Parallel Image Inpainting

Saileshwar Karthik
Information Networking Institute
Carnegie Mellon University
Pittsburgh, PA 15213
saileshk@andrew.cmu.edu

Ayushi Bansal
Information Networking Institute
Carnegie Mellon University
Pittsburgh, PA 15213
ayushib@andrew.cmu.edu

1 Summary

The objective of this project is to accelerate the patch-match based image inpainting algorithm using multiple parallelization strategies and to analyze their different tradeoffs and bottlenecks. We implemented parallel versions of the algorithm in CUDA on the GPU and a hybrid version using both GPU and CPU parallelism.

Keywords: Nearest Neighbor Field (NNF), PatchMatch, CUDA, OpenMP, Expectation-Maximization (EM), Bidirectional Similarity (BDS)

2 Background

Image inpainting is the task of filling in damaged, deteriorated, or missing regions of an image so that the reconstructed image appears visually plausible to a human observer. Before the widespread use of digital editing tools such as Adobe Photoshop, automatic algorithmic approaches were largely *diffusion-based*. These methods solve partial differential equations that propagate color and gradient information from the boundaries of the hole inward. They handle small holes and thin scratches well, but result in poor-quality images on larger images because diffusion is fundamentally incapable of synthesizing texture, causing the filled area to appear smeared.

Exemplar-based methods were the first to fill holes while maintaining the texture and structure of the rest of the image, by sampling and copying color patches from known parts of the image that best match the boundary. Criminisi et al. [2] introduced a greedy variant that prioritized filled along the strong edges first, allowing structure to propagate into the hole before texture filled in around it. Wexler et al. [5] later reformulated the problem as a global optimization solved by an EM-like loop. Expectation-Maximization (EM) is a general iterative strategy that alternates between an *expectation* (E) step, which uses the current estimate to infer hidden

information, and a *maximization* (M) step, which uses that information to produce a better estimate, gradually converging on a solution.

2.1 Patch Match (Serial)

PatchMatch [1] is a randomized algorithm for efficiently computing an approximate NNF between two images, Rather than exhaustively comparing every target patch against every source patch, it exploits two key insights that drives the algorithm. The first being, good patch matches can be found via random sampling, and secondly, natural coherence in the imagery allows the propagation of these good matches to surrounding areas. A correct match at one pixel can therefore be *propagated* to its neighbors without second thought.

The approximate nearest neighbor algorithm proceeds in two main stages:

2.1.1 Initialization

The NNF is filled with completely random offsets, effectively associating each target patch a completely random candidate match in the source image. Despite the poor quality of these guesses, the larger the image gets, the more statistically likely it is that at least a few are close to their true nearest neighbors.

2.1.2 Iterative Refinement

After initialization, the field is iteratively refined through two interleaved operations: propagation and random search

Propagation The field is traversed in scan order (left-to-right, top-to-bottom on odd iterations, and reversed on even iterations). At each pixel, the current offset is compared against the offsets of its already-processed neighbors, and the one yielding the smallest patch distance is kept. Good matches thus spread rapidly across coherent regions of the image.

Random Search To escape local minima introduced by propagation, each pixel additionally tests a sequence of candidate offsets sampled at exponentially decreasing distances from its current best guess. This allows the algorithm to discover better matches that propagation alone could not reach.

Halting Criteria In practice, it has been found that the NNF has almost always converged at around 4-5 iterations in total

2.2 Inpainting using PatchMatch

To turn PatchMatch's algorithm into an editing tool such as inpainting, the paper builds on the *bidirectional similarity* (BDS) synthesis framework of Simakov et

al. [4]. Given a source (input) image S and a target (output) image T, the BDS distance measures how well the two images cover each other in terms of small patches:

$$d_{BDS}(S,T) = \underbrace{\frac{1}{N_S}\sum_{s\subset S}\min_{t\subset T} D(s,t)}_{d_{\text{complete}}(S,T)} + \underbrace{\frac{1}{N_T}\sum_{t\subset T}\min_{s\subset S} D(t,s)}_{d_{\text{cohere}}(S,T)} \tag{1}$$

The *completeness* term ensures the output retains as much visual information from the input as possible, and the *coherence* term ensures the output is consistent with the input and does not invent spurious new structures. The patch distance D is the sum of squared differences in pixel values. Inpainting is recovered as the special case where T is the image with the hole and S is the image with the known region: solving for the T that minimizes d_{BDS} produces a fill that is both faithful to and coherent with the surrounding image.

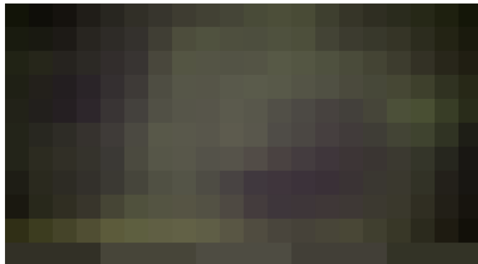
This objective is minimized iteratively by an EM-like loop, with PatchMatch used inside the E step to compute the required NNFs at interactive speeds. Each EM iteration consists of:

E step (patch voting). PatchMatch is run twice to compute both directions of the NNF: source-to-target (for the completeness term) and target-to-source (for the coherence term). Then, for every patch correspondence found, every pixel inside that patch contributes a weighted "vote" for what its color should be in the output, where the weight is a similarity score derived from the patch distance. Votes from all overlapping patches are then accumulated into a per-pixel sum.

M step (pixel update). Each pixel in T is updated to the weighted average of all the votes it received in the E step, producing the new estimate T_{i+1}.

(a) Level 7 (res: 5x2)

(b) Level 6 (res: 10x5)



(c) Level 5 (res: 20x11)

(d) Level 4 (res: 40x22)

(e) Level 3 (res: 80x45)

(f) Level 2 (res: 160x90)

(g) Level 1 (res: 320x180)

Figure 1: The inpainted result at each level of the image pyramid (coarse to fine)

These two steps are repeated for several iterations at each level of a *coarse-to-fine iamge pyramid*. This pyramid (Figure 1) is a stack of progressively downsampled versions of the input image, where the coarsest level only captures low-frequency structure, and subsequent levels double in resolution and add finer details until the original image size is reached.

The EM loop begins at the coarsest level, where the hole is small and the global structure of the fill is easier to recover; the result is then upsampled to initialize the next finer level, where additional EM iterations refine the result with higher-frequency detail. This way, large-scale structure is resolved first and progressively sharpened into texture, avoiding the bad local minima that a direct full-resolution optimization would fall into.

