High Performance Graphics (2010)
M. Doggett, S. Laine, and W. Hunt (Editors)

A Work-Efficient GPU Algorithm for Level Set Segmentation

Mike Roberts, Jeff Packer, Mario Costa Sousa, Joseph Ross Mitchell

University of Calgary, Canada

Abstract

We present a novel GPU level set segmentation algorithm that is both work-efficient and step-efficient. Our algorithm: (1) has linear work-complexity and logarithmic step-complexity, both of which depend only on the size of the active computational domain and do not depend on the size of the level set field; (2) limits the active computational domain to the minimal set of changing elements by examining both the temporal and spatial derivatives of the level set field; (3) tracks the active computational domain at the granularity of individual level set field elements instead of tiles without performance penalty; and (4) employs a novel parallel method for removing duplicate elements from unsorted data streams in a constant number of steps. We apply our algorithm to 3D medical images and we demonstrate that in typical clinical scenarios, our algorithm reduces the total number of processed level set field elements by 16× *and is* 14× *faster than previous GPU algorithms with no reduction in segmentation accuracy.*

1. Introduction

Identifying distinct regions in images – a task known as *segmentation* – is an important task in computer vision [OP03] and medical imaging [SPM*09]. The level set method is a powerful and flexible numerical technique for image segmentation under challenging conditions [Whi94] since the segmentation process can depend on both intrinsic factors (e.g. the curvature of the segmented regions) and extrinsic factors (e.g. the intensity or texture of the image). Lefohn et al. [LCW03] and Cates et al. [CLW04] showed that level set segmentation methods can reduce the variability of difficult segmentation tasks in medical imaging. However the flexibility of the level set method has historically resulted in long computation times and therefore limited clinical utility.

In this paper we describe a new GPU level set segmentation algorithm that dramatically improves computational efficiency without affecting segmentation accuracy. Our new algorithm results from two distinct contributions:

1. A method for limiting the active computational domain to the minimal set of changing elements by examining both the temporal and spatial derivatives of the level set field; and
2. A work-efficient [AF98] and step-efficient [NPGM00] mapping of this method to manycore GPU architectures that avoids traversing the entire level set field after initialization and employs a novel method of removing duplicate elements from unsorted data streams in a constant number of steps.

We describe our algorithm and demonstrate significant performance benefits over previous GPU algorithms [LCW03, LKHW03, CLW04, LKHW04]. In a series of controlled experiments using noisy magnetic resonance images (MRIs) generated from the BrainWeb Simulated Brain Database [SBD, KEP96, CKK*97, CZK*98, KEP99], we demonstrate that our algorithm: (1) reduces the total number of processed level set field elements by 16× and and converges 14× faster than previous GPU algorithms; and (2) produces equally accurate segmentations to previous GPU algorithms with less than 0.2% variability in all experiments. Our algorithm runs entirely on the GPU without requiring any additional data processing on the CPU, thereby enabling interactive 3D visualization and real-time control of the evolving level set surface.

2. Background and Previous Work

2.1. Level Set Segmentation

The level set method for image segmentation [Whi94] embeds an implicitly represented seed surface within an image, and then iteratively deforms the surface to envelop the

containing region-of-interest (ROI). Each implicitly defined point on the surface is deformed along a path normal to the local surface. Level set segmentation methods commonly use an application-specific speed function $F(\mathbf{x},t)$ to determine the local rate of surface motion, where $\mathbf{x}$ is a coordinate in the image and t is the current time in the simulation. For a more comprehensive review of level set methods and their applications to image segmentation, we refer the reader to Sethian [Set99], and Osher and Paragios [OP03].

In this paper we adopt the speed function proposed by Lefohn et al. [LCW03, LKHW03, LKHW04]. This function determines surface speed according to the local mean surface curvature and the local intensity of the image. By taking into account the level set surface's curvature, we encourage a smooth surface and prevent the surface from leaking into undesired areas across weak, incidental connections at ROI boundaries. For the scalar field $\phi(\mathbf{x},t) : \Re^4 \mapsto \Re$, we define our level set surface as $\{\mathbf{x} \mid \phi(\mathbf{x},t) = 0\}$.

As proposed by Lefohn et al. [LCW03, LKHW03, LKHW04], we define the data term of our speed function $D(\mathbf{x}) = \varepsilon - (|I(\mathbf{x})| - T)$ to be a function of the image intensity $I(\mathbf{x})$, the user-specified target intensity T that will encourage maximal surface growth, and a user-specified intensity window parameter ε within which the level set surface is encouraged to grow. If $I(\mathbf{x})$ is between $T-\varepsilon$ and $T+\varepsilon$, then $D(\mathbf{x})$ will encourage surface growth, otherwise $D(\mathbf{x})$ will encourage surface contraction. We define the curvature term of our speed function as $C(\mathbf{x},t) = \nabla \cdot \frac{\nabla\phi(\mathbf{x},t-\Delta t)}{|\nabla\phi(\mathbf{x},t-\Delta t)|}$, where $\nabla \cdot \frac{\nabla\phi(\mathbf{x},t-\Delta t)}{|\nabla\phi(\mathbf{x},t-\Delta t)|}$ is the local mean surface curvature of the level set field from the previous iteration.

