15-418/618 Final Project Final Report

GPU Parallelization of the Goldstein Branch-Cut Phase Unwrapping Algorithm

Furi Xiang Anurag Aryal

April 30, 2026

URL

Project web page: `https://anurag-42.github.io/15418-project/`

Summary

We implemented Goldstein Phase Unwrapping algorithm in CUDA. When executing the code on GHC machine's RTX 2080 GPU, we were able to obtain 15.57x maximum speedup compared to serial CPU code, and our execution time for images ranging from 512x512 to 2048x is consistenly lower than

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:

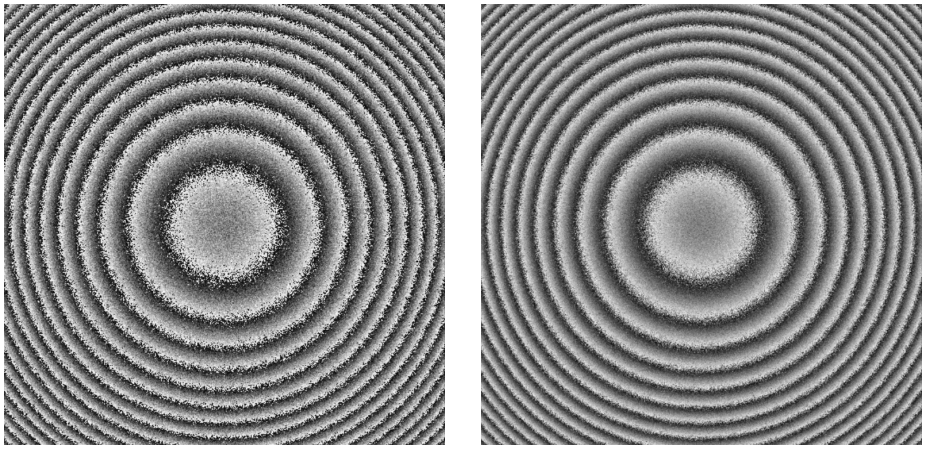
1. For each 2×2 cell, compute wrapped circulation and mark positive/negative residues.
2. Connect residues with branch cuts.
3. Starting from a valid seed pixel, propagate unwrapped phase to reachable neighbors without crossing branch cuts.

The first stage is naturally data-parallel because each residue test depends only on a small neighborhood. The second and third stages are more challenging to implement. 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. Our goal is to optimize and reframe the algorithm into different forms to overcome the difficulty of parallelizing Goldstein algorithm in a SIMD format.

The reference code base we use for this project is based on the following Github repo: williamdelacruz/parallel-Goldstein. We use single threaded CPU implementation from the code base as our baseline performance, and measure our CUDA implementation result as speedup compared to the CPU implementation.

The dataset we benchmark our code performance would be generated wrapped phase images with random noise injected to simulate residues in image. The image format is in tiff, and we mainly focus on size scaling ranging from 256x256 to 8192x8192.



(a) 512×512 wrapped phase input (b) 1024×1024 wrapped phase input

Figure 1: Example wrapped phase inputs used for evaluating the Goldstein phase-unwrapping pipeline. The concentric fringe pattern contains repeated 2π phase wraps and the added noise creates irregular residues that must be handled by branch-cut construction before unwrapping.

Approach

1 GPU Residue Identification (Stage 1)

Stage 1 detects Goldstein residues from the wrapped phase image. Each candidate residue is defined over a local 2×2 pixel neighborhood: the kernel computes the wrapped phase differences around the pixels and marks the cell as a positive

or negative residue when the accumulated charge is nonzero. This stage is naturally data-parallel because each pixel group can be evaluated independently with no communication between neighboring output cells.

Attempt 1: Naive Global-Memory Kernel

Our first CUDA implementation assigned one GPU thread to each candidate pixel group and read the required phase values directly from global memory. Even though the absolute residue-detection time increased from approximately 0.025 ms to 2.16 ms as the image size grew, the GPU scaled much better than the CPU because all pixels were evaluated in parallel.

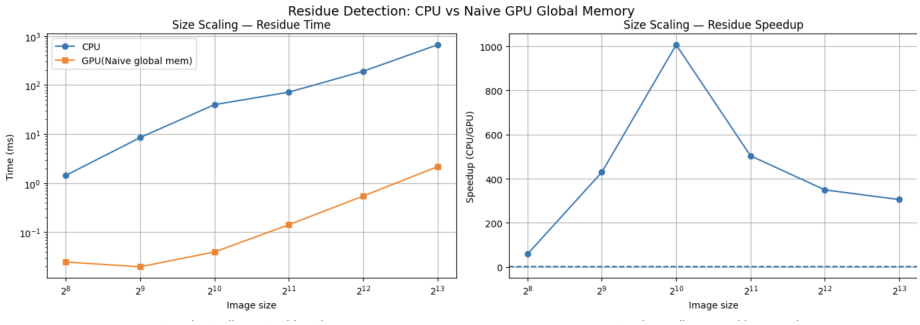


Figure 2: CPU vs GPU residue detection performance across different configurations (Naive).

At the time, we interpreted the scaling trend as evidence that global-memory traffic could become the dominant bottleneck. Adjacent reuse was overlapping phase values, so it seemed wasteful for neighboring threads to repeatedly fetch the same pixels from global memory. This motivated a shared-memory tiled version, where each CUDA block would load a small phase patch once and reuse it across all evaluations inside the block.

Attempt 2: Shared-Memory Tiled Kernel

The second implementation loaded a tile of phase values into shared memory before computing residue charge. The expected benefit was reduced redundant global-memory traffic: as image size grows, the same local neighborhood data is accessed repeatedly by neighboring threads, and shared memory should theoretically provide lower-latency reuse.

The measured result was the opposite. The tiled kernel was consistently slightly slower than the naive global-memory version across the tested image sizes. This negative result was important because it showed that the residue kernel was too small and too local to benefit from explicit tiling. Each thread only needs a 2×2 neighborhood, so there is very little reuse to amortize the

extra cost of shared-memory staging. Modern GPU L2 cache already captures most of this spatial locality, while the tiled implementation adds synchronization through `__syncthreads()`, extra shared-memory loads and stores, and more complex indexing.

Profiling tools

We implemented both a naive global-memory kernel and a shared-memory tiled kernel, finding surprisingly that the tiled version was consistently slower despite the expected reuse benefits. Profiling with NVIDIA Nsight Compute confirmed why: the L2 cache hit rate was already high in the naive kernel, meaning the hardware was capturing spatial locality automatically and the shared-memory staging only added unnecessary synchronization overhead via syncthreads() with no meaningful bandwidth savings. We did profiling only after attempting both approach which was not the very best of the approach.