Together, this yields the full serial version of the inpainting pipeline we implemented in C++.

2.3 Algorithmic Structure and Parallelism Analysis

Input. The pipeline takes a 3-channel RGB image, a 1-channel grayscale mask and the patch size as input. The mask identifies the missing region (the hole) within the image passed in.

Output. A single image with the same dimensions of the input image is returned in which the masked region has been filled with content synthesized from the unmasked region.

Key data structures. The implementation is built around three principal data structures.

`MaskedImage` bundles together a `cv::Mat` (an n-dimensional dense numerical single-channel or multi-channel array) of RGB pixels with an associated mask and (optionally) a global mask and precomputed image gradients. It exposes pixel and mask accessors and is the unit operated on at every level of the pyramid.

`NearestNeighborField` stores, for every patch in a source image, a pointer to its current best-matching patch in a target image. It is implemented as a 3-channel integer matrix indexed by source coordinates: channel 0 holds the matched target y, channel 1 the matched target x, and channel 2 the patch distance. Two NNFs are maintained simultaneously, `m_source2target` for the BDS completeness term and `m_target2source` for the coherence term.

`Image pyramid` is a vector of `MaskedImages`, each successive entry being a down-sampled version of the previous, terminating when the image becomes

smaller than the patch size. The EM loop traverses this vector from coarsest to finest.

Key operations. Each EM iteration performs four main operations. A fifth, *patch distance computation*, is called out separately because it is the shared inner routine of both the NNF minimization and the E step.

1. **Set identity** Set the nearest patch of each and every pixel to itself if the patch centered at the pixel overlaps with the mask
2. **NNF minimization** via PatchMatch (propagation along scan order plus random search), updating both the source-to-target and target-to-source fields;
3. **E step** where every patch correspondence in the NNF contributes weighted "votes" for the colors of the pixels it covers, accumulating into a per-pixel sum;
4. **M step** where each output pixel is updated to the weighted average of its accumulated votes; and
5. **Patch distance computation** an SSD (sum of squared differences) over a $(2p+1) \times (2p+1)$ window (p being the patch radius) of RGB and gradient values, called once per candidate offset inside both PatchMatch and the E step.

Where the cost lies. Profiling of the serial implementation as seen in Figure 2 shows that majority of runtime is spent in two places: the NNF minimization, and the E step. Thus, these two are the primary targets for parallelism.

Operation	Time (s)	% of Total Time
Set identity	0.06	1.98
NNF Minimize	2.079	68.72
Expectation	0.879	29.05
Maximization	0.007	0.23

Figure 2: Time spent per operation within one iteration of the serial algorithm against the finest level of the pyramid on an image of size 640x360

Dependencies and parallelism. All four operations covered earlier differ significantly in how parallelizable they are.

1. **Set identity**. This is embarrassingly parallel. Each pixel simply performs an independent local read of the mask and writes only its own NNF entry, with

no cross-pixel dependencies and no write conflicts. Memory access is strictly stride-1 in scan order, which gives excellent cache behavior and coalesces well on the GPU.

2. **NNF minimization**. This is serial in nature according to PatchMatch's original design. The *propagation* step at pixel (y, x) reads the recently updated offset at $(y-1, x)$ and $(y, x-1)$, so a strict scan-order traversal is sequential. This is the hardest part of the algorithm to parallelize while preserving the original convergence properties. The *random search* step within each pixel is independent across pixels, since each only reads and writes its own NNF entry. However, this step has terrible spatial locality across patch searches - the patch SSD computation at each random probe reads a full patch at a random location in the image while consecutive pixel probes point to completely different patches
3. **E step**. This is data-parallel across source patches. In this step, every patch independently determines what to vote for. The complication is a write conflict on the shared accumulator buffer where multiple patches overlap and write to the same target pixels.
4. **M step**. This is embarassingly parallel. Each output pixel divides its accumulated color sum by its accumulated weight; no cross-pixel dependencies exist.

2.4 Evaluation Metrics

We primarily used two different metrics:

1. **Peak Signal-to-Noise Ratio** To verify that our parallel implementations generate a visually high quality image in comparison to the serial baseline, we evaluate the similarity between the inpainted output of each parallel version and that of the serial reference using the *Peak Signal-to-Noise Ratio* (PSNR). PSNR is a standard pixel-level fidelity metric in image processing, defined as

$$\text{PSNR} = 10 \cdot \log_{10}\left(\frac{\text{MAX}_I^2}{\text{MSE}}\right) \tag{2}$$

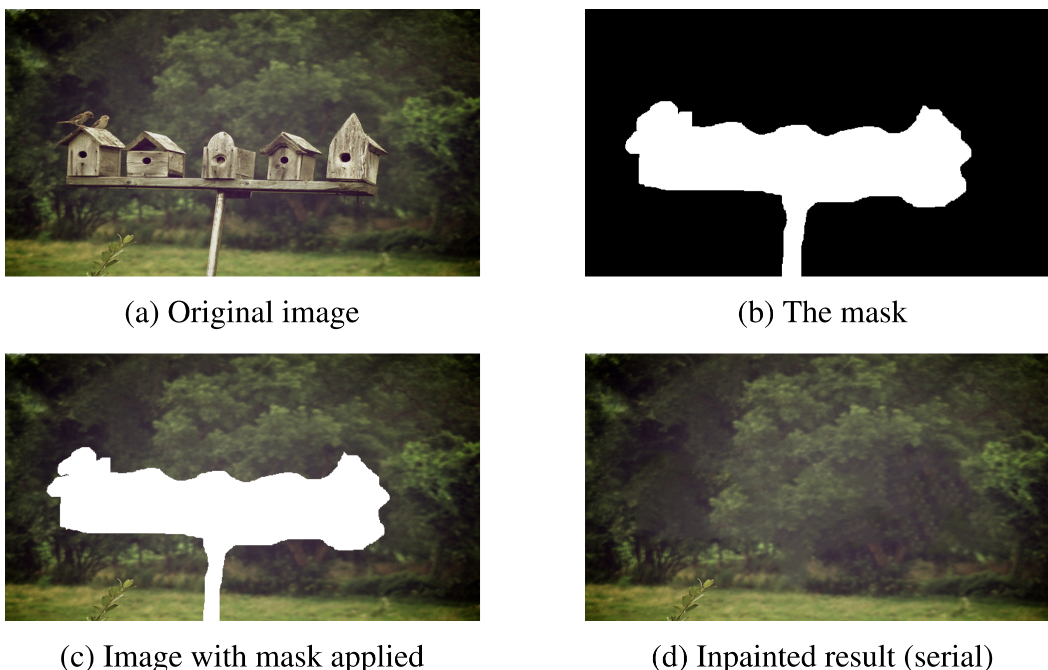
where MAX_I is the maximum possible pixel value (255 for 8-bit images) and MSE is the mean squared error between the two images, computed as the average squared difference of corresponding pixel values across all channels. Because PSNR is on a logarithmic scale, higher values indicate greater similarity (identical images produce an undefined PSNR since MSE is zero), while values above 40 dB typically indicate that the two images are visually indistinguishable, and values in the 30-40 dB range correspond to small but perceptible differences. Bit-exact comparison is unsuitable here, since

