \mathbf{p}_2^{1\top} \\ v_2\mathbf{p}_2^{3\top} - \mathbf{p}_2^{2\top} \end{bmatrix}$$

The solution $\mathbf{X}$ corresponds to the right singular vector of A associated with the smallest singular value.

The workload is "embarrassingly parallel" but granular. We have N independent correspondences, each requiring solving for the SVD of a 4x4 matrix. The primary challenge is ensuring the overhead of kernel launches and global memory transactions does not overshadow the matrix operations in computation time.

3.4.1 Initial ideas

Attempt 1: Global Memory Bound (cuSOLVER) Our initial strategy leveraged NVIDIA's cuSOLVER library. While robust, cuSOLVER is optimized for large matrices where computational intensity naturally hides latency. For our small 4×4 matrices, the overhead of the "batched" interface was prohibitive. Further investigation into the library indicated that the SVD implementation made several global memory accesses. The need to have 2 more kernels to prepare the inputs and extract them also caused suboptimal performance.

```
// High latency: Data makes multiple round-trips to DRAM
void triangulate_naive(pts1, pts2, OutputP) {

    // Step 1: Kernel to format data (Global Mem Write)
    // Launch N threads
    prepare_A_kernel<<<N>>>(pts1, pts2, A_global_buffer);
    cudaDeviceSynchronize();

    // Step 2: Library Call (Global Mem Read/Write)
    // Operates on data in DRAM. High overhead for small N.
    cusolverDnSgesvdjBatched(handle,
                             A_global_buffer,
                             S_global_buffer,
                             V_global_buffer, // Singular vectors
                             N);
```

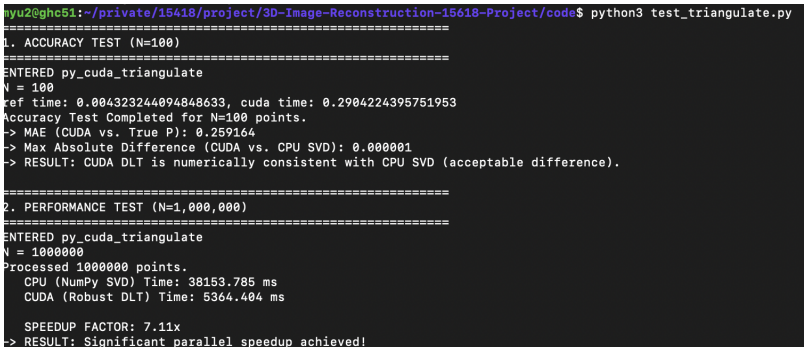

Figure 5: Triangulation Using cuSOLVER's SVD implementation

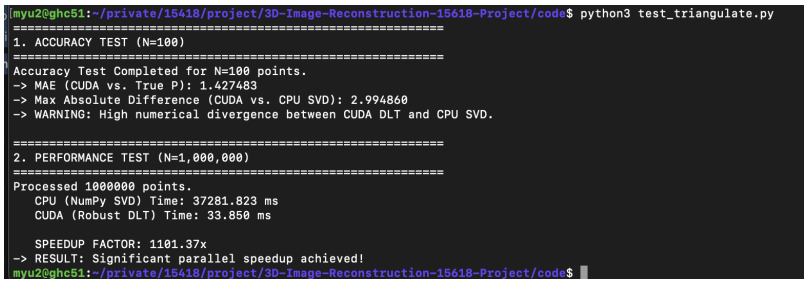

Figure 6: Triangulation Using DLT solving implementation

```
16      cudaDeviceSynchronize();
17
18      // Step 3: Extract Result (Global Mem Read)
19      // Launch N threads
20      extract_P_kernel<<<N>>>(V_global_buffer, OutputP);
21  }
```

Listing 1: Naive Pipeline using cuSOLVER

As seen in Figure 5, speedup only materialized at extreme scales ($N > 10^6$), which is unrealistic for this project as test images only had a few hundred corresponding points on average and up to a few thousand in rare cases.