We define our speed function as $F(\mathbf{x},t) = \alpha C(\mathbf{x},t) + (1-\alpha)D(\mathbf{x})$ where $\alpha \in [0,1]$ is a user-specified blending term that controls the relative influence of the curvature and data terms on the behavior of our speed function. For a detailed account of how this speed function can be implemented, we refer the reader to Lefohn et al. [LKHW04]. We express the level set field update equation as follows.

$$\phi(\mathbf{x},t) = \phi(\mathbf{x},t-\Delta t) + \Delta t F(\mathbf{x},t)\,|\nabla\phi(\mathbf{x},t-\Delta t)| \qquad (1)$$

Two distinct algorithms for efficiently solving Equation 1 are directly relevant to our work: The GPU narrow band algorithm [LCW03, LKHW03, CLW04, LKHW04, JBH*09] and the sparse field algorithm [Whi98, PMO*99].

2.2. The GPU Narrow Band Algorithm

The narrow band algorithm [Ada95] only computes level set field updates inside a small region (i.e. a narrow band) of elements around the implicitly defined level set surface. This algorithm has been successfully ported to the GPU [LCW03, LKHW03, CLW04, LKHW04, JBH*09] by using various virtual memory paging schemes to map the irregular and dynamic narrow band onto a physically contiguous domain better suited for GPU computation. These virtual memory paging schemes partition the level set field into tiles and map each active tile (i.e. each tile containing elements in the narrow band) to a contiguous physical block of GPU memory. GPU narrow band algorithms only perform level set computations on active tiles. This significantly improves performance and saves a large amount of GPU memory because inactive virtual tiles do not need to be stored on the GPU. In turn, this allows for images larger than the size of available GPU memory to be segmented.

Until very recently, GPU narrow band algorithms [LCW03, LKHW03, CLW04, LKHW04] have maintained the narrow band by using parallel data reduction techniques on the GPU in cooperation with the CPU. The GPU generates a down-sampled memory image of active tiles which is subsequently downloaded by the CPU at the end of each iteration. The CPU is responsible for traversing the down-sampled image, determining the narrow band for the next iteration, and communicating this result back to the GPU. Since the CPU must sequentially traverse the down-sampled image when updating the narrow band, these algorithms have $O(n)$ work-complexity [AF98] and step-complexity [NPGM00] where n is the size of the active computational domain. Moreover since the CPU work for each iteration cannot begin until the GPU work is finished and vice versa, these algorithms are limited by the communication latency between the GPU and CPU.

Developed in parallel with our own work, the GPU narrow band algorithm very recently described by Jeong et al. [JBH*09] avoids the communication latency inherent in previous GPU narrow band algorithms by traversing the domain of active tiles in parallel on the GPU. When a thread determines that a new tile should become part of the narrow band, the thread appends that tile to a list of active tiles stored in GPU memory. Since many threads are potentially appending to this list in parallel, Jeong et al. use GPU atomic memory operations to serialize access to the list. Therefore this system also has $O(n)$ work-complexity and step-complexity.

2.3. The Sparse Field Algorithm

In contrast to the GPU narrow band algorithm, the sparse field algorithm [Whi98, PMO*99] incrementally updates a linked list of active elements on the CPU during each iteration. Therefore this algorithm's work-complexity is $O(n)$. The sparse field algorithm also reduces its total amount of work by a constant factor by tracking the active computational domain at the granularity of individual elements instead of tiles. Historically the sparse field algorithm has been poorly suited to parallel implementation due to its reliance on linked lists. No parallel implementation of the sparse field algorithm currently exists in the literature. Therefore its step-complexity is equal to its work-complexity.

GPU narrow band systems [LCW03, LKHW03, CLW04, LKHW04, JBH*09] have historically outperformed optimized sequential sparse field systems. Nonetheless the GPU

narrow band system we tested took over 100 seconds to converge on the white and grey matter in a 256^3 MRI of a human head on a state-of-the-art GPU. This limitation constrains clinical applications and motivates our work-efficient algorithm, which leverages ideas from both the GPU narrow band algorithm and the sparse field algorithm. We begin the presentation of our algorithm by describing our novel method for tracking the active computational domain.

