Parallel Synthetic Optical Flow Rendering

Team Members: Allen Wu, Steven Yang

https://wuallenwu.github.io/

https://github.com/wuallenwu/418FinalProject

Summary

Synthetic data generation is crucial in data-limited fields like robotics. We will implement a GPU-accelerated approach which parallelizes this heavy workload across pixels, achieving real-time ground-truth optical flow synthesis for high-resolution training data.

Background

- What is the application or algorithm you are implementing?

  GPU-accelerated synthetic optical flow generation. Given a 2D starting pixel map, and 3D movement of the camera (translation and rotation), compute the 2D pixel map after the movement.

- What are the key data structures and operations?

  Input: Depth maps (16-bit), camera intrinsics (fx, fy, cx, cy), extrinsic poses (SE(3) 4×4 matrices)
  Pipeline: Image coordinates → camera frame → world frame → current camera frame → new image coordinates
  Operations: Coordinate transformations (matrix multiplications), reprojection, bounds checking
  Output: 2D flow vectors (u, v displacements) and validity masks per pixel

- What aspects of the problem might benefit from parallelism and why?

  This problem is intrinsically embarrassingly parallel, as each of the calculations are independent on a per pixel basis. Any decently sized image will have hundreds of thousands of pixels to compute values over, meaning that we can have good computation-to-speedup ratio, even if the per-pixel work is small. Real-world robotics systems require low-latency optical flow predictions between camera frames, making real-time performance essential.

- Are there any figures, diagrams, or pseudocode examples that would help illustrate your idea?

  The general flow of the cuda part will be something like this.

  For each pixel (u, v) in parallel across GPU threads:

  1. Load depth value at this pixel If depth is invalid → mark pixel as invalid, skip
  2. Convert 2D pixel (u, v) + depth into 3D point in reference camera
  3. Transform that 3D point from reference camera → world coordinates
  4. Transform that 3D point from world coordinates → current camera

5. Project the 3D point back down to 2D image coordinates in current camera If projected point is outside image bounds → mark pixel as invalid, skip

6. Optical flow = (new 2D position) - (original 2D position) Store the flow vector (du, dv)

The Challenge

Workload Characteristics

This problem is difficult because there can be highly divergent execution. Some pixels go entirely out of frame, some pixels are translated drastically, and some pixels barely move. Memory bandwidth is shared between threads, and we have memory strain at high image resolutions. At such high scale, we need to cleverly hide the calculations (12 multiply-accumulate operations) with actual data movement. Additionally, errors compound through coordinate transformations, making validation against ground truth critical.

System Constraints

A big question in this problem is how we actually want to represent the data—does it make more sense to send chunks of pixels into the threads, or one pixel by itself? What does shared memory look like? We need to carefully consider the hardware limitations. The underlying depth array is stored in 1-D, so we must be careful with memory accesses in the same warp to avoid cache misses. We also want to restructure computation to ensure that as much as possible is stored in registers rather than caches.

Resources

We will be starting from scratch, but drawing inspiration from code written for homework 2. Our inspiration will primarily be from the following article, which introduces the main algorithm: https://arxiv.org/pdf/2411.04413 (See IVB, rendering optical flow).

We will be running this code on the GPU resources available through the GHC lab (NVIDIA RTX GPUs). We will use OpenCV for image I/O and pose file parsing.

Goals and Deliverables

Plan to Achieve (Required Goals)

1. *Goal 1*—Create a CUDA kernel that runs our parallel synthetic optical flow rendering on 1920×1080 resolution images

2. *Goal 2*—Determine a methodology of testing the end result to ensure correctness. We will validate against the VKITTI 2.0 dataset using End-Point Error (EPE) metrics

3. *Goal 3*—Create a complete pipeline that takes input depth maps and pose files, feeds them into the system, and outputs optical flow results with statistics (EPE mean, median, std, min, max)

Justification: These goals are achievable because the scope is similar to one of the homeworks. We need to determine the level of parallelism and communication/synchronization between processors. Furthermore, we need to determine the data structure needed that facilitates the parallelism

(such as how we represent images and robot poses). All of this is within the scope of the project that is doable in a similar manner to one of the homeworks.

Hope to Achieve (Stretch Goals)

Extra goals if the project goes really well and you get ahead of schedule:

1. *Stretch Goal 1*—Support 4K resolution (3840×2160) at real-time performance (30+ FPS)
2. *Stretch Goal 2*—Batch processing of multiple frames to improve overall throughput and amortize GPU launch overhead

Performance Goals

- Target speedup: 50-100x speedup vs. optimized CPU implementation
- Target execution time: Real-time performance on HD images (60+ FPS on 1080p, 30+ FPS on 4K)
- Justification: We can achieve this because of the embarrassingly parallel nature of the problem. Each pixel is computed independently with no synchronization overhead between threads. GPU parallelism across thousands of cores will hide memory latency and provide massive speedup over sequential CPU execution.

Platform Choice

GPU programming on CUDA is well-suited for this project because we have massive amounts of data being processed (millions of pixels) with little communication needed between threads. GPU programming is effective because the kernel structure will allow us to tune arithmetic-intensive matrix transformations on the GPU while handling I/O and validation on the CPU. CUDA provides superior debugging and profiling tools compared to alternatives like OpenCL.

Schedule

Week	Dates	Tasks	Assigned To
1	Nov 10-17	• Write up project proposal • Set up development environment • Review reference materials • Design architecture	Steven/Allen
2	Nov 17-24	• Implement core algorithm • Initial baseline version	Steven/Allen
3	Nov 24-Dec 3	• Parallel implementation • Performance profiling	Steven/Allen
4	Dec 4-10	• Optimization and tuning • Poster preparation	Steven/Allen

Key Milestones

- **Nov 13:** Project Proposal Due
- **Nov 27:** Milestone Report Due
- **Dec 9-13:** Poster Session
- **Dec 15:** Final Report Due