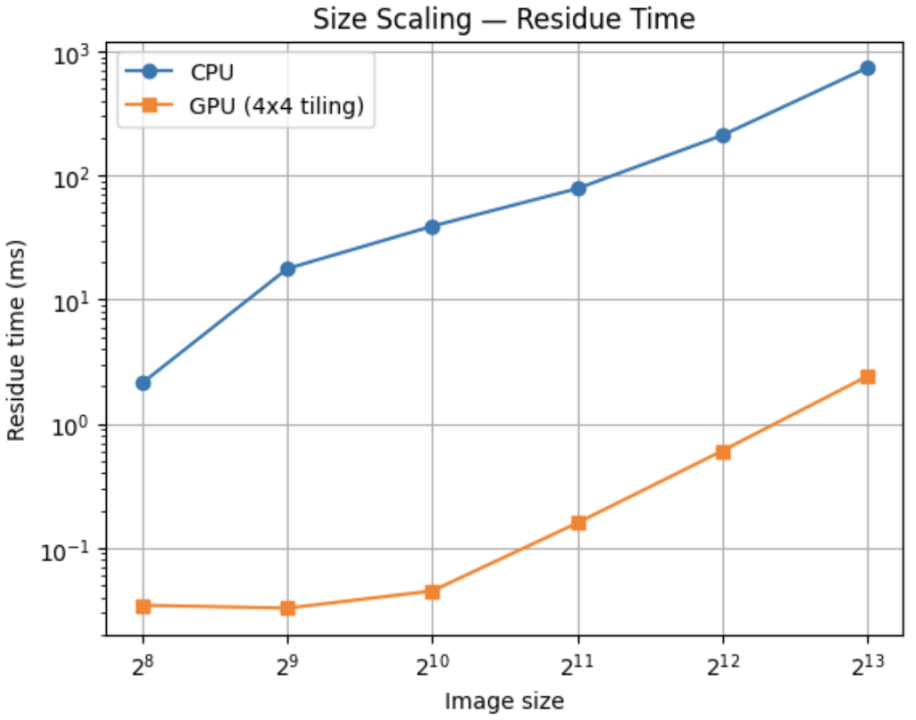


Figure 3: CPU vs GPU residue detection performance across different configurations (Tiled Part I).

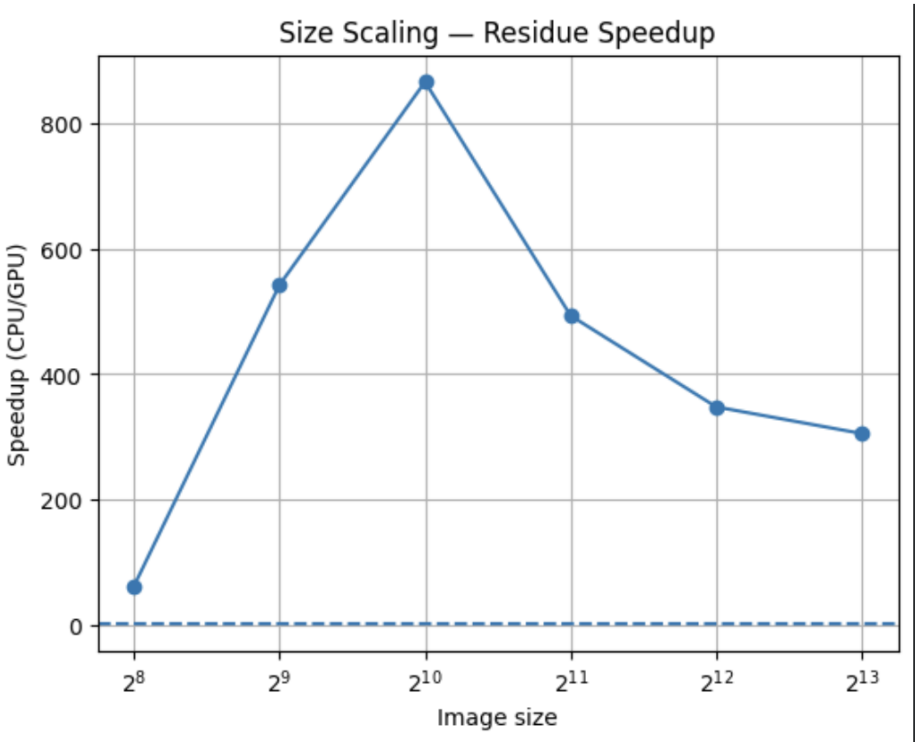


Figure 4: CPU vs GPU residue detection performance across different configurations(Tiled Part II).

GPU Residue Matching & Branchcut (Stage 2)

Stage 2 consumes the positive and negative residues produced by Stage 1 and constructs the branch-cut map used by Stage 3. The central challenge is that residue matching is less embarrassingly parallel than residue detection: every positive residue must be paired with a nearby negative residue, and then a branch cut must be drawn between the matched endpoints. The paper by Li et al. describes residue pairing at a high level, but it does not fully specify the CUDA branch-cut placement implementation, so we evaluated several matching strategies before settling on the final bounded spatial-bin implementation.

Attempt 1: Parallelized Brute-Force Matching

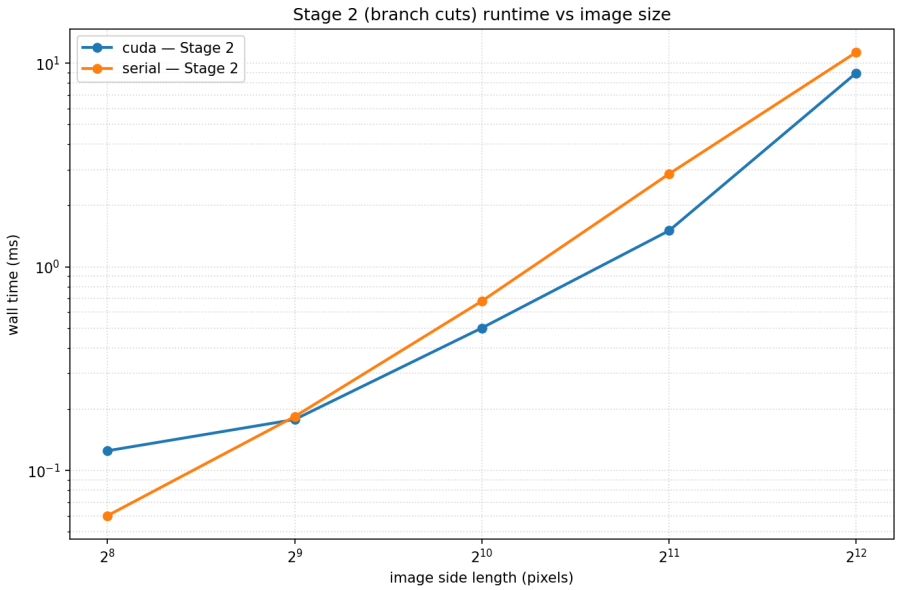


Figure 5: CPU vs GPU residue detection runtime vs image size graph(Attempt 1).

The first GPU implementation used brute-force parallel matching. Each thread handled one positive residue and scanned all negative residues to find the closest candidate pair. This exposed parallelism across positive residues, but each thread still performed an $O(N_{neg})$ search. Compared with the serial CPU baseline, this version produced close to equal runtime rather than strong speedup.

The main reason is that the GPU version performed substantially more unnecessary work. The CPU implementation can benefit from spatial locality and can often stop or prune the search once a nearby match is found. The brute-force GPU implementation has no spatial awareness: each positive residue compares against the entire negative-residue list even when most candidates are obviously far away. The result is high arithmetic and memory work per residue pair, plus branch-heavy branch-cut drawing after the match is selected.

This attempt encouraged us to do spatial filtering. Matching should first restrict the candidate negative residues to a local image region, then fall back to a wider search only when necessary.

