15-418/618 Final Project Proposal

GPU Parallelization of the Goldstein Branch-Cut Phase Unwrapping Algorithm

Furi Xiang Anurag Aryal

March 25, 2026

URL

Project web page:

SUMMARY

We propose to implement and evaluate a GPU-accelerated version of the Goldstein branch-cut phase unwrapping algorithm on NVIDIA GPUs using CUDA. Our work will focus on three stages—residue identification, branch-cut placement, and phase integration—and will compare multiple parallelization strategies, including tiled GPU kernels, residue-list-based cut placement, and multi-point BFS-style frontier integration.

BACKGROUND

Phase unwrapping arises in interferometric imaging applications such as InSAR, digital holography, and optical metrology, where measured phase is wrapped in 2π discontinuity. Goldstein's branch-cut algorithm is one of the classical path-following approaches: it first detects *residues*, then places *branch cuts* to connect opposite-polarity residues or connect residues to the image boundary, and finally performs *integration/unwrapping* along paths that avoid those cuts.

At a high level, the algorithm can be summarized as:

1. For each 2×2 cell, compute wrapped circulation and mark positive/negative residues.
2. Connect residues with branch cuts.
3. Starting from a valid seed pixel, propagate unwrapped phase to reachable neighbors without crossing branch cuts.

The first stage is naturally data-parallel because each residue test depends only on a small neighborhood. A GPU implementation can assign one thread per pixel or per cell, use shared-memory tiling to reduce global memory traffic, and compute residues independently.

The second and third stages are more challenging. Branch-cut placement is traditionally greedy and serial because earlier pairings influence later decisions. Integration is naturally a graph traversal over the non-cut phase domain, and a naive flood fill has queue dependence, irregular control flow, and poor SIMD behavior. Recent work suggests two main directions for exposing more parallelism: (1) tile/block-based GPU processing with shared memory, and (2) frontier/BFS-style traversal that processes an entire wavefront in parallel.

The motivation for this project is that Goldstein combines one straightforward stencil-like kernel with two much more irregular stages, making it a good case study for mapping branch-heavy algorithms to GPUs. The project also lets us investigate how different reformulations change the tradeoff between locality, synchronization, and available parallelism.

THE CHALLENGE

This project is challenging because Goldstein's algorithm mixes a regular image-processing stage with two irregular, path-dependent stages.

Workload Characteristics

Residue identification This stage has regular memory access, strong spatial locality, and little synchronization. It is a good fit for tiled CUDA kernels and should scale well.

Branch-cut placement This stage is much less regular. A straightforward greedy implementation introduces serial dependence because once one residue is paired, later pairing choices change. Memory access becomes sparse and data-dependent, and different residues may take very different control paths. We expect branch divergence, contention from atomics, and load imbalance across thread blocks to be major issues.

Integration(unwrapping) Integration is effectively a graph traversal over the valid non-cut pixels. A serial flood fill has queue dependence and poor SIMD utilization. A frontier-based BFS can expose more parallelism, but it also introduces synchronization between levels and irregular frontier sizes. A tiled flood-fill method may improve locality but may also require repeated directional passes and coordination between neighboring tiles.

System Constraints

CUDA GPUs performs SIMD operations, and therefore are most effective for workloads with regular memory access, abundant thread-level parallelism, and limited synchronization. Goldstein violates these assumptions in stages 2 and 3. The main system-level challenges are:

- avoiding warp divergence in residue matching and phase propagation
- reducing atomic and synchronization overhead during sparse residue compaction and frontier expansion
- maintaining coalesced memory access for sparse data structures
- balancing locality-oriented tiling against the need for globally coordinated traversal
- handling large differences in residue density across phase maps

By doing this project, we hope to learn which reformulation gives the best GPU performance for the difficult stages of Goldstein: tile-local processing, residue-list-based matching, or frontier/BFS-style integration.