PatchMatch's randomness causes thread-order variations to produce slightly different but equally valid outputs; PSNR instead measures whether such variation stays within the algorithm's expected noise level.

2. **Computational Speedup** The primary objective of the parallel implementations is to reduce end-to-end runtime, and we measure this using *speedup*, defined as

$$\text{Speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \tag{3}$$

where T_{serial} is the wall-clock execution time of the serial baseline and T_{parallel} is the wall-clock time of the parallel version under evaluation, both measured on identical inputs.



(a) Original image (b) The mask

(c) Image with mask applied (d) Inpainted result (serial)

Figure 3: Input pipeline and inpainting result (patch size = 3).

3 Approach

All the different approaches (including the serial version) discussed below are run against the input pipeline shown in Figure 3. GHC machines were used as the underlying platform (NVIDIA RTX 2080 GPU).

3.1 Version 0 - Serial Implementation

We directly followed the algorithm laid out in the PatchMatch paper [1] in developing the serial implementation of the inpainting algorithm. We referred to several open source repositories in achieving this (cited in the code).

The resulting inpainted image of the serial implementation on the input pipeline shown in Figure 3 can be seen in Figure 3d.

3.2 Version 1 - Red Black Ordering

After getting a working version of the serial algorithm, we adopted an iterative approach to parallelize the different parts of the algorithm using CUDA. The first target being the NNF minimize step.

The challenge with parallelising NNF minimization is the propagation dependency described earlier: in the original serial formulation, the update at pixel (y, x) reads the recently updated offsets at $(y - 1, x)$ and $(y, x - 1)$, so adjacent pixels cannot be processed simultaneously without breaking the dependency chain. To break this chain while preserving the spirit of the algorithm, we adopted a *red-black ordering* scheme, borrowed from one of the 15-618 classes.

Under red-black ordering, the pixel grid is partitioned into two interleaved sets like the squares of a checkerboard: "red" pixels are those where $(y + x)$ is even, and "black" pixels are those where $(y + x)$ is odd. Each iteration is then split into two passes. In the first pass, every red pixel is updated in parallel while in the second pass, every black pixel is updated in parallel using the freshly written red values as input. Together, the two half-passes accomplish the same effective propagation as a single serial scan, but with all the work inside each half-pass exposed as independent data-parallel updates.

After implementing the CUDA kernel using the red-black ordering in the propagate step of the NNF minimization phase, we observed a total speedup of **2.9x** over the serial algorithm and a PSNR score of **36.558 dB**.

3.3 Version 2 - Ping-Pong Scheme

An inefficiency in Version 1 was that every NNF minimization call uploaded its inputs to the device and freed everything before returning. The host-side `NearestNeighborField::minimize_cuda` routine would `cudaMalloc` fresh buffers for the source and target images, their gradients, masks, and the field itself, copy all of this data from host to device, run the kernels, copy the result back, and then `cudaFree` everything. This pattern repeated twice per EM iteration, once for the source-to-target field and once for the target-to-source field, and at every level of the pyramid. At small pyramid levels the device-side computation is fast enough that the allocation and transfer overhead dominated total runtime entirely. This idea was taken from the ping-pong scheme explained within Hanli et al. [7]

Version 2 addressed this by introducing two new structs to hold persistent device-side state across the entire pipeline:

Algorithm 1 NNF Minimize Kernel (Version 1: Red-Black Ordering)

Require: Field f, source/target images and gradients, masks, patch size p, color $c \in \{0, 1\}$, random seed

Ensure: Updated field f with improved nearest-neighbor offsets for one color group

1: $idx \leftarrow \texttt{blockIdx.x} \cdot \texttt{blockDim.x} + \texttt{threadIdx.x}$
2: **if** $idx \geq H_{src} \cdot W_{src}$ **then return**
3: **end if**
4: $(y, x) \leftarrow (\lfloor idx/W_{src} \rfloor,\ idx \bmod W_{src})$
5: **if** $(x + y) \bmod 2 \neq c$ **then return** ▷ skip pixels of the wrong color this pass
6: **end if**
7: $(b_y, b_x, b_d) \leftarrow f[idx]$ ▷ load current best match for this pixel
// Propagation phase
8: **for all** $(\Delta y, \Delta x) \in \{(-1, 0), (1, 0), (0, -1), (0, 1)\}$ **do**
9: $\quad (y', x') \leftarrow (y + \Delta y, x + \Delta x)$
10: $\quad$ **if** (y', x') out of bounds **or** masked by gmask **then continue**
11: $\quad$ **end if**
12: $\quad (n_y, n_x, \cdot) \leftarrow f[y' \cdot W_{src} + x']$ ▷ neighbor's current offset
13: $\quad c_y \leftarrow \text{clamp}(n_y - \Delta y,\ 0,\ H_{tgt} - 1)$
14: $\quad c_x \leftarrow \text{clamp}(n_x - \Delta x,\ 0,\ W_{tgt} - 1)$ ▷ shifted candidate
15: $\quad d \leftarrow \textsc{PatchDistance}(y, x, c_y, c_x)$
16: $\quad$ **if** $d < b_d$ **then**
17: $\quad\quad (b_y, b_x, b_d) \leftarrow (c_y, c_x, d)$
18: $\quad$ **end if**
19: **end for**
// Random search phase
20: $r \leftarrow (\min(H_{tgt}, W_{tgt}) - 1)/2$
21: $step \leftarrow 0$
22: **while** $r > 0$ **do**
23: $\quad y_p \leftarrow \text{clamp}(b_y + \textsc{RandRange}(-r, r),\ 0,\ H_{tgt} - 1)$
24: $\quad x_p \leftarrow \text{clamp}(b_x + \textsc{RandRange}(-r, r),\ 0,\ W_{tgt} - 1)$
25: $\quad d \leftarrow \textsc{PatchDistance}(y, x, y_p, x_p)$
26: $\quad$ **if** $d < b_d$ **then**
27: $\quad\quad (b_y, b_x, b_d) \leftarrow (y_p, x_p, d)$
28: $\quad$ **end if**
29: $\quad r \leftarrow r/2$
30: $\quad step \leftarrow step + 1$
31: **end while**
32: $f[idx] \leftarrow (b_y, b_x, b_d)$ ▷ write back winning offset