The second idea for solving the SVD involved a simple DLT solver. This was in an attempt to use a faster method that would sacrifice some accuracy for performance. This also came with the added benefit of allowing us to fuse the kernels for the preperation and extraction part of the implementation, which will save time by reducing the number of kernel launches, which can be expensive operations. This implementation was the opposite of the cuSOLVER implementation, in that while it was extremely fast due to the lack of global variable use and a fused kernel, it struggled with accuracy, deviating significantly from the CPU implementation of triangulate.

3.4.2 Final Implementation Choice

To achieve both performance and accuracy, we implemented a custom "Fused" SVD kernel based on the One-Sided Jacobi algorithm (Hestenes, 1958).

```
// High Throughput: Data stays in registers
__global__ void triangulate_fused(pts1, pts2, OutputP) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // Step 1: Read Global Memory only once
    float u = pts1[i].u;
    float v = pts1[i].v;

    // Step 2: Build A and V locally
    // 'A' and 'V' are local arrays (mapped to registers)
    float A[16];
    float V[16];

    build_A_in_registers(u, v, A);
    initialize_identity(V);

    // Step 3: Compute SVD using iterative jacobi SVD
    #pragma unroll
    for (int iter = 0; iter < 4; iter++) {
        for (int p = 0; p < 3; p++) {
            for (int q = p + 1; q < 4; q++) {
                // Dot products and Givens rotations
                // computed entirely on local registers
                apply_rotation(A, V, p, q);
            }
        }
    }

    // Step 4: Write Result only once
    // Extract column corresponding to min singular value
    float3 P_local = extract_min_col(A, V);
    OutputP[i] = P_local;
}
```

Listing 2: Fused Kernel using Register-Level One-Sided Jacobi

This implementation minimizes global memory traffic to exactly one read (input) and one write (output) per point. As shown in the table below, this yields a $\sim 220\times$ speedup over the CPU baseline for standard workload sizes ($N = 10,000$).

Performance of triangulate.cu (GPU implemention) vs triangulate.py (CPU Implementation)

# Correspondent Points	CPU Time (ms)	GPU Time(ms)	GPU Speedup
1	0.082	0.180	0.46
10	0.448	0.192	2.33
100	4.155	0.207	20.03
1000	37.997	0.342	110.98
10000	379.908	1.720	220.91
100000	3965.583	15.707	252.48
1000000	38552.759	168.694	228.54

3.5 Adapting for multiple image pairs

While our kernel successfully reconstructs 3D geometry from a single image pair, extending this to a sequence of N images requires addressing two fundamental problems: **Error Accumulation** and **Scale Ambiguity**.

3.5.1 Bundle Adjustment

As we chain multiple pairwise reconstructions, small errors in camera pose estimation (R, t) and triangulation accumulate, leading to "drift" where the reconstructed trajectory diverges from reality. To correct this, we implemented **Bundle Adjustment**. BA is a non-linear least squares optimization problem that simultaneously refines 3D point coordinates $\mathbf{X}_j$ and camera parameters P_i to minimize the total reprojection error:

$$\min_{\mathbf{P},\mathbf{X}} \sum_{i=1}^{N} \sum_{j=1}^{M} v_{ij} \|\mathbf{x}_{ij} - \pi(\mathbf{P}_i, \mathbf{X}_j)\|^2 \tag{1}$$

Where π is the projection function and v_{ij} is a visibility indicator (1 if camera i observes point j, else 0).

Due to time constraints, we utilized a serial implementation of BA using a least squares regressor to find the minimum re-projection error. This algorithm is very slow relative to the RANSAC and triangulation parts of the pipeline, scaling with a larger number of points. To mitigate this, we set a limit on the number of points that could be passed into bundle adjustment at 200, which prevented frames with many correspondences from taking extremely long to converge. Despite this, BA was still a major slowdown point for the pipeline, and is a key target for future optimizations.

3.5.2 Scaling

The translation vector t between two cameras is only recovered up to an unknown scale factor. Mathematically, the decomposition of the Essential Matrix E yields a unit translation vector ($\|t\| = 1$). This then means that the reconstruction of Pair $(1, 2)$ exists in a different metric space than Pair $(2, 3)$. To combine these disjoint reconstructions, we implemented a *Relative Scale Estimation* function in order to rescale the computed points. This serial algorithm is shown below:

1. **Identify Overlaps:** We find the set of 3D points $\{X_i\}$ visible in all three views (Images 1, 2, and 3).
2. **Compute Ratios:** For every pair of shared points (X_a, X_b), we calculate the distance between them in the first reconstruction (d_{12}) and the second reconstruction (d_{23}).
3. **Derive Scale Factor:** The scaling ratio $s = \text{median}(\{d_{12}/d_{23}\})$ is computed robustly to reject outliers.
4. **Apply Transform:** All points and camera centers in the second reconstruction are scaled by s to align with the first coordinate system.