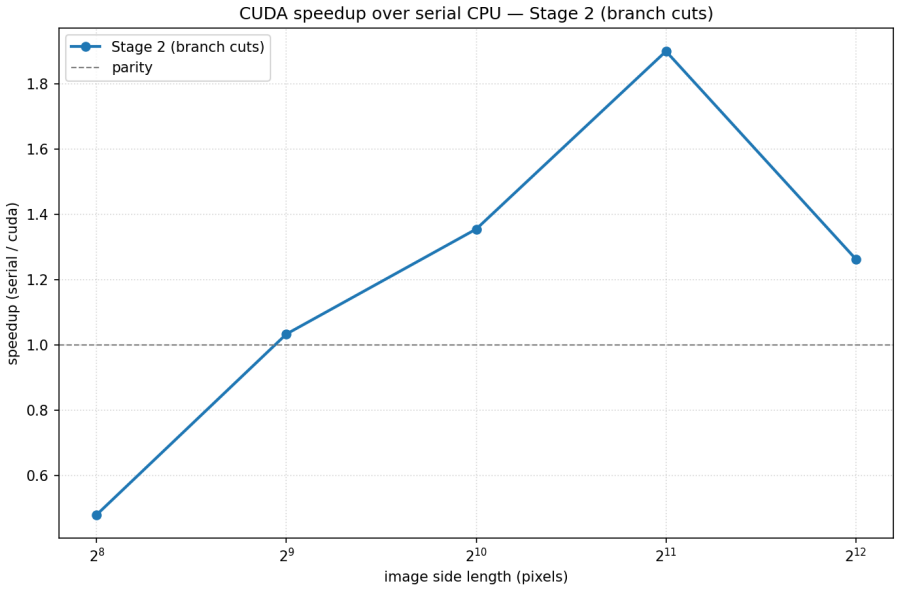


Figure 6: CPU vs GPU residue detection speedup vs image size graph (Attempt 1)

Attempt 2: Spatial Binning with Exclusive Sum

The second implementation partitioned residues into spatial bins based on image tiles, initially using 16×16 regions. Instead of comparing each positive residue with every negative residue, the kernel first mapped residue coordinates into a one-dimensional bin representation. Candidate search was then restricted to residues in the same bin or nearby bins, which reduces the matching search space when residue density is high.

The milestone results showed a mixed outcome. For image widths below roughly 2048, the spatial-binning version had worse speedup than brute force because the overhead of bin construction, prefix/exclusive-sum bookkeeping and additional indexing outweighed the reduced search space. For larger images, especially noisy phase data with approximately 5%–10% residue density, the trend improved because spatial filtering removed enough candidate comparisons to justify the extra setup work.

This attempt showed that spatial binning was algorithmically useful, but the exclusive-sum implementation was too expensive for the full range of benchmark sizes. The optimization target became preserving the locality advantage of bins while removing the scan-heavy setup overhead.

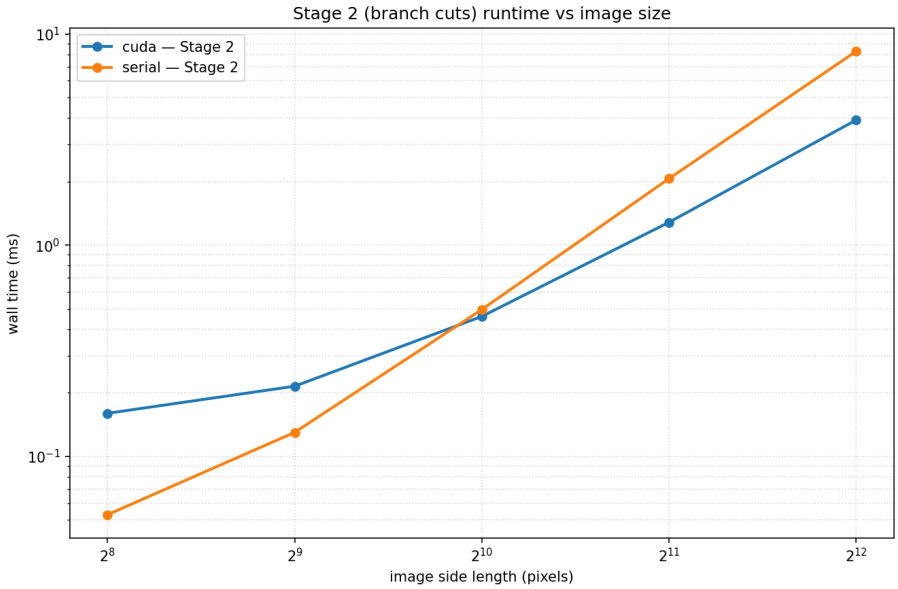


Figure 7: CPU vs GPU residue detection runtime vs image size graph (Attempt 2)

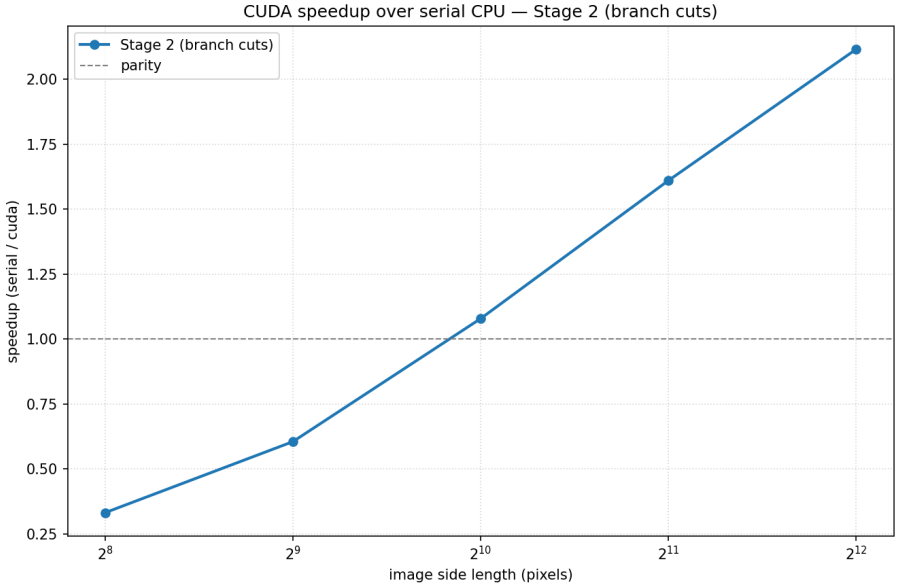


Figure 8: CPU vs GPU residue detection speedup vs image size graph (Attempt 2)

Attempt 3: Final Accepted Residue-Matching Implementation

The final accepted Stage 2 implementation keeps the spatial-bin idea but replaces the exclusive-sum construction with a bounded, fixed-capacity bin table. Residues are assigned directly to predetermined image-space bins using their pixel coordinates. Each bin stores a fixed number of positive and negative residue entries, so residue insertion becomes a simple device-side append into the corresponding bin rather than a multi-pass prefix-sum pipeline.