3. A Temporally Coherent Active Computational Domain

The narrow band and sparse field algorithms described in the previous section avoid unnecessary computation by only updating field elements near the level set surface. We make the observation that even computations near the level set surface can be avoided in regions where the level set field has locally converged. This observation motivates our method of tracking the active computational domain according to both the temporal and spatial derivatives of the level set field.

We define the minimal set of active coordinates at time t as $A(t) = \{\mathbf{x} \mid \phi(\mathbf{x},t) \neq \phi(\mathbf{x},t-\Delta t)\}$. From Equation 1 we derive two conditions, each of which is sufficient to imply that $\mathbf{x} \notin A(t)$. Both of these conditions are independent of our speed function and could therefore be applied to a variety of level set simulations.

The first condition ς_1 follows directly from Equation 1 and can be expressed as follows: $\varsigma_1(\mathbf{x},t) \equiv |\nabla\phi(\mathbf{x},t-\Delta t)| = 0$. We note that ς_1 is described by Lefohn et al. [LKHW03, LKHW04].

The second condition ς_2 requires a more detailed discussion. We define the set $\eta(\mathbf{x})$ as the set of all coordinates in the immediate neighborhood of $\mathbf{x}$ (including $\mathbf{x}$ itself). We observe that if $\phi(\mathbf{x},t-\Delta t) = \phi(\mathbf{x},t-2\Delta t)$, then $\frac{\Delta\phi(\mathbf{x})}{\Delta t} = 0$. In other words ϕ is in a state of temporal equilibrium at $\mathbf{x}$. Assuming the speed function is defined locally, the only event that could potentially disrupt this state of temporal equilibrium at $\mathbf{x}$ is if $\phi(\mathbf{n})$ changes for some neighbor $\mathbf{n} \in \eta(\mathbf{x})$. If the level set field is in a state of temporal equilibrium in the neighborhood around $\mathbf{x}$ at time $t-2\Delta t$, then $\mathbf{x}$ will continue to be in a state of temporal equilibrium at time $t-\Delta t$. This leads to the following expression for ς_2: $\varsigma_2(\mathbf{x},t) \equiv \forall_{\mathbf{n}\in\eta(\mathbf{x})} : \phi(\mathbf{n},t-\Delta t) = \phi(\mathbf{n},t-2\Delta t)$.

We include a more formal derivation of ς_2 in Appendix A. For the logical *not* operator $\neg$ we formally express our active set as follows:

$$A(t) = \begin{cases} \text{Domain}(\phi) & t = 0 \\ \{\mathbf{x} \mid \neg\varsigma_1(\mathbf{x},t)\} & t = \Delta t \\ \{\mathbf{x} \mid \neg\varsigma_1(\mathbf{x},t)\} \cap \{\mathbf{x} \mid \neg\varsigma_2(\mathbf{x},t)\} & t > \Delta t \end{cases} \quad (2)$$

A diagram describing our method of tracking the active computational domain is shown in Figure 1.

```
1: for all coordinates x ∈ Domain(φ) in parallel do
2:     φ_x^read ← clamp(‖x − c‖ − r)
3:     φ_x^write ← clamp(‖x − c‖ − r)
```

Listing 1: Initializing the level set field to the signed and clamped distance transform relative to a user-specified seed sphere with the center $\mathbf{c}$ and the radius r.

4. A Work-Efficient Parallel Algorithm

We want to track the active computational domain and perform updates on the level set field in a way that is both work-efficient and step-efficient. In other words we want a parallel algorithm with work-complexity and step-complexity of at most $O(n)$. This upper bound on work-complexity and step-complexity motivates the algorithm described in this section. A high level overview of our algorithm is as follows:

1. Initialize the level set field and generate a dense list of active coordinates.
2. Update the level set field at all active coordinates.
3. Generate new active coordinates. During this step generating duplicate active coordinates is permitted.
4. Remove all duplicate active coordinates generated in (3).
5. Compact all the unique new active coordinates from (4) into a new dense list.
6. If there are no active coordinates in the new dense list, the segmentation has globally converged. Otherwise go to (2).

4.1. Assumptions

We assume ϕ is 3D and the voxels in ϕ are 6-connected. Therefore it is guaranteed that for all voxels $\mathbf{x} \in \text{Domain}(\phi)$, the set $\eta(\mathbf{x})$ contains at most seven elements: the 6-connected neighbors of $\mathbf{x}$ and $\mathbf{x}$ itself. We define the set $E = \{(0,0,0),(\pm1,0,0),(0,\pm1,0),(0,0,\pm1)\}$ as the set of offset vectors from a voxel to its 6-connected neighbors.

4.2. Data Structures and Notation

Our algorithm requires three 3D buffers: ϕ^{write} and ϕ^{read} to store the current and previous level set field respectively; and U to use as a scratchpad. Our algorithm also requires eight 1D buffers: V to store the current dense list of active coordinates; and $B^{(0,0,0)}$, $B^{(\pm1,0,0)}$, $B^{(0,\pm1,0)}$, and $B^{(0,0,\pm1)}$ to use as auxiliary buffers when generating new active coordinates. The size of each buffer is equal to the size of the entire level set field. All buffers are initially filled with null values.