- `CudaImageDeviceBuffers` owns the device pointers for one `MaskedImage`: its RGB pixels (`img`), the precomputed x- and y-gradients (`gx`, `gy`), the mask, and the optional global mask. It also tracks its own `pixel_capacity` so it can grow lazily as the pipeline ascends the pyramid to higher resolutions.
- `CudaNNFDeviceBuffers` owns the NNF field pointer along with two `CudaImageDeviceBuffers` (one for the source image, one for the target). It is constructed once at the start of the pipeline and lives for the duration of the run, with its `allocate_device_buffers` method ensuring the underlying buffers are large enough for the current pyramid level.

With this in place, we were able to observe a total speedup of **2.92x**. The PSNR score remains the same.

3.4 Version 3 - Jump Flooding

Before moving on to parallelizing the rest of the operations, we spent some time researching different ways to parallelize the propagate phase in particular. Recent work on parallel patch-match implementations [7] [6] have all used jump-flood [3] based algorithms.

The fundamental limitation of red-black ordering is that information still propagates only one pixel per pass. A good match found at the top-left corner of the image takes $O(N)$ passes to reach the bottom-right corner, since each pass can only push offsets by a single step in any direction. For images of even modest size this is a severe bottleneck: not only does it require many sequential passes to converge, it also wastes the GPU's massive parallelism on shallow per-pass work that is dominated by kernel launch overhead.

Jump-flood propagation [3], originally introduced for computing Voronoi diagrams on the GPU, replaces the one-pixel-per-pass scheme with one in which each pixel reads candidate offsets from neighbors at exponentially decreasing distances. In the first pass, every pixel checks neighbors at a jump distance of 8 pixels in each direction; in the next, at 4; then 2, 1, and so on. After only $\log_2 N$ passes, information from any pixel can reach any other pixel in the image, compared to the $O(N)$ passes required by red-black. The PatchMatch authors themselves prototype this scheme for their GPU variant, noting that the resulting algorithm is roughly $7\times$ faster than their serial implementation [1].

Jump-flood also fits the ping-pong buffer scheme introduced in Version 2 naturally. Each jump distance is a single kernel launch that reads from the "previous" field buffer and writes to the "current" one, with no synchronization needed within the pass since reads and writes are disjoint. There is no parity bookkeeping, no checkerboard

predicate at every thread, and no concern about whether a neighbor's offset has been updated yet in the current pass.

For these reasons, we replaced the red-black propagation kernel from Version 1 with a jump-flood kernel using the jump sequence $\{8, 4, 2, 1, 2, 1\}$. The two trailing $\{2, 1\}$ passes are a refinement step recommended by Yu et al.[6] to clean up the small number of pixels that the coarse early jumps may have missed, at negligible additional cost. After implementing jump-flood propagation, total speedup over the serial algorithm rose from **2.92x** (Version 2) dropped to **2.855x**, but the PSNR score went up to **41.2469 dB** (higher is better).

For a slight drop in speedup, we were able to achieve a better quality output image, so we continued to iterate on top of this design.

3.5 Version 4 - Initialization on CUDA

Versions 1-3 focused exclusively on the NNF minimization operation. Profiling the pipeline with timing code revealed that roughly half the time spent on each EM iteration of a given pyramid level was now being consumed by the `_initialize_field_from` routine, which fills the field for the current level by upsampling the result from the previous (coarser) level and then randomizing any entries that fall inside the hole.

Thus, we ported the field initialization routine to a CUDA kernel, in which each thread computes the upsampled and scaled offset for one source pixel and writes it directly into the device field buffer. This eliminates both the CPU computation and the subsequent host-to-device transfer of the freshly initialized field.

With initialization, set identity, propagation, and random search now all running on the device, it became wasteful to continue allocating and freeing the NNF field buffers around each minimization call. In Version 2, the persistent device state held only the source and target image buffers while the NNF field itself was still allocated and freed inside the minimization function. The actual ping-pong introduced in Version 3 was implemented using local pointers within that function, so its lifetime was bounded by a single call. Version 4 lifts these buffers into `CudaNNFDeviceBuffers` itself, with capacity counters that grow the underlying allocations only when a larger pyramid level is reached. The result is that an entire run of the inpainting pipeline allocates each NNF buffer exactly once at the largest pyramid level visited, and reuses it for every EM iteration at every level thereafter.

This version yielded with a speedup of **4.229x** and a PSNR score of **42.55 dB** (similar to V3).

Algorithm 2 NNF Minimize Kernel (Version 3: Jump Flood)

Require: Field f, source/target images and gradients, masks, patch size p, jump distance j, random seed

Ensure: Updated field f with improved nearest-neighbor offsets at jump distance j