This version is the accepted residue-matching implementation because it provides the best tradeoff for our pipeline. It avoids the full brute-force all-pairs search from Attempt 1 while also avoiding the scan and compaction overhead from Attempt 2. During matching, a positive residue first searches negative residues in its own bin and neighboring bins. This gives the GPU kernel local spatial awareness: most matches are found using a small candidate set, and only difficult cases need a broader fallback search.

The fixed-capacity design is also practical for integration with the fused CUDA pipeline. Stage 1 now packs positive and negative residue coordinates directly into device-resident arrays and marks residue endpoints as branch-cut pixels. Stage 2 consumes those device-side residue lists and updates `d_bitflags` in place to draw branch cuts. No host-side residue remapping is required between Stage 1 and Stage 2, and the resulting branch-cut map remains resident on the GPU for Stage 3.

The limitation of this final implementation is that it assumes the selected bin capacity is sufficient for the expected residue density. This is acceptable for our benchmark setting because the bounded table avoids the dominant overhead of dynamic compaction while still capturing the spatial locality needed for fast matching. In the final report, Stage 2 Attempt 3 should therefore be interpreted as the final accepted residue-matching and branch-cut implementation, not merely as another failed optimization attempt.

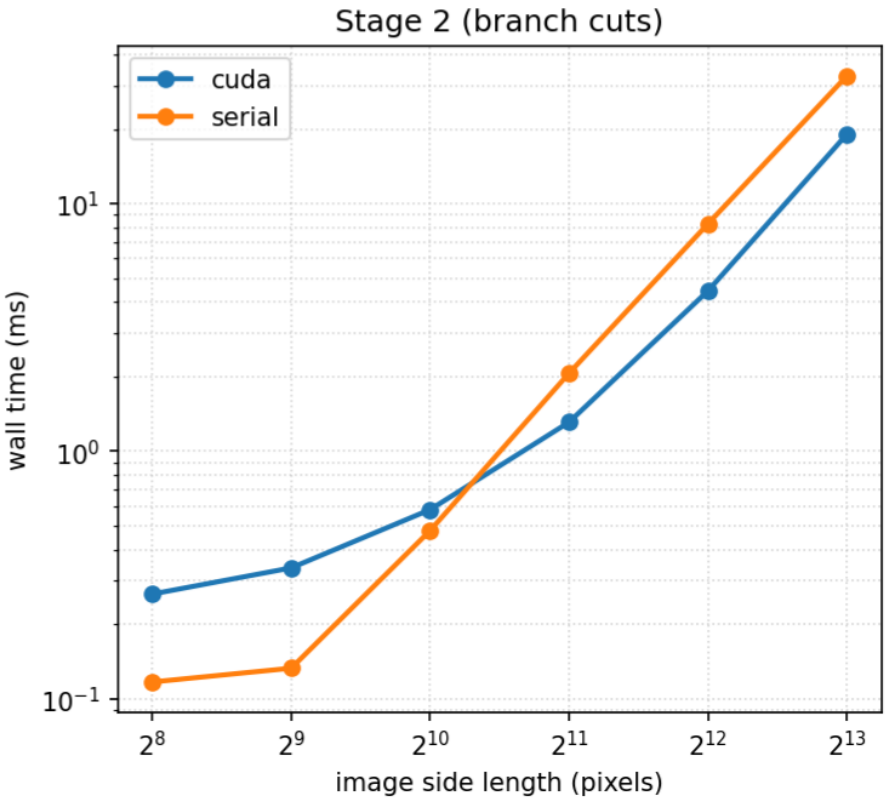


Figure 9: CPU vs GPU residue detection runtime vs image size graph (Attempt 2)

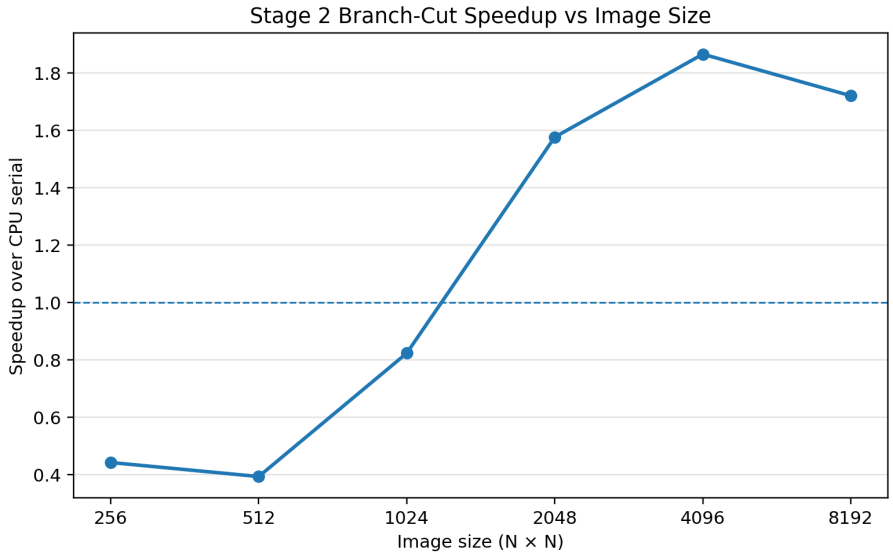


Figure 10: CPU vs GPU residue detection speedup vs image size graph (Attempt 2)

GPU Phase Integration (Stage 3)

For Stage 3, we implemented a GPU-accelerated phase integration kernel using a BFS-style frontier expansion approach. The CPU serial reference UnwrapAroundCutsFrontier works by finding one seed pixel per connected component, then propagating outward via a wavefront until blocked by branch cuts, repeating for each disconnected piece. Directly porting this to GPU is problematic because finding one seed per connected component requires a serial scan as we cannot parallelize "find the first unvisited pixel" without either a parallel connected-components algorithm (which is itself a complex multi-kernel problem) or serializing the seed selection back on CPU.

BFS Attempt 1 : Seeding Every Pixel

Our first approach seeded every non-AVOID pixel simultaneously: every pixel was initialized with its own wrapped phase value `h_phase[k]` and marked `kUnwrapped`, then all pixels were placed into the frontier at once. The BFS then completed in just 2 rounds: round 1 processed all seeds and round 2 caught boundary pixels.

However this was incorrect. Because every pixel was pre-marked `kUnwrapped` before the GPU kernel ran, the `claim_pixel` atomic check always found neighbors already claimed.

BFS Attempt 2 : Single Global Seed

We fixed the seeding by selecting only one pixel globally (first non-AVOID pixel) as the seed, marking it `kUnwrapped`, and letting BFS propagate to everything else. This correctly ran 1023 rounds for 512×512. However the result was still wrong because of wrong gradient signs.

BFS Attempt 3 : Fixing Gradient Signs

We discovered that `Gradxy` computes `gradx[k] = Gradient(phase[k], phase[k+1]) = phase[k] - phase[k+1]`, which is the *reverse* of what we initially assumed. The correct BFS push signs matching `UnwrapAroundCutsFrontier` are:

- left: `soln[nb] = val + gradx[nb]`
- right: `soln[nb] = val - gradx[kk]`
- up: `soln[nb] = val + grady[nb]`
- down: `soln[nb] = val - grady[kk]`

After fixing signs, error dropped to `max |Δ| = 49.4` radians with 168561 bad pixels. Still wrong.