In our implementation the data type of ϕ^{write} and ϕ^{read} is 32-bit floating point and the data type of all other buffers is 32-bit integer. To maximize memory efficiency when storing 3D level set field coordinates in these integer buffers, we pack the x, y, and z components of each 3D coordinate into 11, 11, and 10 bits respectively. However if the level set field contains more than $2048 \times 2048 \times 1024 = 2^{32}$ elements, the

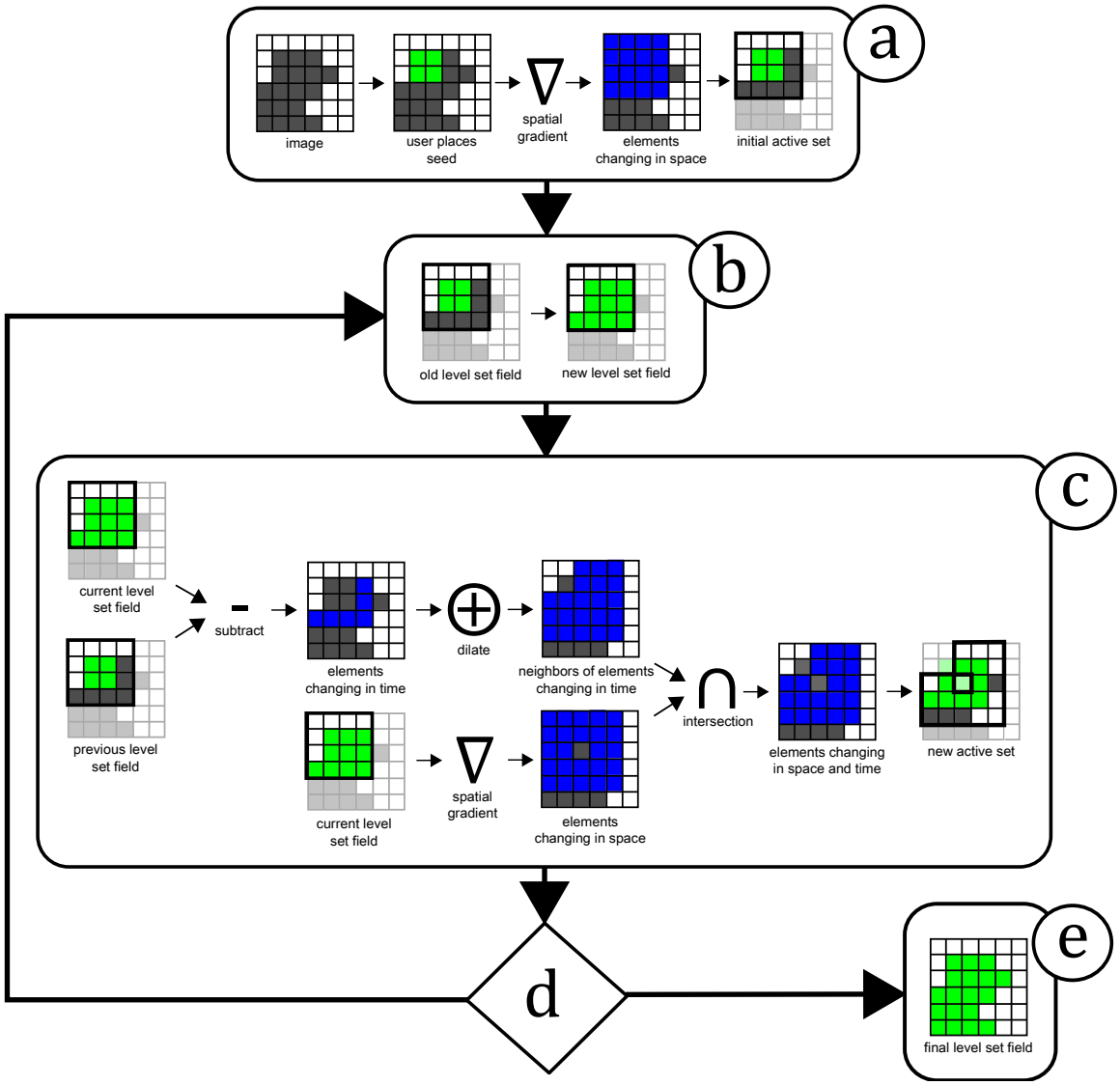


Figure 1: *Our algorithm for tracking the active computational domain. Image data is shown in grey, currently segmented regions are shown in green, and intermediate results for computing the active computational domain are shown in blue. The active computational domain is outlined in black, and inactive elements are shown as partially transparent. The user places a seed to initialize the level set field and the initial active computational domain is determined according to the spatial derivatives of the level set field (a). During each iteration the level set field is updated at all active elements (b). The new active computational domain is computed according to the temporal and spatial derivatives of the level set field (c). If the new active computational domain is empty (d) then our segmentation has globally converged (e). Otherwise we go to (b).*

data type of each integer buffer must be expanded such that the largest possible level set field coordinate can fit into a single element.