3.5.3 Multi Image Reconstruction Output

After implementing BA and Scaling, we are now able to make a looping reconstruction pipeline that builds a 3D shape from multiple image pairs instead of just a single image pair.

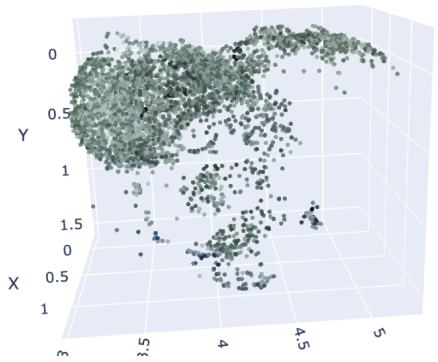

Figure 7: Reconstructed Object from 20 Images

4 Results

We measured the GPU performance with CUDA kernel against the numpy/OpenCV implementation using Python. One thing to note is that by default numpy vectorizes using SIMD instructions, so there are some parallelism even in our Python implementation (but not handled by us). However, numpy runs on CPU, so our speedup will be comparing the GPU speedup versus Python numpy's vectorized SIMD instructions.

For most computer vision practitioners, numpy is the default tool for prototyping. Measuring the speedup relative to numpy provides the most practical metric for a developer deciding whether the complexity of porting a pipeline to CUDA is worth the performance gain.

This section will only compare the full pipeline results to individual step's performances. For detailed analysis on the RANSAC and triangulation results, look at the previous sections.

4.1 Performance of Overall Pipeline (RANSAC + Triangulation/FindM2)

4.1.1 Iteration Scaling

Table 3: Full Pipeline Performance: Iteration Scaling (Fixed $N = 5000$)

Iterations	**Py (ms)**	**Block (ms)**	**Warp (ms)**	**Speedup (vs Py)**	**Speedup (vs Blk)**
1,000	647.92	8.99	6.00	107.9×	1.5×
5,000	1,672.01	30.63	15.15	110.4×	2.0×
10,000	2,955.94	57.88	26.58	111.2×	2.2×
50,000	13,212.98	255.18	95.15	138.9×	2.7×
100,000	26,007.04	511.22	188.67	137.8×	2.7×
200,000	51,600.99	955.51	373.42	138.2×	2.6×
400,000	102,421.16	1,799.79	741.96	138.0×	2.4×

The peak speedups for the full pipeline are slightly lowered or shifted compared to RANSAC only when comparing the two GPU versions. This is due to the fact that the findM2/triangulation

