GPU-Accelerated Image Inpainting: Performance Analysis and Implementation

Your Name

June 28, 2025

1 GPU Acceleration in Patch-Based Image Inpainting

1.1 Motivation and Computational Challenges

Patch-based image inpainting algorithms face significant computational bottlenecks that limit their practical applicability. The core algorithm requires intensive operations across multiple computational domains including gradient calculations, priority computations, patch matching, and iterative processing.

For an image of dimensions $H \times W$ with patch size P, the computational complexity of patch matching scales as $\mathcal{O}(H \times W \times H \times W \times P^2)$ in the naive CPU implementation.

1.2 GPU Implementation Architecture

Our GPU-accelerated implementation leverages CUDA parallel computing to address computational challenges through strategic parallelization of three critical components. **Memory management** forms the foundation of our approach, where all primary data structures including the original image, working image, mask, confidence map, and intermediate computation arrays are allocated directly on GPU memory using CUDA device allocation. This strategy eliminates expensive CPU-GPU data transfers during the iterative inpainting process, as all operations remain within GPU memory space. The CUDA kernel configuration employs optimized 2D thread blocks of size 16×16 (256 threads per block) with grid dimensions calculated to cover the entire image dimensions, ensuring optimal GPU occupancy and memory coalescing patterns.

```
Memory Management Pseudocode:
ALLOCATE d_image, d_mask, d_working_image ON GPU
ALLOCATE d_confidence, d_data, d_priority ON GPU
CONFIGURE threads_per_block = (16, 16)
CALCULATE grid_dimensions = ceil(width/16) × ceil(height/16)
```

Parallel gradient computation transforms the traditionally sequential gradient calculation into a massively parallel operation where each pixel's gradient is computed independently by a separate GPU thread. The implementation uses central difference approximation for interior pixels and forward/backward differences for boundary pixels, maintaining numerical consistency with standard CPU implementations. Each thread processes one pixel coordinate (y,x) obtained from the CUDA grid, computes both horizontal and vertical gradients, and calculates the gradient magnitude. This approach achieves near-linear speedup with the number of available GPU cores, effectively eliminating gradient computation as a sequential bottleneck.

```
Parallel Gradient Pseudocode:
FOR each thread (y,x) in parallel:
    IF boundary_pixel:
        gradient = forward_or_backward_difference
    ELSE:
        gradient = central_difference
    STORE gradient_x[y,x], gradient_y[y,x], gradient_magnitude[y,x]
```

Massively parallel patch distance calculation represents the most significant computational acceleration, transforming patch matching from an $\mathcal{O}(H \times W \times H \times W \times P^2)$ sequential operation into an $\mathcal{O}(H \times W \times P^2)$ parallel computation. Each GPU thread evaluates one potential source patch location against the target patch, computing combined Minkowski and Chebyshev distance metrics in LAB color space for perceptual accuracy. The kernel first validates source patch boundaries and ensures all pixels are from known regions, then performs pixel-wise comparisons within the patch area. Thousands of threads execute simultaneously, each computing distances for different source locations, with the final minimum distance selection performed using GPU reduction operations or CPU-side processing for consistent tie-breaking behavior.

```
Parallel Patch Distance Pseudocode:
```

```
FOR each thread (source_y, source_x) in parallel:
    VALIDATE source_patch_boundaries
    IF any_pixel_in_source_patch_is_unknown:
        SET distance = infinity
        RETURN

    total_distance = 0
    FOR each pixel in patch:
        IF target_pixel_is_known:
            pixel_distance = compute_LAB_distance(target, source)
            UPDATE total_distance

    STORE distance[source_y, source_x] = total_distance
```

1.3 Performance Analysis

1.3.1 Algorithmic Complexity Reduction

The parallel implementation achieves significant complexity reduction:

$$\text{CPU Complexity:} \quad \mathcal{O}(H \times W \times H \times W \times P^2) \tag{1}$$

$$\text{GPU Complexity:} \quad \mathcal{O}(H \times W \times P^2) \tag{2}$$

$$\text{Speedup Factor:} \quad \mathcal{O}(H \times W) \tag{3}$$

For high-resolution images (1920×1080), this represents a theoretical speedup factor of over 2 million for patch matching.

1.3.2 Implementation Benefits

- **Memory Bandwidth**: GPU memory bandwidth ($\sim$1000 GB/s) vs CPU ($\sim$100 GB/s)
- **Parallel Execution**: Thousands of GPU cores vs 4-16 CPU cores
- **Optimized Access Patterns**: Memory coalescing and occupancy optimization

1.4 Practical Applications

This GPU acceleration enables:

1. **Interactive Image Editing**: Real-time inpainting for user interfaces
2. **Batch Processing**: Efficient processing of large image datasets
3. **High-Resolution Processing**: Ultra-high-resolution image handling
4. **Parameter Exploration**: Rapid algorithm parameter optimization

The implementation successfully adapts classical computer vision algorithms to modern parallel computing architectures while preserving algorithmic fidelity and numerical accuracy.