1: $idx \leftarrow$ `blockIdx.x` $\cdot$ `blockDim.x` $+$ `threadIdx.x`
2: **if** $idx \geq H_{src} \cdot W_{src}$ **then return**
3: **end if**
4: $(y, x) \leftarrow (\lfloor idx/W_{src} \rfloor,\ idx \bmod W_{src})$
5: **if** globally masked at (y, x) **then return** ▷ nothing to match for fully-masked source pixels
6: **end if**
7: $(b_y, b_x, b_d) \leftarrow f[idx]$ ▷ load current best match for this pixel
// Propagation phase: 8 neighbors at jump distance j
8: **for all** $(\Delta y, \Delta x) \in \{(\pm j, 0),\ (0, \pm j),\ (\pm j, \pm j)\}$ **do**
9: $(y', x') \leftarrow (y + \Delta y, x + \Delta x)$
10: **if** (y', x') out of bounds **or** globally masked **then continue**
11: **end if**
12: $(n_y, n_x, \cdot) \leftarrow f[y' \cdot W_{src} + x']$ ▷ neighbor's current offset
13: $c_y \leftarrow \mathrm{clamp}(n_y - \Delta y,\ 0,\ H_{tgt} - 1)$
14: $c_x \leftarrow \mathrm{clamp}(n_x - \Delta x,\ 0,\ W_{tgt} - 1)$ ▷ candidate shifted by j
15: $d \leftarrow \textsc{PatchDistance}(y, x, c_y, c_x)$
16: **if** $d < b_d$ **then**
17: $(b_y, b_x, b_d) \leftarrow (c_y, c_x, d)$
18: **end if**
19: **end for**
// Random search phase
20: $r \leftarrow (\min(H_{tgt}, W_{tgt}) - 1)/2$
21: $step \leftarrow 0$
22: **while** $r > 0$ **do**
23: $y_p \leftarrow \mathrm{clamp}(b_y + \textsc{RandRange}(-r, r),\ 0,\ H_{tgt} - 1)$
24: $x_p \leftarrow \mathrm{clamp}(b_x + \textsc{RandRange}(-r, r),\ 0,\ W_{tgt} - 1)$
25: **if not** globally masked at (y_p, x_p) **then**
26: $d \leftarrow \textsc{PatchDistance}(y, x, y_p, x_p)$
27: **if** $d < b_d$ **then**
28: $(b_y, b_x, b_d) \leftarrow (y_p, x_p, d)$
29: **end if**
30: **end if**
31: $r \leftarrow r/2$
32: $step \leftarrow step + 1$
33: **end while**
34: $f[idx] \leftarrow (b_y, b_x, b_d)$ ▷ write back winning offset

3.6 Version 5 - Memory Coalescing

At this stage, we made use of the Nsight profiler to spot any hints on improving the performance of the two CUDA kernels implemented thus far. The profiler revealed that **88%** of the sectors fetched were redundant, indicating a large amount of uncoalesced memory accesses within the *nnf_jump_flood_kernel*.

The cause was the data layout of the per-image device buffers. Up to Version 4, each image's RGB pixels, mask, and per-channel gradients were stored as four separate `unsigned char *` arrays:

```
struct DeviceImageBuffers {
    unsigned char *img;   // 3 bytes per pixel
    unsigned char *gx;    // 1 byte per pixel
    unsigned char *gy;    // 1 byte per pixel
    unsigned char *mask;  // 1 byte per pixel
    ...
};
```

This layout had been carried over from the original CPU implementation, which used `cv::Mat` containers for each. On the GPU it interacts poorly with the patch distance computation, which is the inner work of both NNF minimization and the E step. For each candidate offset, every thread reads the RGB, mask, and gradient values for every pixel in a $(2p + 1) \times (2p + 1)$ patch. With the data spread across four separate arrays, a single patch read costs four independent global-memory transactions, and because the per-pixel data for a given pixel sits at the same index in each array but in different cache lines, none of the four reads share a sector with one another. The hardware fetches a full 32-byte sector for each, but uses only one or three bytes of it.

Version 5 addresses this by repacking the per-pixel image data into a single `uchar4` array per buffer.

```
struct DeviceImageBuffers {
    uchar4 *rgb_mask;     // RGB + mask packed into 4 bytes
    uchar4 *gx;           // gradient x (3 useful bytes per pixel)
    uchar4 *gy;           // gradient y (3 useful bytes per pixel)
    ...
};
```

The three RGB bytes and the mask byte share one 32-bit word, so a pixel's color and its mask flag arrive together in a single aligned load. The gradients are similarly promoted from `unsigned char` to `uchar4` (with three of the four bytes carrying useful values, keeping the layout uniform and aligned). The patch distance routine is updated accordingly. It now issues a single `uchar4` load, after which the individual channels are extracted by simple byte unpacking.

After this change, Nsight reported that **71%** of the sectors fetched were redundant indicating that theres an increase in the amount of coalesced memory accesses.

With Version 5, we observed a speedup of **4.537x** and a PSNR score of **44.3016 dB**

3.7 Version 6 - EM on CUDA + Miscellaneous Optimizations

With Version 6, we finally turned towards parallelising the Expectation and the Maximization steps and ported them to CUDA kernels. Up to this point, at the end of every NNF minimization, the field had to be downloaded from the device, after which the E and M steps were performed on the CPU, and the resulting target image was then ready for the next iteration. The combination of repeated host-device transfers of the NNFs and the inability to overlap the E/M work with anything else had become the dominant cost of the iteration loop now that the NNF stage was largely optimized.

The E step was rewritten as `expectation_step_kernel`, with one thread per source-domain pixel. Each thread reads its NNF entry, looks up the corresponding target patch, and atomically accumulates weighted color votes into a shared device `vote` buffer of `float4` values, where the fourth component holds the accumulated weight. The kernel is launched twice per EM iteration - once over the source domain for the source-to-target term, and once over the target domain for the target-to-source term. The M step is then a single straightforward kernel, `maximization_step_kernel`, that assigns one thread per target pixel and divides the accumulated color sum by the accumulated weight to produce the final pixel value.

A key benefit of this port is that the E and M kernels read from exactly the same packed `uchar4` image buffers introduced in Version 5 and the same NNF buffers persisted in `CudaNNFDeviceBuffers` since Version 4. There is no need to upload either the source or the NNF before launching the E step: both are already resident on the device from the preceding minimization. Likewise, the M step writes directly into the target image's device-size `rgb_mask` buffer, in place. The only host-device transfer in the entire EM iteration is a single download of the updated target image at the end of the loop. The intermediate vote buffer never crosses the PCIe bus at all.

Beyond this, some other miscellaneous optimizations we made were the following:

1. Added early termination in the patch distance computation function. This was an idea suggested by the original PatchMatch paper [1].
2. Ran the initialization randomize kernels on parallel streams since they work on non-overlapping data structures
3. Removed redundant loads across all the kernels
4. Added **__*restrict*__** keyword to the read only buffers used by the CUDA kernels to allow the compiler to route memory loads through the read-only data cache.
5. Used OpenMP parallelism to parallelise the data preparation stages in the kernel launch functions

Having implemented all of this, we observed a total speedup of **29.47x** (which is well beyond the **7x** speedup reported by the PatchMatch paper [1]) and a PSNR score of **39 db**.

4 Results

The primary goal of this project was to implement a parallel version of PatchMatch that achieves a speedup greater than **7x**, which is the speedup reported by the original authors of the PatchMatch paper with their GPU version of the algorithm. We were able to beat this target on every image we tried (and for every patch size). We even completed the stretch goal of developing a web application that allows users to upload images, select a mask of their liking and run the inpainting algorithm.