We use a subscript notation to refer to individual buffer elements. For example V_i refers to the i^{th} element of V; $V_{j...k}$ refers to the range of elements in V from V_j to V_k; and $U_{\mathbf{x}}$ refers to the element of U with the 3D coordinates $\mathbf{x}$.

4.3. Initialization

We initialize in parallel every coordinate in ϕ^{read} and ϕ^{write} according to a user-specified seed region as shown in Listing 1. For more details on this initialization step we refer the reader to Lefohn et al. [LKHW04].

We then generate a densely packed buffer of active coor-

```
1: for all coordinates x ∈ Domain(φ) in parallel do
2:     g ← false
3:     for all coordinates n ∈ η(x) do
4:         if not g then
5:             if φ_x^read ≠ φ_n^read then
6:                 g ← true
7:     if g then
8:         U_x ← x
9: V ← compact(U)
```

Listing 2: Initializing the list of active coordinates. The spatial derivative of the level set field is tested on line 5.

```
1: for all coordinates v ∈ V_{0...n} in parallel do
2:     φ_v^write ← φ_v^read + ΔtF(v,t) |∇φ_v^read|
3:     U_v ← null
4: swap(φ^read, φ^write)
```

Listing 3: Updating the level set field according to Equation 1 and clearing the scratchpad at all active coordinates. n is the current size of the active computational domain.

dinates V based on the contents of ϕ^{read} as shown in Listing 2. We test every coordinate of ϕ^{read} in parallel to determine which ones are active. If a coordinate is deemed active according to Equation 2, we write that coordinate to our 3D scratchpad U using the coordinate itself as the 3D array index. We compact U in parallel to produce V. We set the initial size of the active computational domain n to be the number of coordinates that were compacted into V. For more details on this buffer compacting operation we refer the reader to Harris et al. [HSO07] and Sengupta et al. [SHZO07, SHG08]. We note that since U contains either a unique value or null at all coordinates, it is guaranteed that there are no duplicate coordinates in $V_{0...n}$. At this point our algorithm has been fully initialized. We avoid traversing the entire level set field for the remainder of our algorithm.

4.4. Updating the Level Set Field

During each iteration we update ϕ^{write} and clear U at all active coordinates as shown in Listing 3. This is guaranteed to completely clear U because the only non-null values in U are at active coordinates, each of which was previously compacted into V. At this point we are finished updating the level set field for this iteration, so we swap references to ϕ^{read} and ϕ^{write}.

4.5. Generating New Active Coordinates

We traverse our current list of active coordinates in parallel and test for new active coordinates according to Equation 2. We describe this process in Listing 4 and Figure 2. We test ς_1 and ς_2 for the next iteration by examining $\phi^{read} = \phi(t)$ and $\phi^{write} = \phi(t - \Delta t)$. We note that at the end of this step, the

```
1: for h ← 0 to n in parallel do
2:     v ← V_h
3:     g ← false
4:     for all coordinates n ∈ η(v) do
5:         if φ_n^read ≠ φ_n^write and φ_n^read ≠ φ_v^read then
6:             g ← true
7:             e ← n − v
8:             B_h^e ← n
9:     if g then
10:        B_h^(0,0,0) ← v
```

Listing 4: Generating new active coordinates. The temporal and spatial derivatives of the level set field are tested on line 5. n is the current size of the active computational domain.

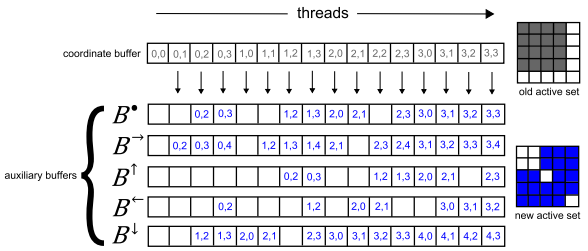


Figure 2: *Generating new active coordinates. There may be duplicate coordinates in the auxiliary buffers when taken collectively. All duplicate coordinates must subsequently be removed (see Figure 3).*

seven auxiliary buffers may contain duplicate coordinates when taken collectively. This is because each thread tests a local neighborhood of coordinates and some coordinates may be tested repeatedly by different threads.