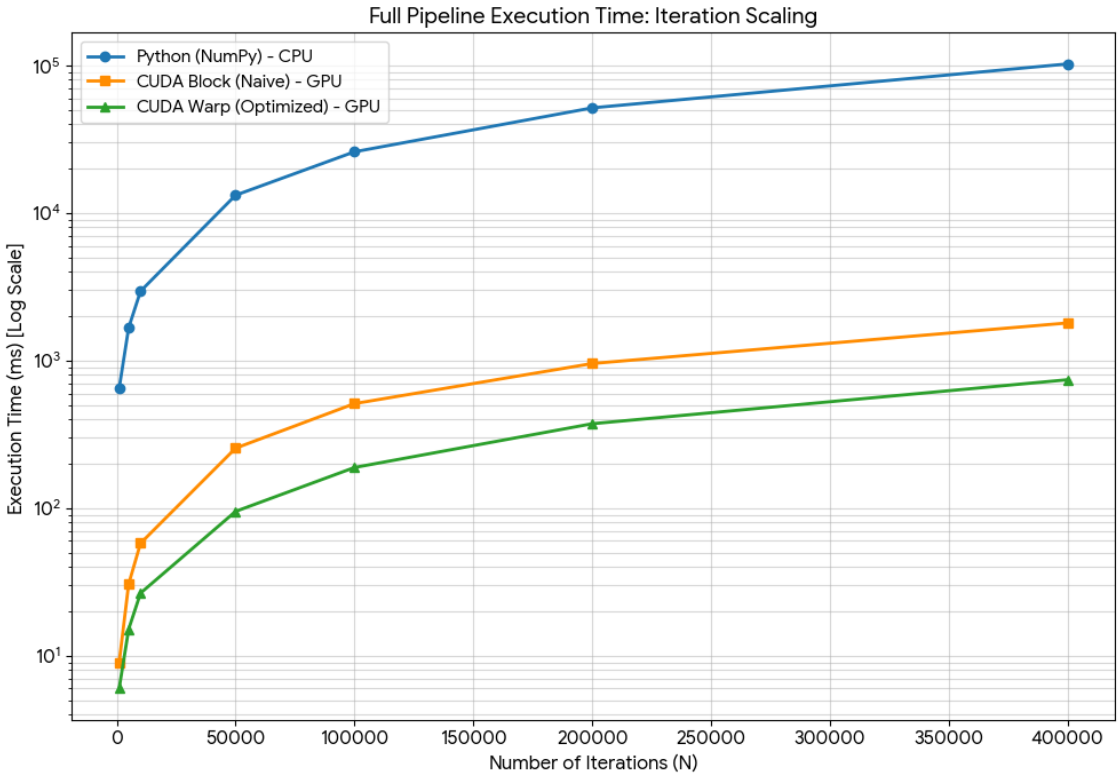

Figure 8: Full Pipeline Execution time for the 3 approaches on different amounts of points

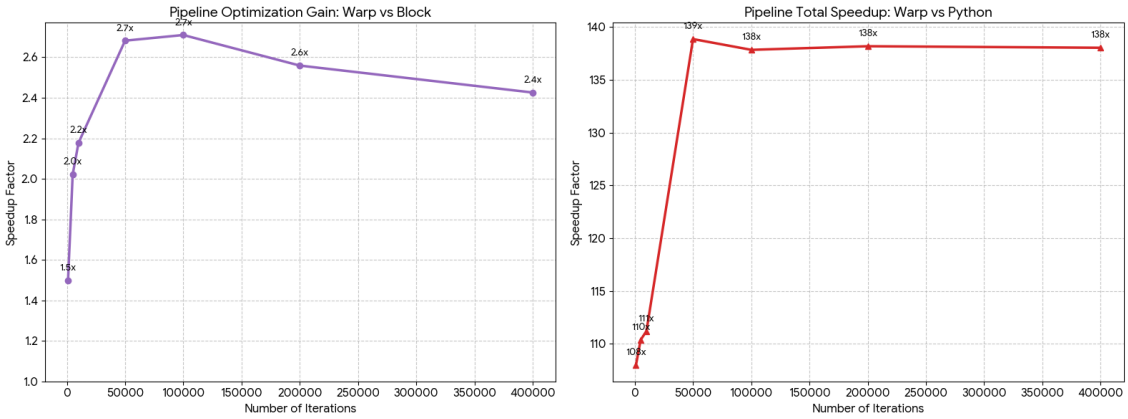

Figure 9: Full Pipeline Speedup between GPU warp vs. GPU block and CPU vs. GPU warp

is completely separate from the RANSAC optimizations, and acts as a fixed overhead that has the same runtime regardless of which GPU RANSAC version we use. This is a demonstration of Amdahl's law where as the parallelized/optimized part becomes faster, the part that is the same (findM2) dictates the total runtime ceiling.

As the number of iterations increases, the speedup curve look extremely similar to the RANSAC only plots. This is because at high iteration counts, the RANSAC is taking much longer than findM2 (which does not get effected by the number of iterations at all). Thus, the pipeline becomes compute bound by the RANSAC loop and thus the performance characteristics are very similar to just running the RANSAC loop.