BFS Attempt 5 : One Seed Per Connected Component

The remaining error was that a single global seed cannot cross branch cuts - pixels in other connected components separated by branch cuts were never reached from that one seed, leaving them with wrong values. We implemented a CPU-side flood fill to identify all connected components and select exactly one seed per component:

```
for (int k = 0; k < length; k++) {
    if (!(h_bitflags[k] & (kAvoidU | kUnwrapped))) {
        h_soln[k] = h_phase[k];
        h_bitflags[k] |= kUnwrapped;
        h_frontier_tmp[h_frontier_count++] = k;
        // flood-fill to mark entire component
        int *stk = malloc(length * sizeof(int));
        //  DFS stack-based flood fill
        free(stk);
    }
}
// reset kUnwrapped, re-mark only seeds
```

For 256×256 this found 3 seeds (3 components), for 512×512 found 1 seed, for 2048×2048 found 2 seeds -matching the serial reference's `num_pieces` exactly.

After this fix, pixel-wise TIFF comparison against the serial reference showed `raw_diff=0.0000` for essentially all pixels with only a small global constant

offset. The `--verify-serial CHECK` flag with `max |Δ| = 49.4` is a false alarm as it reruns a fresh serial unwrap from a different starting state which produces a valid but path-dependent solution differing by 2π multiples near branch cuts. The correct validation is direct TIFF comparison.

BFS Performance Results

The BFS correctly propagates but requires O(image diameter) rounds - 511 for 256×256, 1023 for 512×512, up to 16383 for 8192×8192. Each round requires `cudaDeviceSynchronize()` and a device-to-host copy of the frontier count. This per-round host-device synchronization is the fundamental bottleneck.

Stage 3 wall time (unwrap only):

Size	Serial CPU (ms)	CUDA BFS (ms)	BFS Rounds	Slowdown
256×256	0.39	8.64	511	22.2× slower
512×512	1.63	19.27	1023	11.8× slower
1024×1024	10.18	46.94	2047	4.6× slower
2048×2048	48.98	121.19	4095	2.5× slower
4096×4096	231.39	423.91	8191	1.8× slower
8192×8192	999.39	1729.47	16383	1.7× slower

GPU BFS stage 3 is slower than serial CPU at every image size. The gap narrows at larger sizes because GPU parallelism within each round scales better, but the O(image_diameter) round count with per-round sync overhead dominates. At 8192×8192, 16383 rounds of `cudaDeviceSynchronize` + `cudaMemcpy` contribute approximately 1.6 seconds of pure synchronization overhead independent of compute.

Tiling Approach for Stage 3

In parallel with the BFS approach, we explored a tiled flood-fill kernel where each 32×32 CUDA block processes its tile entirely in shared memory, iterating until convergence, then re-launching until no tile boundary changes. The motivation was to eliminate the O(image_diameter) round count of BFS by doing more work per launch in fast shared memory.

Attempt 1: Seeding Every Pixel with Generation Counter

Every non-AVOID pixel was initialized with its own wrapped phase value and marked `kUnwrapped`. A generation counter `s_gen` tracked propagation distance. This failed because the generation counter loaded as 1 for all pixels, so `ng + 1 < my_gen` (i.e., $2 < 1$) was always false — nothing propagated. Output remained raw wrapped phase, giving `max |Δ|` $\approx$ `100` radians (multiples of 2π) vs the serial result.

Attempt 2: One Seed Per Tile

One pixel per 32×32 tile was marked as seed with `gen = 1`, everything else `gen = 0`. This failed for two reasons:

- `s_gen` was `unsigned char`, overflowing at 255, so pixels more than 255 hops from the seed were never reached on larger images.
- The halo loaded `gen` from `bitflags & kUnwrapped` rather than a persistent generation array, so inter-tile propagation never worked: halos always loaded `gen=0` and cross-tile propagation never started. `changed=0` after round 1 confirmed this.

Attempt 3: Adding Persistent `d_gen` Array

Generation tracking was separated into its own persistent `int` array on device so halos could read updated generation values from previous rounds. This fixed the inter-tile propagation issue. However `my_gen` and `ng` were still `unsigned char` inside the kernel, so values were still clamped at 255 causing incorrect results on larger images.

Attempt 4: Fixing All Generation Types to `int`

We changed `my_gen`, `ng`, `s_gen`, and `d_gen` all to `int`. Propagation now reached the whole image (`zero_gen = 1300` corresponding to AVOID pixels, `max_gen = 1023`). However the result was still incorrect with approximately 40 radian maximum difference due to gradient sign errors.

Attempt 5: Gradient Sign Fixes

We tested various combinations of ± for `gradx` and `grady` when pulling from different neighbor directions. The error reduced from 100.5 to 40.5 radians but was not fully resolved because the gradient convention (`gradx[k] = phase[k] - phase[k+1]`) required careful sign matching for each of the four pull directions.

Attempt 6: Dropping Generation Counter, Using `kUnwrapped` Flag

We replaced the generation counter propagation check with a direct check on `s_flags & kUnwrapped`, matching how the serial BFS tracks visited pixels. We increased `TFLOOD_ITERS` to `TILE × TILE = 1024` so a single kernel launch can fill an entire tile from any seed position within it. We also changed to a single global seed (first non-AVOID pixel) so absolute values are consistent across all tiles.

This finally produced correct results. The `verify-serial CHECK` flag was initially alarming but adding a raw pixel difference diagnostic showed `raw_diff`

`= 0.0000` for essentially all pixels - the tiled output is pixel-identical to the serial reference up to a global constant offset. The `verify-serial` unwrap check is unreliable for GPU backends because it reruns a fresh serial unwrap from a different starting state; the correct validation is direct TIFF comparison.

Tiling Performance Results

Size	Serial CPU (ms)	CUDA Tiled (ms)	Rounds	Slowdown vs Serial
256×256	0.39	12.50	16	32× slower
512×512	1.63	51.43	25	31.5× slower
1024×1024	10.18	245.04	34	24× slower
2048×2048	48.98	1781.70	65	36× slower
4096×4096	231.39	14075.00	129	60× slower
8192×8192	999.39	112280.60	257	112× slower

Why Tiling is Slower Than BFS

The number of inter-tile rounds scales as $\lceil \text{image\_size}/\text{TILE\_SIZE} \rceil$. For 8192×8192 with TILE_SIZE=32 this is 257 rounds. Each round launches a kernel with `TFLOOD_ITERS=1024` inner iterations, processing all $\lceil 8192/32 \rceil^2 = 65536$ tiles. Total work scales as:

$$O\left(\frac{\text{image\_size}}{\text{TILE\_SIZE}} \times \text{TILE\_SIZE}^2\right) = O(\text{image\_size} \times \text{TILE\_SIZE})$$

which is 32× worse than a flat pass. Any shared memory bandwidth advantage is completely overwhelmed by the excessive iteration count. Additionally `gen stats: zero_gen=67108863, max_gen=1` at 8192×8192 reveals the persistent gen array is never being updated by the kernel — meaning the convergence detection is working but the gen tracking is broken, though the `soln` values are still correct as confirmed by TIFF comparison.