4.6. Removing Duplicate Active Coordinates

We make the observation that although there may be duplicate coordinates in the seven auxiliary buffers taken collectively, it is guaranteed that there are no duplicate coordinates in each of the seven auxiliary buffers taken individually. This is because there are no duplicate coordinates in $V_{0...n}$ and for all offset vectors $\mathbf{e} \in E$, either $B_i^{\mathbf{e}} = V_i + \mathbf{e}$ or $B_i^{\mathbf{e}} = \text{null}$ for all array indices i where $0 \leq i \leq n$.

Based on the guarantee in the previous paragraph, we are able to remove all duplicate coordinates in seven passes without requiring any additional sorting or synchronization primitives. We describe this process in Listing 5 and Figure 3.

4.7. Compacting the New Active Coordinates

We compact the seven auxiliary buffers in parallel to produce a new dense list of active coordinates and store the result in

```
1:  E' ← E − {(0,0,0),(0,0,1)}
2:  for all coordinates b ∈ B^(0,0,0)_0...n in parallel do
3:      if b ≠ null then
4:          U_b ← tagged
5:  for all offset vectors e ∈ E' do
6:      for h ← 0 to n in parallel do
7:          b ← B^e_h
8:          if b ≠ null then
9:              if U_b = tagged then
10:                 B^e_h ← null
11:             else
12:                 U_b ← tagged
13: for h ← 0 to n in parallel do
14:     b ← B^(0,0,1)_h
15:     if b ≠ null and U_b = tagged then
16:         B^(0,0,1)_h ← null
17: V ← compact(B^(0,0,0)_0...n, B^(±1,0,0)_0...n, B^(0,±1,0)_0...n, B^(0,0,±1)_0...n)
```

Listing 5: Generating a new dense list of unique active coordinates without sorting the auxiliary buffers. n is the current size of the active computational domain.

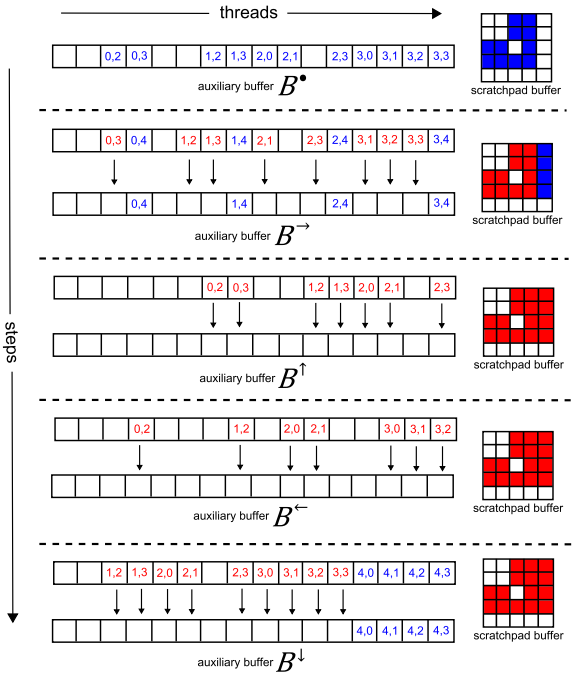


Figure 3: *Removing duplicate coordinates from the auxiliary buffers in parallel without sorting. Coordinates that have not been previously tagged in the scratchpad buffer are shown in blue. Coordinates that have been previously tagged in the scratchpad buffer are shown in red, and are removed from their containing auxiliary buffer. This process is free of race conditions because each step examines one auxiliary buffer and there are no duplicate coordinates within each auxiliary buffer.*

V as shown in Listing 5. Since we only ever write to the first n elements of each auxiliary buffer, we only need to compact $7n$ elements in total, rather than compacting the total size of each buffer. In order to further improve the efficiency of this buffer compacting step, we allocate the auxiliary buffers dynamically at the beginning of each iteration by partitioning a larger pre-allocated buffer.

After compacting the seven auxiliary buffers, we check if any new active coordinates were compacted into V. If so, we clear $B^{\mathbf{e}}_{0\ldots n}$ for all offset vectors $\mathbf{e} \in E$, update n to be the number of new active coordinates that were compacted into V, and go to the step described in section 4.4. Otherwise our algorithm has globally converged on the segmented region contained in ϕ^{read}.

4.8. Memory Efficiency

All memory accesses to the 1D buffers V, $B^{(0,0,0)}$, $B^{(\pm1,0,0)}$, $B^{(0,\pm1,0)}$, and $B^{(0,0,\pm1)}$ are fully coalesced. These buffers are always accessed via a unique thread index such that neighboring threads always access neighboring locations in memory.