4.1.2 Point Scaling

Table 4: Full Pipeline Performance: Point Scaling (Fixed Iterations = 5000)

Points (N)	**Py (ms)**	**Block (ms)**	**Warp (ms)**	**Speedup (vs Py)**	**Speedup (vs Blk)**
1,000	822.37	23.04	10.94	75.2×	2.1×
5,000	1,663.27	30.59	15.12	110.0×	2.0×
10,000	2,644.82	33.07	17.29	152.9×	1.9×
50,000	10,742.55	52.81	36.07	297.8×	1.5×
100,000	21,704.71	93.75	61.01	355.8×	1.5×
200,000	50,186.17	150.60	126.68	396.2×	1.2×
400,000	109,820.98	281.90	235.30	466.7×	1.2×

The speedup for warp vs block version for the full pipeline drops significantly faster than the one in the RANSAC-only version. For instance, for 400k points, RANSAC only still observes a 1.4x speedup whereas the full pipeline only sees a 1.2x speedup. This showcases a memory bottleneck in the findM2 stage, as the warp optimizations no longer allow for a high speedup. At high point counts, as seen in the execution graphs, the warp and block version actually starts to converge to similar execution times, since findM2 starts to become the bottleneck, and it behaves the exact same for the two versions (as the optimization is purely for RANSAC). Thus, at high point counts, the global memory bandwidth gets exhausted within findM2 and thus the warp optimizations and improvements become negligible.

The overall speedup compared to CPUs is much lower for the full pipeline compared to only running RANSAC due to a variety of reasons. First, since the RANSAC loop is heavily optimized while we perform 4 triangulation operations serially within findM2 (individual triangulations are parallelized, but we run one iteration at a time), Amdahl's law tells us that the bottleneck will dominate the runtime and thus limit the speedup we are able to achieve. Additionally, when only running RANSAC, we only need to compute the max inlier count. However, in order for findM2 to work, we need to iterate through all of the points and determine which ones are the inliers based on the best fundamental matrix, which is bandwidth bound and thus limits speedup as well.

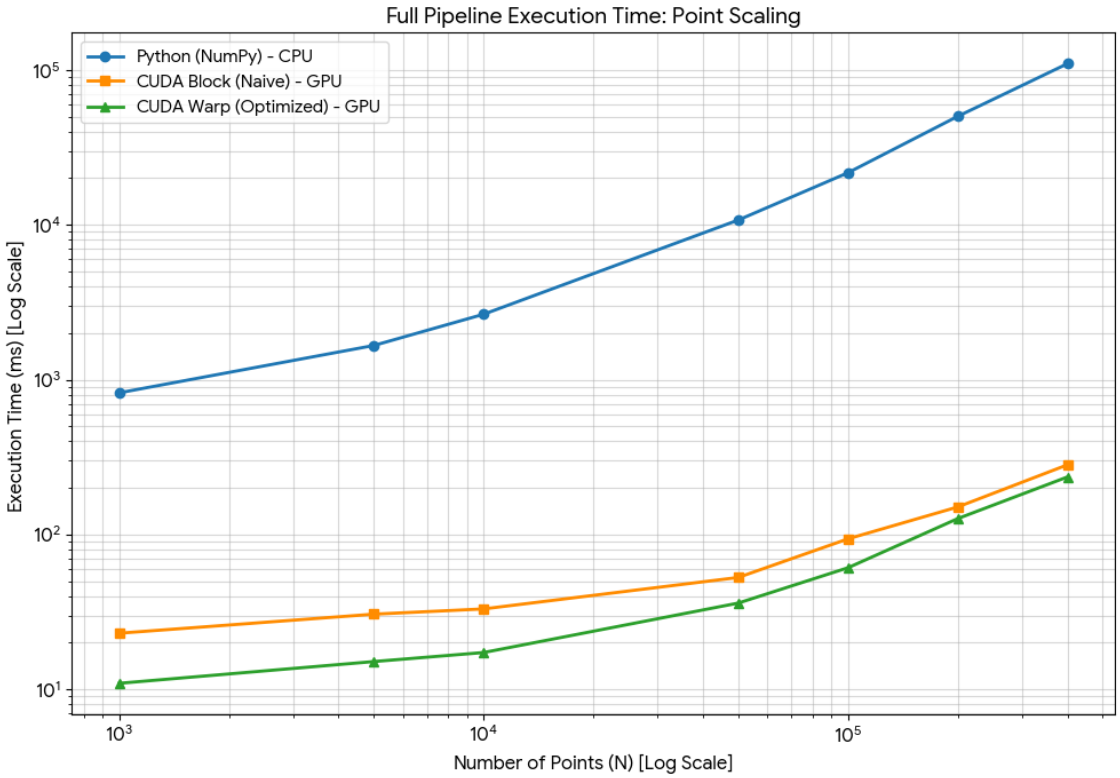

Figure 10: Full Pipeline Execution time for the 3 approaches on different amounts of points