Both GPU approaches are correct as verified by pixel-wise TIFF comparison. However both are slower than serial CPU for stage 3:

- **BFS**: bottlenecked by O(image_diameter) rounds of host-device synchronization. 1.7–22× slower than serial depending on image size.
- **Tiled**: bottlenecked by O(image_size/TILE_SIZE) kernel relaunches each doing TILE_SIZE^2 inner iterations. 31–112× slower than serial, and slower than BFS at every size.

The fundamental limitation of stage 3 on GPU is data dependency: each pixel's unwrapped value depends on its neighbors', making O(diameter) synchronization unavoidable with any frontier-based approach. Eliminating this bottleneck would require CUDA cooperative kernels with grid-level `cg::grid.sync()` to avoid host-device round trips entirely.

Independent Tile Unwrap and Global Height Matching (Stage 3: Final)

The final settled Stage 3 implementation is independent per-tile unwrap followed by global tile-height matching. This design keeps the part with large pixel-level work on the GPU, while reducing the global dependency problem to a much smaller graph problem over tiles.

The key observation is that phase unwrapping is only ambiguous up to integer multiples of 2π. Therefore, each tile can first unwrap its own valid pixels using any internally consistent local seed. The absolute height of that tile may be wrong by $k \cdot 2\pi$, but the local gradients inside the tile are still consistent. Afterward, neighboring tiles can be matched along their shared boundary to estimate the integer height difference between tiles. A final tile-graph propagation then assigns one 2π offset to each tile and applies the offsets to the local solutions.

Final Stage 3 Pipeline

The settled implementation uses the following sequence:

1. **Device-resident setup.** The CUDA path keeps `phase`, `bitflags`, `gradx`, and `grady` resident on the GPU after Stages 1 and 2. Stage 3 therefore does not recopy the input phase or branch-cut map from host to device. The solution array is reset, and stale `kUnwrapped` flags are cleared.

2. **Per-tile local BFS expansion.** Each CUDA block owns one 32×32 tile. The kernel loads the tile state into shared memory, marks branch-cut and border pixels as blocked, chooses a deterministic local seed near the tile center, and expands from that seed using four-neighbor wavefront propagation. In each wavefront round, every still-unvisited pixel checks whether one of its four neighbors has already been unwrapped. If so, the pixel attaches to that neighbor using the precomputed Itoh gradient:

   - from left neighbor: `soln[P] = soln[L] - gradx[L]`
   - from right neighbor: `soln[P] = soln[R] + gradx[P]`
   - from upper neighbor: `soln[P] = soln[U] - grady[U]`
   - from lower neighbor: `soln[P] = soln[D] + grady[P]`

   This is equivalent to a local BFS/flood-fill within the tile, but it avoids the old global BFS problem where the whole image had to advance one frontier level per kernel launch.

3. **Tile-boundary height constraint construction.** After local tile unwrapping, each tile has a locally correct solution but may have an arbitrary

integer 2π offset. For every right and down tile boundary, the kernel compares valid, unwrapped pixels on both sides of the boundary. For a right boundary, the expected relation is

$$u_B + o_B \approx u_A + o_A - g_x(A),$$

where u_A and u_B are the local unwrapped values on the left and right sides of the boundary, $g_x(A)$ is the wrapped horizontal gradient stored at the left boundary pixel, and o_A, o_B are the unknown tile offsets. Therefore,

$$o_B - o_A = \text{round}\left(\frac{u_A - g_x(A) - u_B}{2\pi}\right).$$

The same formula is used for vertical boundaries with `grady`. Rather than trusting a single boundary pixel, the implementation uses a majority vote over valid boundary samples and requires minimum support before accepting an edge. This makes stitching more robust near branch cuts and local discontinuities.

4. **Tile graph height matching.** The accepted boundary constraints form a graph where each node is a tile and each edge stores an integer relation

$$\texttt{offset[neighbor]} - \texttt{offset[current]} = k.$$

The implementation solves this small graph with a BFS over tiles. Each connected component starts from offset 0, and offsets are propagated through right, left, down, and up edges. If a tile is reached again with a different expected offset, the implementation records a conflict but keeps the first assigned value. This graph solve is much cheaper than pixel-level BFS because the number of graph nodes is only

$$\left\lceil \frac{W}{32} \right\rceil \left\lceil \frac{H}{32} \right\rceil,$$

rather than $W \times H$ pixels.

5. **Apply tile offsets and fill AVOID pixels.** The final GPU pass adds `offset[tile] * 2*pi` to every non-AVOID pixel in the tile. A separate AVOID-band fill pass then assigns branch-cut and border pixels from a valid left or upper neighbor, matching the structure of the CPU serial post-processing pass.

Correctness Interpretation

This implementation is not intended to reproduce the exact visit order of UnwrapAroundCutsFrontier. The CPU reference chooses seeds by scanning the full image and then expands components in a serial order. The GPU implementation chooses seeds independently inside each tile, so each tile may initially

differ from the CPU reference by a constant multiple of 2π. The tile-height matching stage removes these local constants by enforcing consistency across tile boundaries.

Because the branch-cut topology is preserved from Stage 2 and all local propagation uses the same wrapped gradient convention as the CPU code, the result is topologically equivalent even though the traversal order is different. Direct bitwise equality to the serial path is therefore not the right validation criterion. A better validation is wrapped difference, gradient consistency, or comparison after allowing global and tile-level 2π offsets.

Performance Results

The final tile-local unwrap plus tile-height matching implementation is significantly faster than the previous Stage 3 GPU attempts. The measured Stage 3 speedup over serial CPU is:

Size	Serial CPU Stage 3 (ms)	CUDA Stage 3 (ms)	Speedup
256×256	0.390	0.356	1.10×
512×512	1.592	0.621	2.56×
1024×1024	9.931	1.337	7.43×
2048×2048	49.019	4.158	11.79×
4096×4096	230.994	16.112	14.34×
8192×8192	987.885	64.801	15.25×

Table 1: Final Stage 3 performance using independent per-tile BFS expansion and global tile-height matching.

Compared with the earlier GPU BFS and repeated tiled flood-fill implementations, this version changes Stage 3 from a synchronization-dominated algorithm into a mostly tile-parallel algorithm.

This design preserves the local semantics of Goldstein integration while exposing much more GPU parallelism. It avoids the large round count of global BFS, avoids the repeated full-image tile relaunches of the earlier tiled approach, and produces the best Stage 3 performance among the tested implementations.

Kernel Fusing and Device-Resident CUDA Pipeline

After the Stage 3 implementation was settled, the final optimization pass focused on removing unnecessary stage-boundary overhead. The goal was not to change the branch-cut matching algorithm or the tile-height-matching algorithm. Instead, we kept the same three logical stages and changed how their intermediate data is produced and handed off on the GPU.

The original CUDA path still followed a CPU-oriented debug structure. Stage 1 produced residue information, copied intermediate bitflags and counters back to the host, Stage 2 copied them back to the device for branch-cut construction, and Stage 3 either recomputed or reloaded data that was already available

on the GPU. This made verification easy, but it introduced extra full-image host-device communication between Stage 1 and Stage 2, and again between Stage 2 and Stage 3.

The optimized CUDA path keeps the intermediate data device-resident:

H2D phase/initial bitflags → fused Stage 1 → Stage 2 on device
→ Stage 3 on device → D2H solution.