As described in Section 2.4, we primarily use wall-clock speedup to measure performance and PSNR to measure inpainted image quality.

4.1 Environment

All experiments were conducted on the CMU GHC cluster machines. Each node is equipped with an 8-core Intel Core i7-9700 CPU running at 3.0 GHz and an NVIDIA RTX 2080 GPU.

4.2 Speedup Comparison and Analysis

4.2.1 Impact of CPU-GPU Co-optimisation

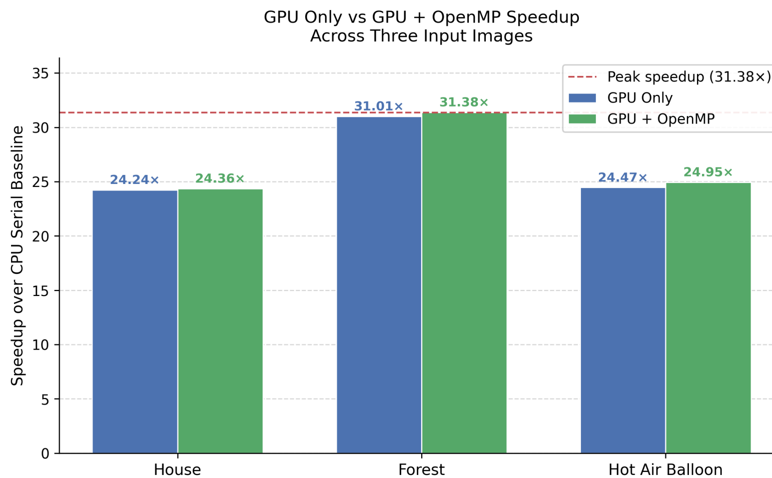


Figure 4: Speedup comparison across three input images for GPU Only vs GPU + OpenMP. The images can be seen in Figures 3, 8 and 7

The marginal difference between GPU-only and GPU+OpenMP execution confirms that the dominant bottleneck has shifted entirely to GPU computation and CPU-side

parallelism now contributes less than 1% additional speedup across all tested images. This is a natural consequence of moving the E and M steps on-device in V6, which eliminated the primary CPU workload. At this stage, further gains would require architectural changes such as reducing PCIe transfers between pyramid levels or increasing kernel arithmetic intensity rather than adding more CPU threads.

4.2.2 Effect of Hole Size on Speedup

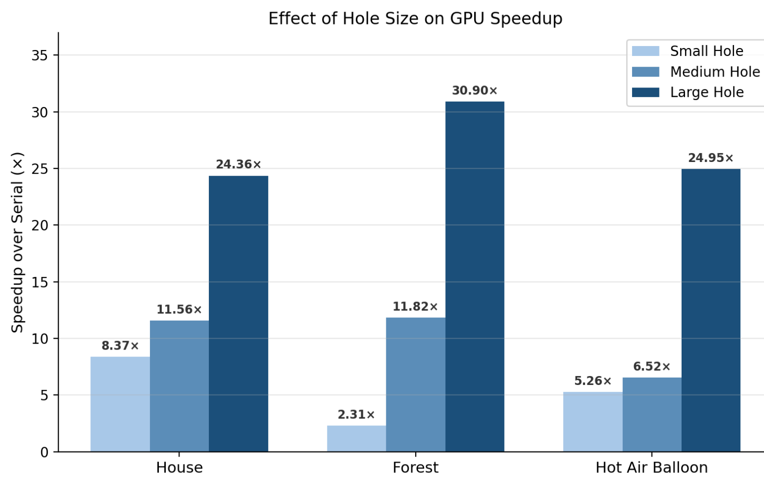


Figure 5: Effect of hole size on GPU speedup across three input images. The images can be seen in Figures 3, 8 and 7

As seen from Figure 5, larger holes consistently yield significantly higher speedups across all three images, with gains scaling from roughly 2–8× for small holes to 25–31× for large ones. This aligns with the theoretical expectation that a larger masked region increases the amount of parallel work available, and allows for higher speedup gains. There are more active threads in the E and M step kernels, better saturating the GPU's parallel execution units. Small holes leave the majority of threads idle, limiting the GPU's advantage over the sequential CPU baseline. Additionally, the NNF step scales well with more GPU threads participating in finding matches, therefore doing useful work, which when done on the CPU increases the sequential runtime.

4.2.3 Effect of Patch Size on Speedup

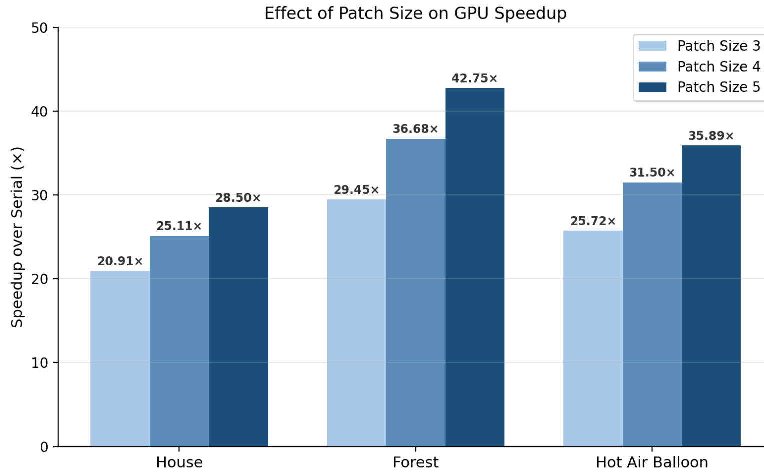


Figure 6: Effect of patch size on GPU speedup across three input images. The images can be seen in Figures 3, 8 and 7

As seen from Figure 6, larger patch sizes consistently amplify GPU speedup, with patch size 5 outperforming patch size 3 by roughly 35–45% across all images. This is because the patch distance computation in `compute_patch_dist` scales quadratically with patch size. This is because 5×5 patch requires nearly three times the arithmetic of a 3×3 patch. Since this computation is the dominant workload in the NNF kernels, larger patches shift the kernel further toward compute-bound behaviour, which the GPU handles far more efficiently than the CPU's sequential execution. This is consistent with the arithmetic intensity results in Table 1, where `nnf_jump_flood_kernel` already shows the highest intensity in the pipeline and larger patches push it even further in that direction. It is worth noting that these speedup gains come at the cost of declining visual quality across both the serial and parallel versions, which is consistent with the speedup-quality tradeoffs we've seen many a times.