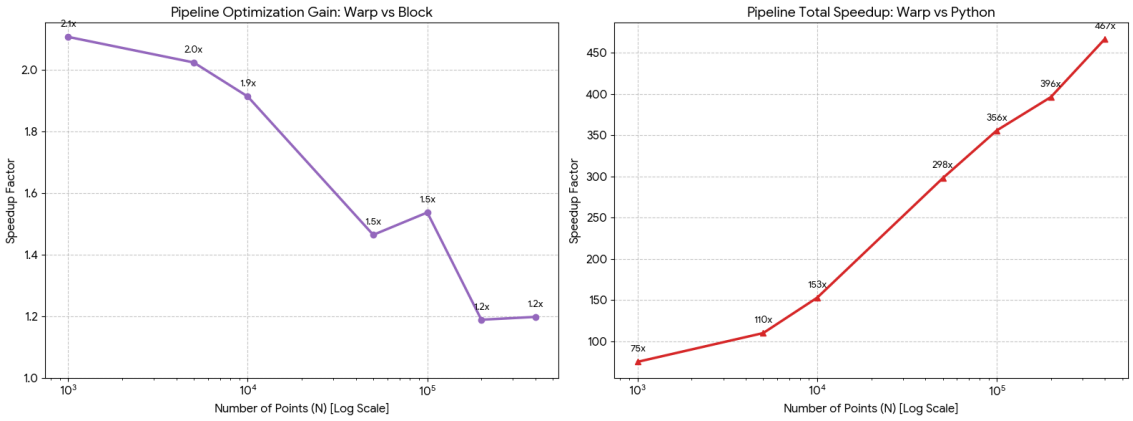

Figure 11: Full Pipeline Speedup between GPU warp vs. GPU block and CPU vs. GPU warp

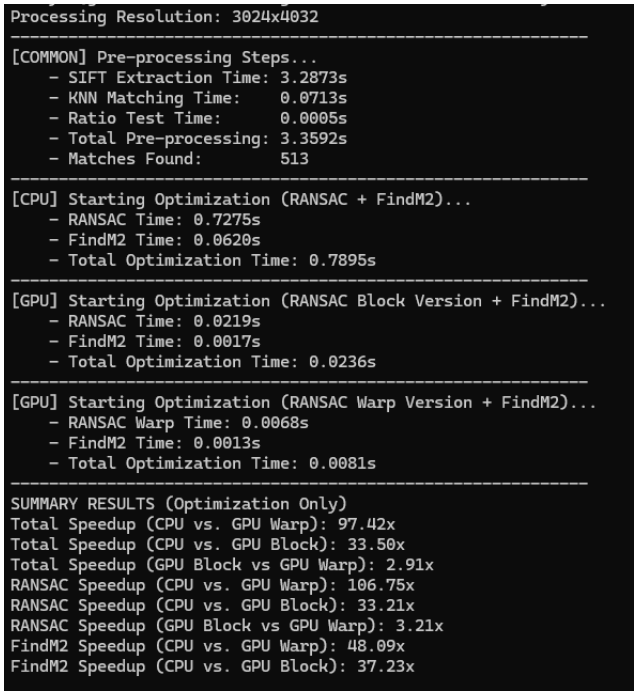

Figure 12: Execution Time & Speedup After Running on PokePlush_09.jpg and PokePlush_10.jpg

4.2 Result of Running Pipeline on Actual Images

A 3D interactive plot of the reconstruction can be found here: `https://colab.research.google.com/drive/183V7dkLI8Ac88ZAJOIh7rUxdzgU-ze9y?authuser=1#scrollTo=o4DI5aeOvNzq`.