RESOURCES

We plan to use:

- **CUDA/C++** for the GPU implementation
- **OpenCV** library for image I/O and image pre/post-processing if needed

- CPU serial reference implementation from github repo: https://github.com/williamdelacruz/parallel-Goldstein
- NVIDIA RTX 2080 GPUs from GHC machine

We will use the following references for implementation guidance:

- R. M. Goldstein, H. A. Zebker, and C. L. Werner, "Satellite Radar Interferometry: Two-Dimensional Phase Unwrapping," *Radio Science*, 1988.
- G. López García, S. V. Veleva, and A. C. De La Campa, "A parallel path-following phase unwrapping algorithm based on a top-down breadth-first search approach," *Optik*, 2020.
- Y. Li, S. Han, X. Li, and C. Xu, "Efficient GPU acceleration for phase unwrapping algorithm," in Optical Metrology and Inspection for Industrial Applications X, vol. 12769, SPIE, 2023, Art. no. 1276917, doi: 10.1117/12.2686780.

GOALS AND DELIVERABLES

Plan to Achieve

1. **CPU serial reference implementation.** Implement a complete CPU baseline for Goldstein branch-cut unwrapping:
   - residue detection
   - greedy branch-cut placement
   - serial flood-fill integration
   - correctness testing on synthetic wrapped phase maps.
2. **GPU residue-identification kernel.** Implement a CUDA kernel for stage 1 and compare:
   - naive global-memory implementation
   - shared-memory tiled implementation
3. **GPU branch-cut placement** Implement at least one GPU-friendly strategy:
   - residue compaction into positive/negative arrays,
   - residue matching on GPU,
   - branch-cut construction.

   We will likely compare at least two variants, such as a residue-list baseline and a spatial tiling/binning approach.
4. **GPU integration** Implement and compare two integration strategies:
   - tiled flood fill / wavefront propagation,
   - BFS-style or frontier-based integration.
5. **Experimental evaluation.** Measure:
   - RMS of unwrapped image between CPU baseline and GPU implementation
   - stage-by-stage runtime
   - end-to-end speedup over CPU
   - scaling with image size
   - scaling with residue density

We aim to achieve matching speedup performance reported in the reference GPU phase unwrapping paper, which claims 61x maximum speedup for CPU: AMD Ryzen 5900H, and GPU: NVIDIA RTX 3070-laptop

Hope to Achieve

If the project goes especially well, we would also like to:

- implement multi-source BFS or connected-component-based integration after cuts
- explore spatial binning to reduce branch-cut pairing search cost
- implement batched phase unwrapping to improve throughput
- achieve better performance than reference GPU phase unwrapping paper

If Progress Is Slower Than Expected

If stages 3 or 4 proves to be under-performing, we will reduce scope while still preserving a meaningful project by:

- fully implementing stage 2 on GPU
- offload one of stage 3 or 4 to CPU to create a CPU-GPU hybrid phase-unwrapping algorithm

PLATFORM CHOICE

We chose NVIDIA GPU and CUDA because at least part of Goldstein is clearly data-parallel, and the project's main interest lies in how to map the irregular stages onto SIMD hardware. The residue detection stage is a pixel-wise parallel image kernel with strong locality, and the use of GPU shared memeory can not only accelerate this stage of computation, but it can also accelerate branch-cut placement and integration stage if we can refactor them into tiled neighborhood computation. The branch-cut and integration operation has poor compatibility with SIMD in its naive implementation, but it would be a good case study on how irregular branching behavior could be reformulated in a SIMD friendly form. These reasons makes CUDA a good platform not only for acceleration but also for learning. The project is not just about speeding up a dense numerical kernel; it is about understanding how branch divergence, sparse matching, atomics, and frontier expansion behave on a real parallel system.

SCHEDULE

We plan to update this schedule each week based on actual progress. Our current schedule is:

Week 1: Read and summarize key references. Evaluate and re-implement CPU serial baseline for Goldstein branch cut based on the reference github repository. Build synthetic wrapped phase test cases and find open-sourced test images.

Week 2: Complete implement naive CUDA residue detection and shared-memory tiled kernel. Implement residue compaction into positive/negative arrays and residual matching.

Week 3 (Project Milestone Report): Implement GPU integration with tiled flood fill or tiled multi-point wavefront propagation. Complete milestone report.

Week 4: Finalize integration kernel implementation. Continue to tune all 3 stages to reduce divergence and improve memory locality. Benchmark the code to record progress.

Week 5 (Final): Complete final benchmark, generate plots, write the report, and prepare poster/demo materials.