4.3 Roofline Analysis

To characterize where the bottlenecks of each kernel lie, we collected per-kernel FLOP counts and DRAM traffic using Nsight Compute on the finest pyramid level of a representative input. The arithmetic intensity gives a quick read on whether a kernel is limited by the GPU's compute throughput or by its memory bandwidth. Table 1 shows these numbers for the five main kernels.

Kernel	FLOPs	DRAM bytes	AI (FLOPs/byte)
`nnf_randomize_kernel`	22,273,327	13,250,000	1.68
`nnf_set_identity_kernel`	230,400	2,960,000	0.08
`nnf_jump_flood_kernel*`	241,238,680	14,410,000	16.74
`expectation_step_kernel`	33,809,199	28,640,000	1.18
`maximization_step_kernel`	9,907,788	5,380,000	1.84

Table 1: Per-kernel arithmetic intensity at the finest pyramid level, run on the same image shown in Figure 3c, with patch size = 3. *Averaged across all jump distances.

The jump-flood kernel is the most compute heavy. At **16.74 FLOPs/byte**, jump-flood is by far the most compute-dense kernel in the pipeline. Most of its work is concentrated inside `compute_patch_dist`, which evaluates a $(2p+1) \times (2p+1)$ SSD over RGB and gradient channels for each of eight neighboring candidates per thread. Because the same patch data is reused across many candidate evaluations within one thread, the bytes pulled from DRAM are amortized over a large amount of arithmetic. This is also the kernel that gained the most from the lower-level optimizations.

Set-identity kernel is the least compute bound The `nnf_set_identity_kernel` has an intensity of only **0.08 FLOPs/byte**, making it almost purely memory-bound. Each thread reads a small mask window, performs a single conditional check, and writes back one NNF entry. Its absolute runtime contribution is also small, which is why it was never a target for optimization in our code iterations.

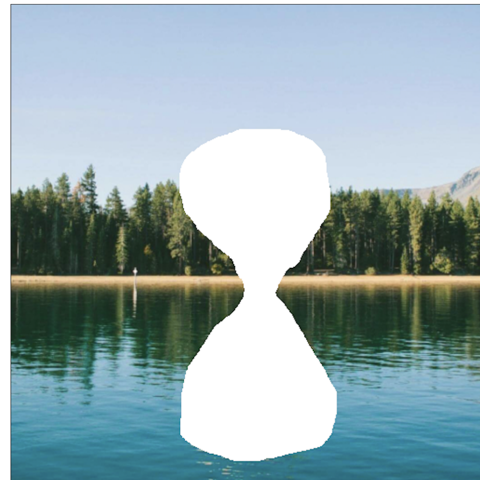
The E and M kernels are roughly balanced. `expectation_step_kernel` and `maximization_step_kernel` land in the middle of the range at **1.18** and **1.84 FLOPs/byte**. The E step in particular pushes the most absolute DRAM traffic of any kernel (28.6 MB), driven by streaming patch reads from both source and target combined with atomic writes into the shared vote buffer.

Randomize kernel's arithmetic intensity is as expected. `nnf_randomize_kernel` at **1.68 FLOPs/byte** indicates a small amount of arithmetic operations done per memory access. All it does is generate a pseudo-random offset and store it.

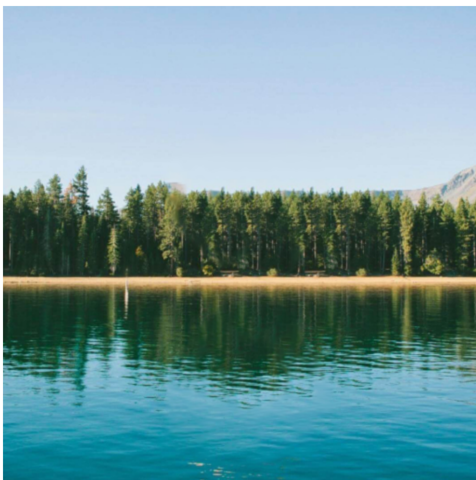
Overall Analysis Overall, the Table 1 indicates that we have reached a point of diminishing returns from low-level optimizations. The Jump-Flood kernel is already well optimized, while the others are all memory-bound.

4.4 Results on Other Inputs

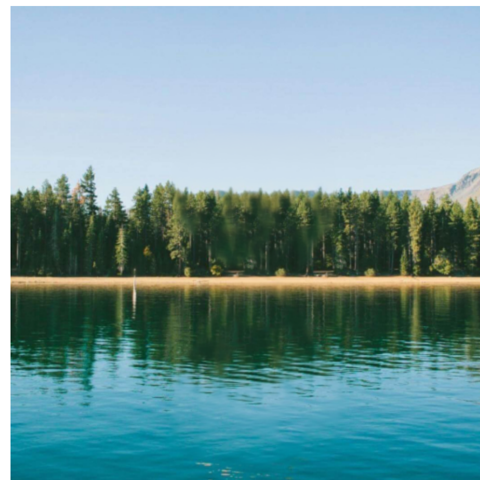
(a) Original image



(b) Masked Image



(c) Inpainted result (CPU)



(d) Inpainted result (GPU)

Figure 7: Hot air balloon inpainting results (patch size = 3).

(a) Original image

(b) Masked Image

(c) Inpainted result (CPU)

(d) Inpainted result (GPU)

Figure 8: House inpainting results (patch size = 3).