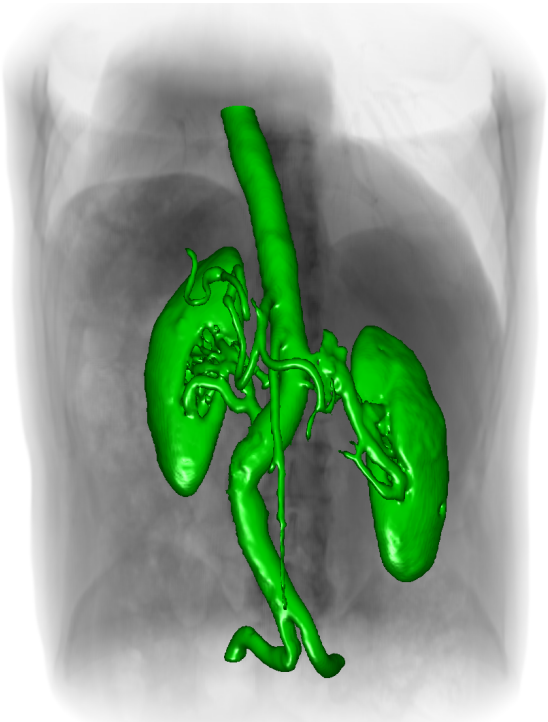
In general none of the memory accesses to the 3D buffers ϕ^{read}, ϕ^{write}, and U can be coalesced. After initialization these buffers are always accessed at sparse active coordinates which are not guaranteed to be neighboring for neighboring threads. Therefore it is not clear how to take advantage of GPU shared memory as a programmer-managed cache for level set computations using our algorithm. For this reason our implementation does not use GPU shared memory.

However since there is some spatial locality in the active computational domain, it is probable but not guaranteed that nearby active coordinates in V are nearby in 3D space. We leverage this spatial locality by reading from our 3D buffers via the hardware-managed texture cache. To improve multi-dimensional cache coherence, we use a simple swizzled memory layout as described by Engel et al [EHK*06].

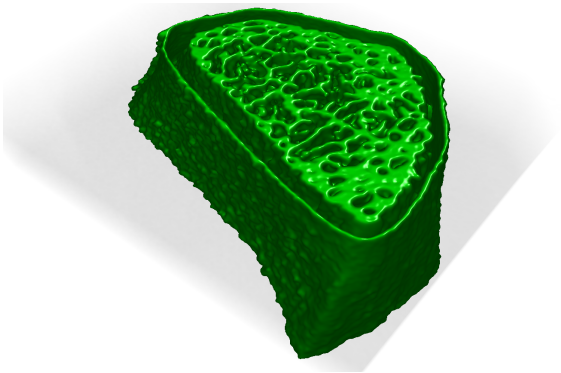
5. Evaluation

5.1. Algorithmic Complexity

The algorithm we use for buffer compacting has $O(w)$ work-complexity and $O(\log_2 w)$ step-complexity where w is the size of the input [HSO07, SHZO07, SHG08]. Therefore our algorithm has $O(p)$ work-complexity and $O(\log_2 p)$ step-complexity during initialization where p is the size of the entire level set field. After initialization our algorithm has $O(n)$ work-complexity and $O(\log_2 n)$ step-complexity where n is the size of the active computational domain. From this



(a) the aorta and kidneys in a $256 \times 256 \times 272$ abdominal CT image – total computation time 16 seconds



(b) the cortical bone and trabecular bone in a $288 \times 352 \times 112$ wrist CT image – total computation time 12 seconds

Figure 4: *Segmentation results from our system.*

we conclude that our algorithm is both work-efficient and step-efficient.

Our algorithm requires $O(p)$ memory. In other words the memory requirements of our algorithm increase linearly with the size of the level set field rather than the surface area of the segmented surface, as is the case with previous GPU algorithms [LCW03, LKHW03, CLW04, LKHW04, JBH*09].

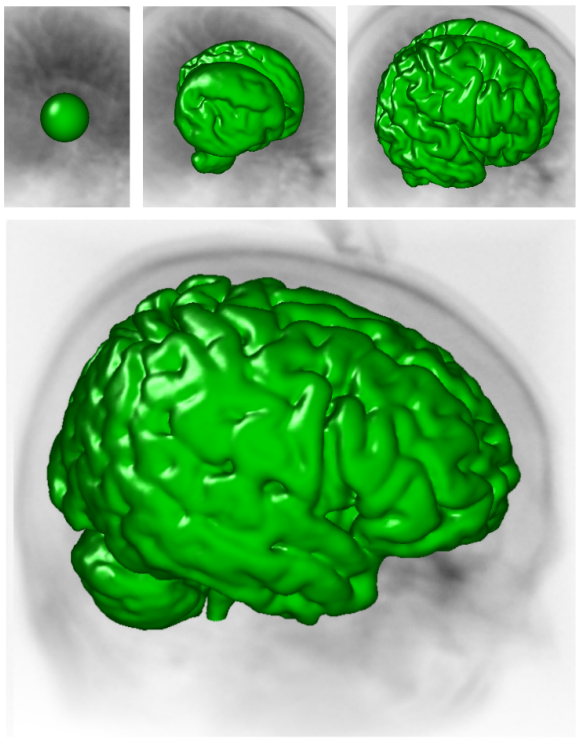


Figure 5: *The progression of our algorithm while segmenting the white and grey matter in a 256^3 head MRI with a signal-to-noise ratio of 11 – total computation time 7 seconds.*