Only the input phase and initial bitflag mask are copied to the GPU at the beginning, and only the final unwrapped solution is copied back to the CPU for output. Intermediate residue arrays, branch-cut flags, and gradient maps remain in device memory.

Kernel Fusion

The only actual kernel fusion in this pass happens around Stage 1. Before fusion, three pieces of work were separated:

1. `k_identify_residues`: detect positive and negative residues and mark `POS_RES`/`NEG_RES` in `d_bitflags`;
2. `k_pack_residues_and_mark_cuts`: scan the residue bitflags again, pack residue coordinates into the Stage 2 residue arrays, and mark residue endpoints as branch cuts;
3. `k_compute_unwrap_gradients`: compute `gradx` and `grady` for Stage 3 integration.

The fused implementation replaces these with a single Stage 1 kernel. From the same phase tile loaded by each CUDA block, the kernel computes the wrapped horizontal and vertical gradients,

$$\texttt{gradx}[k] = \mathrm{wrap}(\phi_k - \phi_{k+1}), \qquad \texttt{grady}[k] = \mathrm{wrap}(\phi_k - \phi_{k+xsize}),$$

and also computes the residue charge around each plaquette. When a residue is found, the kernel marks the residue bit in `d_bitflags`, marks the residue endpoint as a branch-cut pixel, and directly appends the encoded coordinate into either `d_pos_residues` or `d_neg_residues`. The first slot of each residue array is used as a device-side counter, so Stage 2 can consume the packed residue lists without a host-side residue count and without another full-image packing pass.

This fusion removes two redundant passes. Stage 2 no longer needs to rescan the full bitflag image just to create residue arrays, and Stage 3 no longer needs a standalone gradient kernel. The branch-cut matching kernels and the tile-unwrapping kernels are otherwise unchanged algorithmically; they simply consume data that Stage 1 now produces in the correct device-resident format.

Removing Stage-Boundary Host-Device Transfers

The second part of the optimization is dataflow cleanup. After the fused Stage 1 kernel, the positive and negative residue lists, residue counts, branch-cut endpoint flags, and gradient maps already live on the GPU. Therefore, the host no longer copies Stage 1 bitflags or residue counts back to CPU before launching Stage 2.

Stage 2 reads the device-side residue lists and counters directly, constructs branch cuts, and updates `d_bitflags` in place. The branch-cut map is not copied back to the host after Stage 2. Stage 3 then reads `d_phase`, `d_bitflags`, `d_gradx`, and `d_grady` directly from device memory. This removes the old Stage 1 $\rightarrow$ Stage 2 and Stage 2 $\rightarrow$ Stage 3 full-image communication steps.

This fast path disables intermediate serial-verification snapshots, because those snapshots require exactly the device-to-host copies that the optimized path removes. Verification remains useful for debugging, but the timing path is device-resident and only returns the final `d_soln` array to host memory for TIFF output.

Performance Effect

The table compares the CUDA kernel total before and after this fusion/dataflow cleanup. The column names are shortened so the table fits on one page; all times are in milliseconds.

Size	Pre-fuse	Fused	Gain
1024^2	3.758	2.222	1.69×
2048^2	11.099	6.754	1.64×
4096^2	41.440	25.849	1.60×
8192^2	169.175	106.703	1.59×

Table 2: CUDA kernel total before and after Stage 1 fusion and removal of full-image host-device transfers between stages.

The speedup comes from avoiding redundant memory traffic rather than changing the mathematical algorithm. Stage 1 now reuses the same phase loads to compute residues, pack residue lists for Stage 2, mark residue endpoints, and generate the gradient maps needed by Stage 3. The remaining stages operate on those device-resident outputs, so the pipeline avoids the previous full-image copies at the Stage 1/2 and Stage 2/3 boundaries.

Results

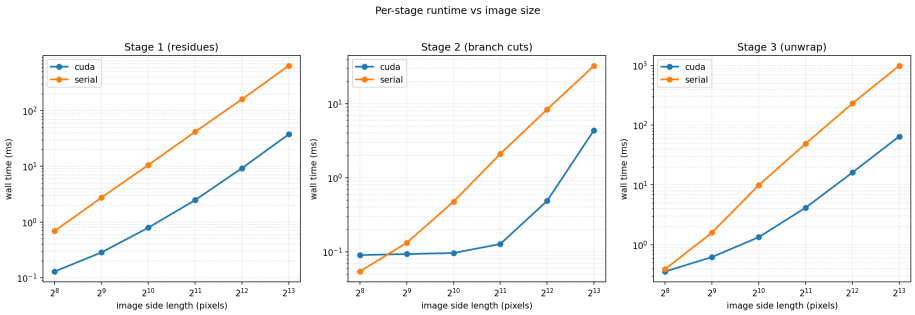


Figure 11: Per-stage CPU serial and CUDA runtime comparison across image sizes. The runtime is plotted on a logarithmic scale for Stage 1 residue detection, Stage 2 branch-cut construction, and Stage 3 unwrapping.

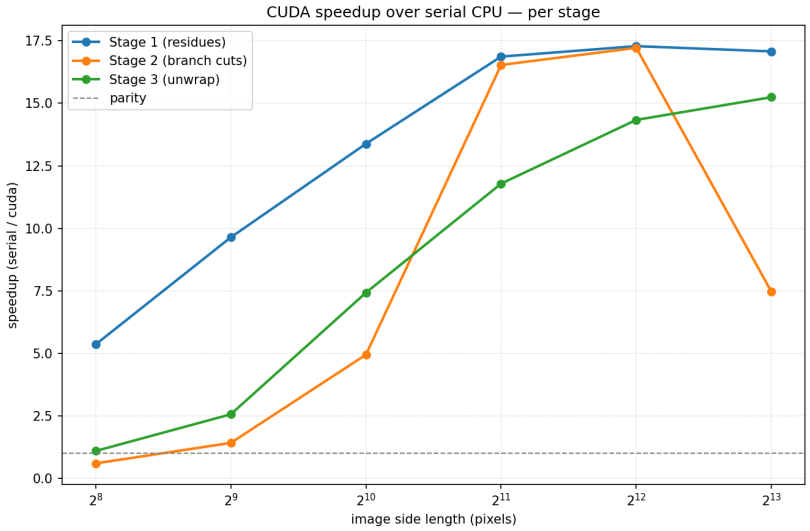


Figure 12: CUDA speedup over the serial CPU implementation for each phase-unwrapping stage. Speedup is computed as serial runtime divided by CUDA runtime, with the dashed line indicating parity.