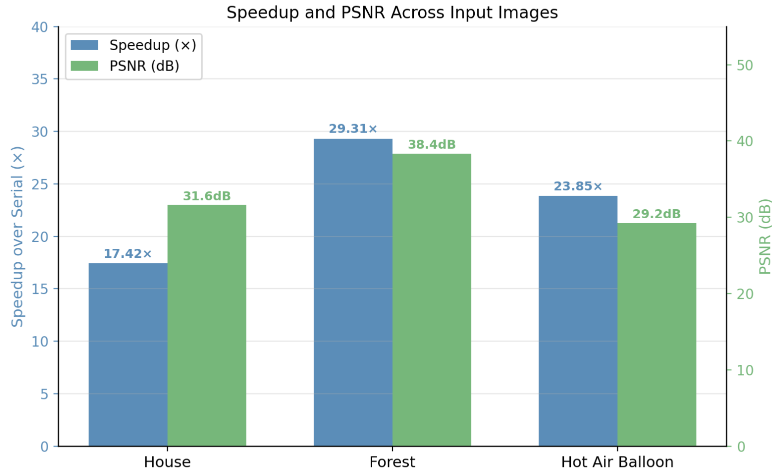


Figure 9: Speedup and PSNR across three input images at patch size 3. The images can be seen in Figures 3, 8 and 7

As seen in Figure 9, the GPU achieves roughly consistent speedup across all three images, subject to the image resolutions and hole sizes. This confirms that the gains are structural rather than image-specific. We see some variation across the PSNR metric with Forest scoring highest at 38.36 dB, while Hot Air Balloon and House are lower at 29.23 dB and 31.62 dB respectively. This reflects the texture complexity of each scene: large uniform regions like forest foliage are easier for PatchMatch to synthesise accurately, while structured content such as sky gradients and architectural edges introduce harder matching problems that reduce output fidelity. The primary quality objective was not absolute visual fidelity but consistency with the serial baseline. The PSNR scores and a visual inspection of the outputs confirm that the GPU implementation produces results that are close to the serial reference across all three images.

4.5 Opportunities for Future Work

Level	Run	Set ID	NNF	Upscale	EM	Init	Total
7	Serial	0.000	0.066	0.000	0.000	0.000	0.066
	GPU	0.000	0.003	0.000	0.001	0.000	0.004
6	Serial	0.000	0.000	0.000	0.002	0.000	0.003
	GPU	0.000	0.008	0.000	0.001	0.000	0.009
5	Serial	0.001	0.000	0.000	0.009	0.001	0.011
	GPU	0.000	0.015	0.000	0.001	0.000	0.017
4	Serial	0.002	0.230	0.000	0.034	0.003	0.269
	GPU	0.001	0.017	0.000	0.002	0.000	0.020
3	Serial	0.007	0.877	0.000	0.117	0.011	1.013
	GPU	0.001	0.017	0.000	0.007	0.001	0.026
2	Serial	0.019	1.915	0.002	0.367	0.056	2.358
	GPU	0.003	0.023	0.002	0.031	0.002	0.061
1	Serial	0.045	2.883	0.006	1.044	0.257	4.235
	GPU	0.006	0.040	0.006	0.127	0.007	0.187
0	Serial	0.059	2.072	0.000	0.898	1.115	4.144
	GPU	0.009	0.050	0.000	0.004	0.027	0.090
Total	**Serial**	0.133	8.044	0.008	2.472	1.442	**12.100**
	GPU	0.021	0.174	0.008	0.173	0.037	**0.413**

Table 2: Time spent (in seconds) in each operation across pyramid levels for the serial and GPU runs on the forest image (Figure 3c) at a patch size of 3. Per-operation values are aggregated across all EM iterations within a level. The end-to-end wall-clock speedup of the GPU run over the serial baseline is approximately $29.3\times$.

Table 2 highlights a few directions for further work. NNF minimization, which absorbed most of our optimization effort, dropped from 66% of the serial runtime to 38% of the GPU runtime, leaving the EM step as the next target to optimize. The two NNF minimize kernels (source-to-target and target-to-source) and the two expectation step kernels operate on independent data and could be fused into single launches, both halving the per-iteration launch count and improving cache reuse across what are currently separate kernels reading the same image buffers.

4.6 Conclusion

In this project, we aimed to implement a parallel image inpainting pipeline built on top of the PatchMatch algorithm, progressively parallelising each component of the EM loop across six versions. Starting from a serial C++ baseline, we ported the NNF minimisation to CUDA using red-black ordering, replaced it with jump-flood propagation for better GPU utilisation, moved field initialisation on-device, improved memory coalescing through packed `uchar4` buffers, and finally ported the E and M steps to CUDA kernels to eliminate repeated PCIe transfers. The final

version achieves a speedup of **29.47x** over the serial baseline, well beyond the **7x** target set by the original PatchMatch paper.

We made significant systems optimisations, however, we reached a point of diminishing returns. Once the E and M steps moved on-device, the remaining CPU-side work became negligible, and adding OpenMP parallelism contributed less than 1% additional speedup. Roofline analysis confirmed that most kernels operate at low arithmetic intensity and are memory-bound, meaning additional compute resources offer little benefit without a corresponding reduction in memory traffic. The atomic accumulation pattern in the E step is a structural bottleneck that cannot be eliminated without a fundamentally different vote aggregation scheme.

Looking ahead, the most promising directions are fusing the source-to-target and target-to-source kernels into single launches to reduce overhead, replacing the CPU upsampling step between pyramid levels with a GPU kernel to remove the last remaining host-device round trip, and exploring shared memory tiling within the patch distance computation to reduce redundant global memory reads.

5 Credit Distribution

60% of the work was done by Saileshwar Karthik, the remaining **40%** was done by Ayushi Bansal.

References

[1] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. PatchMatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 28(3), August 2009.

[2] A. Criminisi, P. Perez, and K. Toyama. Object removal by exemplar-based inpainting. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, volume 2, pages II–II, June 2003. doi: 10.1109/CVPR.2003.1211538.

[3] Guodong Rong and Tiow-Seng Tan. Jump flooding in gpu with applications to voronoi diagram and distance transform. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, page 109–116, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 159593295X. doi: 10.1145/1111411.1111431. URL `https://doi.org/10.1145/1111411.1111431`.

[4] Denis Simakov, Yaron Caspi, Eli Shechtman, and Michal Irani. Summarizing visual data using bidirectional similarity. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008. doi: 10.1109/CVPR.2008.4587842.

[5] Yonatan Wexler, Eli Shechtman, and Michal Irani. Space-time completion of video. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(3): 463–476, 2007. doi: 10.1109/TPAMI.2007.60.

[6] Pei Yu, Xiaokang Yang, and Li Chen. Parallel-friendly patch match based on jump flooding. In Wenjun Zhang, Xiaokang Yang, Zhixiang Xu, Ping An, Qizhen Liu, and Yue Lu, editors, *Advances on Digital Television and Wireless Multimedia Communications*, pages 15–21, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[7] Hanli Zhao, Heyang Guo, Xiaogang Jin, Jianbing Shen, Xiaoyang Mao, and Junru Liu. Parallel and efficient approximate nearest patch matching for image editing applications. *Neurocomputing*, 305:39–50, 2018. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2018.03.064. URL `https://www.sciencedirect.com/science/article/pii/S0925231218304703`.