The terminal output summarizes the whole workflow starting from preprocessing, along with the performance of each of the 3 implementations of the pipeline that we optimized. As seen by the output, RANSAC achieves 106.75x speedup from the CPU version, and FindM2 achieves 48.09x speedup. This makes sense since the number of inliers for this image is only 396 for CPUs and 406 for GPUs. GPU has slightly more work than CPUs, and also the speedup for findM2 is not super apparent as the number of points is not high to a point where we could utilize the GPU fully. However, unlike findM2 where the speedup determines on the amount of points only, RANSAC's speedup depends on both the number of iterations and the number of points. This is why RANSAC was able to observe good speedup despite the limited amount of points.

The two approaches have very similar number of inlier counts, which validates the correctness of our GPU implementation. The reason why we weren't able to completely mimic the CPU's results is due to the randomness of the RANSAC algorithm - for each iteration, it randomly picks 8 points and attempts to find the fundamental matrix. The only way for us to ensure the two uses same points for each iteration would be to pregenerate the nonrepeating indexes for every iteration, which is infeasible and unnecessary. Thus, we use the relative number of inliers along with the

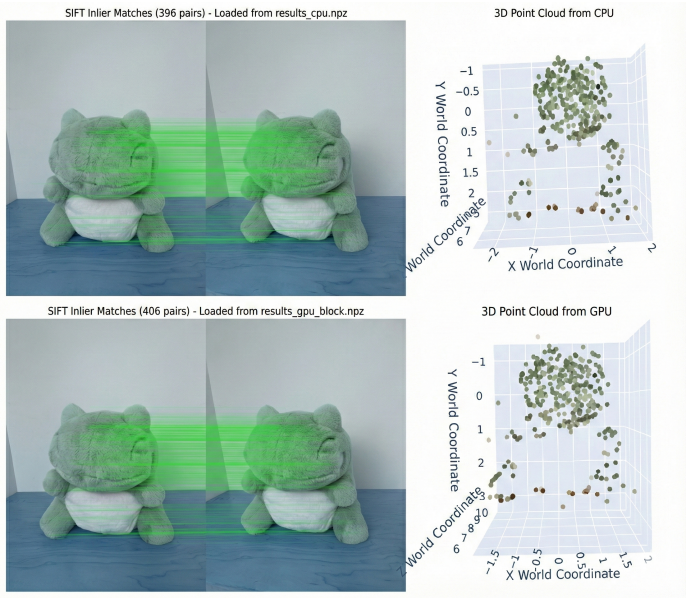

Figure 13: Correspondence Pipeline and 3D Reconstruction Final Result

reconstruction output to determine the corerctness.

As seen in the left two images, each green line represents an inlier pair after the RANSAC algorithm. It maps a point on image 1 to its corresponding point on image 2. The two images on the right shows the final reconstructed output after taking all the inliers from RANSAC and running the findM2 / triangulation algorithm. As seen in the images, it is able to pretty closely reconstruct the shape of the plush, with the head being where the majority of the inliers lie.

4.2.1 Speedup Limitations and Bottlenecks

While we achieved significant speedups ($> 100\times$), our performance was bounded by distinct factors depending on the input size and kernel configuration.

- **Memory Bandwidth (The "FindM2" Bottleneck):** As shown in Table 4, as the number of points (N) scales to 400,000, the speedup of the Warp implementation over the Block implementation drops to $1.2\times$. At this scale, the pipeline becomes **memory-bound**. The triangulation kernel requires streaming massive amounts of coordinate data from global memory. Since the arithmetic intensity of a simple SVD/linear solve is relatively low compared to the memory traffic required to fetch coordinates and write back 3D points, we saturate the global memory bandwidth. No amount of compute optimization (like the warp shuffle reduction optimization) can overcome the bus speed limits here.

- **Amdahl's Law (Serial Overhead):** In the full pipeline, the non-parallelizable portions, specifically the Python-side preprocessing, memory allocations, and the final reduction on the host to select the best Fundamental Matrix, act as a limit on execution time. For small datasets ($N < 1000$), the PCIe data transfer overhead and kernel launch latency constitute a

significant percentage of the runtime, preventing the GPU from achieving peak throughput. Additionally, between RANSAC and findM2, there were multiple components that were implemented in Python on the CPUs (computing inlier mask, findM2's loop) which also limits the performance gains we gathered from the parallel execution.