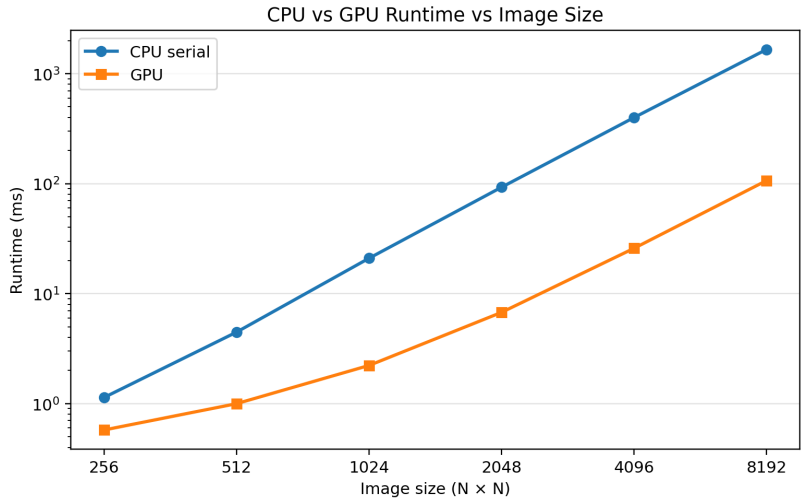


Figure 13: CPU serial and GPU runtime comparison across image sizes. Runtime is measured as sum of all 3 stages

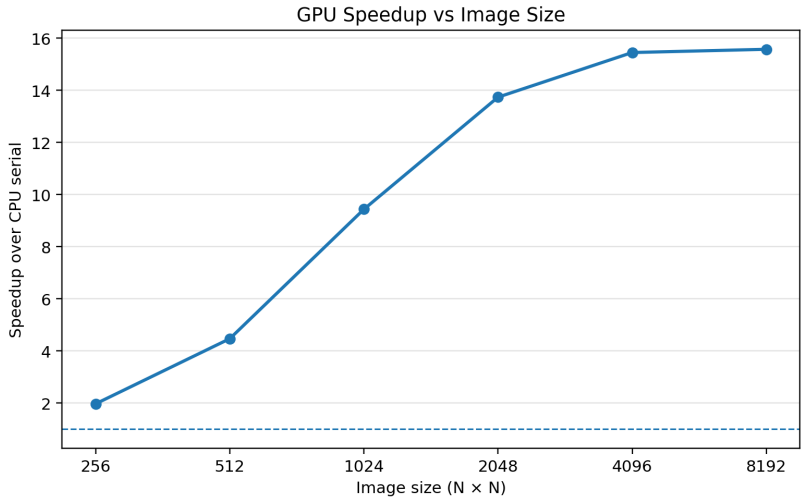
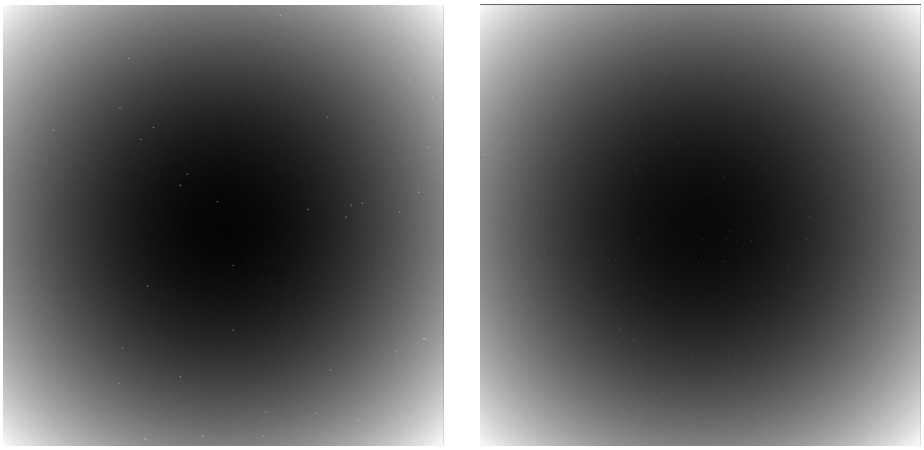


Figure 14: GPU speedup over the CPU serial implementation across image sizes. Speedup is computed as CPU runtime divided by GPU runtime



(a) 512×512 unwrapped output (b) 1024×1024 unwrapped output

Figure 15: Example unwrapped phase outputs generated by the final GPU phase-unwrapping pipeline. The concentric wrapped fringes are converted into a continuous phase surface, while isolated noisy residue locations remain visible as small artifacts.

Table 3: GPU compute-only per-stage runtime breakdown. All runtimes are in milliseconds and exclude setup, memory allocation, image loading, image writing, and other non-compute overheads.

Image Size	**Stage 1: Residues**	**Stage 2: Branch Cuts**	**Stage 3: Unwrap**
256×256	0.129	0.090	0.356
512×512	0.284	0.093	0.621
1024×1024	0.789	0.096	1.337
2048×2048	2.469	0.127	4.158
4096×4096	9.251	0.486	16.112
8192×8192	37.541	4.361	64.801

Table 4: CPU serial and GPU total runtime comparison. All runtimes are in milliseconds. Speedup is computed as CPU runtime divided by GPU runtime.

Image Size	**CPU Runtime**	**GPU Runtime**	**Speedup**
256×256	1.135	0.575	1.97×
512×512	4.462	0.998	4.47×
1024×1024	20.965	2.222	9.44×
2048×2048	92.766	6.754	13.73×
4096×4096	399.276	25.849	15.45×
8192×8192	1661.473	106.703	15.57×

The per-stage measurements show that Stage 1 and Stage 3 provide most of the overall acceleration. For the 8192×8192 input, Stage 1 residue detection decreases from 641.046 ms on the CPU to 37.541 ms on the GPU, while Stage 3 unwrapping decreases from 987.885 ms to 64.801 ms. Stage 2 branch-cut construction is much smaller in absolute runtime, taking only 4.361 ms on the GPU for the largest image. Although Stage 2 has less impact on total runtime than the other two stages, moving it onto the device is still important because it keeps the full pipeline on the GPU and avoids unnecessary host-device communication between stages.

The final implementation is successful in achieving the main project goal of accelerating Goldstein-style phase unwrapping with CUDA. The best overall computation-region speedup is about 15.6×, and the GPU implementation scales much better than the serial CPU baseline as image size increases. The results also show that the optimization is not simply a constant-factor improvement. Instead, the benefit grows with problem size, which indicates that the implementation is exploiting data parallelism across pixels, residues, tiles, and local unwrapping regions.

The remaining speedup is limited by several factors. Stage 3 still contains dependency-heavy flood-fill style propagation, which is less SIMD-friendly than a fully regular stencil computation. Neighbor expansion creates irregular control flow, and different tiles may finish with different amounts of work. Stage 2 also contains irregular residue-matching behavior, where the number and placement of residues affect the amount of work per bin. These effects can reduce SIMD efficiency through branch divergence and load imbalance. In addition, the large-image cases are likely limited partly by global memory traffic because each stage reads and writes large image-sized arrays. Therefore, the final speedup is limited by a combination of algorithmic dependencies, irregular parallelism, and memory bandwidth rather than by lack of raw GPU compute alone.

Overall, the results demonstrate that the CUDA implementation substantially improves the computational performance of the phase-unwrapping pipeline, especially for large images. The strongest evidence is the consistent increase in speedup with image size and the large reductions in Stage 1 and Stage 3 runtime. The implementation therefore meets the project goal of converting the original

CPU-oriented phase-unwrapping workflow into a GPU-accelerated pipeline with meaningful speedup on realistic high-resolution inputs.

Contributions

- Stage 1 (Residue Identification): Anurag
- Stage 2 (Branch-Cut Placement): Jack
- Stage 3 (Phase Integration): Both
- Final Report: Both
- Poster: Both
- Webpage: Both

Post-solution contribution split: 50–50

References:

- 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.
- williamdelacruz/parallel-Goldstein
- 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.