- **Warp Divergence:** Although we utilized `__shfl_down_sync` to reduce divergence during the reduction phase, the RANSAC model evaluation step still suffers from minor divergence. In the "check inliers" loop, threads within a warp may diverge if the decision branch ('error ¡ threshold') is taken by only half the warp. While this is unavoidable in RANSAC, our data-parallel approach minimizes this cost compared to a task-parallel approach.

4.2.2 Deeper Analysis: Execution Breakdown

As seen in Figure 12, for approximately 500 matches to compute (with around 400 inlier matches at the end of RANSAC), the exeucution time is broken down as follows:

1. **RANSAC Kernel (∼92% on CPUs, ∼84% on GPUs):** The pipeline spends the majority of the time exploring different hypothesis within the RANSAC algorithm. For the CPU, it takes a larger percentage of overall runtime compared to the GPUs, meaning that the RANSAC performance gains are more apparent. This is also because the number of iteration was fixed at 1000, which means RANSAC is a lot more computationally expensive than triangulation.
2. **Triangulation Kernel (∼8% on CPUs, ∼16% on GPUs)**: Triangulation does not take up as much of the time since there are only 400 inliers to compute and thus it is not as computationally expensive.

Room for Improvement: The most significant room for improvement lies in Multi-Stream Concurrency. Currently, RANSAC and Triangulation run sequentially and in separate kernel functions. However, since the triangulation of inliers from a *previous* frame pair is independent of the RANSAC calculation for the *current* frame pair, these could be pipelined using CUDA Streams to hide the latency of the memory-bound Triangulation behind the compute-bound RANSAC. As we scale up to handle multiple images in parallel, this would be useful in both maximizing the utilization of our GPUs, pipeline the operations to hide latency, reduce the amount of data transfers between host and device, and limit kernel launch overhead.

Additionally, preprocessing steps also take a significant amount of the total runtime as it is completely done on CPU. For the CPU version, it takes up 80.96% of the runtime, and for the GPu version it takes up 99.76% of the runtime. These steps could also be parallelized, and that would increase the overall runtime beyond the RANSAC and triangulation. More specifically, SIFT extraction and ratio test benefits greatly from data parallelism, and kNN could be optimized using task parallelism (each thread computes the KNN for 1 point pair).

4.2.3 Justification of Machine Target

GPU was the correct choice for this optimization problem due to the fact that 3D reconstruction is a throughput oriented problem. For RANSAC, the goal is to explore as many hypothesis (fundamental matrix) as possible to find the one that yields the max number of inliers. The latency of

individual iterations is not as important. GPUs are designed for massive data and task parallelism, so we could utilize all the tiny cores on the GPU to test and explore many iterations.

Additionally, this problem requires reading from the pts1 and pts2 array multiple times, which makes it a bandwidth bound problem at high point counts. GPUs are designed to handle higher memory bandwidth compared to CPUs, which also makes it a better choice.

Choosing a CPU over a GPU implementation would mean we optimize how fast each individual iteration runs. However, due to the lack of cores, the amount of iterations we can test simultaneously is severely less than on the GPU, hence the overall runtime of the algorithm would be much longer.

5

References

[1] M. A. Fischler and R. C. Bolles. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Communications of the ACM*, 24(6):381–395, June 1981.

[2] M. R. Hestenes. Inversion of Matrices by Biorthogonalization and Related Results. *Journal of the Society for Industrial and Applied Mathematics*, 6(1):51–90, 1958.

[3] D. G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[4] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2nd edition, 2003.

[5] Matthew Yu. Assignment 4: Reconstruction. 16-385 Computer Graphics, Fall 2025. Course assignment, CMU. Provided Serial Reconstruction Pipeline to match against GPU implementations.

6 Contributions & Grade Distribution

1. **Matt (50%)**: Serial pipeline, triangulation parallelization, bundle adjustment & scaling up to many images
2. **Emily (50%)**: RANSAC parallelization (warp & block version), consolidated pipeline, plotting colab & env setup