5.2. Experimental Methodology

We performed all our experiments on an Intel 2.5 gigahertz Xeon Processor with 4 gigabytes of memory and an NVIDIA GTX 280 GPU. We implemented our algorithm using CUDA [NVI10] and we used the CUDA Data Parallel Primitives Library [CUD] for buffer compacting. We implemented the GPU narrow band algorithm using OpenGL [SWND05] and GLSL [Ros06] with a tile size of 16^2, as described by Lefohn et al. [LCW03, LKHW03, LKHW04]. We feel this is the fairest method of evaluating each algorithm since the GPU narrow band algorithm relies on hardware features not exposed in CUDA (e.g. direct write access to texture memory). Likewise our algorithm makes use of hardware features not exposed in OpenGL and GLSL (e.g. random write access to global memory).

We performed various segmentation tasks on volumetric images generated from the BrainWeb Simulated Brain Database [SBD, KEP96, CKK*97, CZK*98, KEP99]. This database can be used to generate simulated head MRIs with a variety of realistic noise characteristics in a controlled setting where the ground truth classification of each voxel is known. We repeated these segmentation tasks using our al-

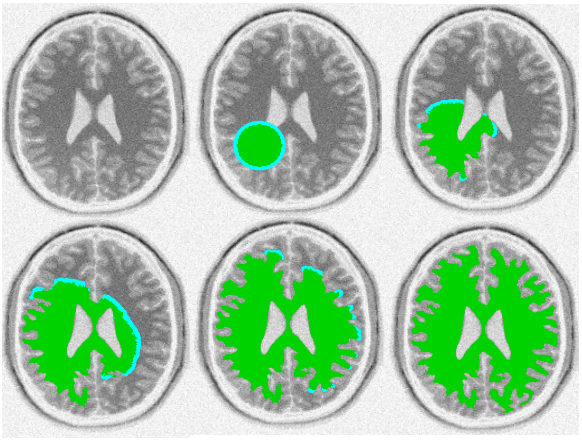


Figure 6: *The progression of the active computational domain (shown in blue) while segmenting the white matter in a 256^3 head MRI. Regions that have locally converged are immediately marked as inactive due to our analysis of the temporal and spatial derivatives of the level set field. The size of the active computational domain drops to zero when the segmentation has globally converged.*

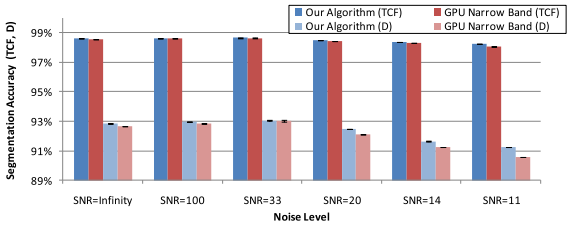


Figure 7: *Accuracy of our algorithm and the GPU narrow band algorithm while performing a set of repeated (N=10) white matter segmentations in a 256^3 head MRI with varying signal-to-noise-ratio (SNR) values. For each segmentation we used a randomly selected seed point and we measured the Dice Coefficient (D) and Total Correct Fraction (TCF).*

gorithm, the GPU narrow band algorithm, and a level set solver implemented in CUDA that unconditionally updates the entire level set field. We also performed various segmentation tasks on a $256 \times 256 \times 272$ abdominal CT image and a $288 \times 352 \times 112$ wrist CT image using our algorithm.

6. Results and Discussion

Figure 4 shows qualitatively accurate segmentations produced with our algorithm. Figure 5 shows the progression of our algorithm while segmenting the white and grey matter in a 256^3 head MRI with $SNR = 11$. Figure 6 shows the progression of the active computational domain while segmenting the white matter in the same MRI.

We compare the accuracy of our algorithm and the GPU narrow band algorithm in Figure 7. We observed that our algorithm was slightly more accurate than the GPU narrow band algorithm with less than 0.2% variability in all experiments. We speculate that this slight accuracy improvement is due to different floating point precision semantics in CUDA and GLSL. The accuracies we observed are comparable to those reported by Lefohn et al. [LCW03] and Cates et al. [CLW04].

We show the performance of our algorithm and the GPU narrow band algorithm in Figure 8. We observed that with our algorithm, the number of active voxels quickly peaked and decreased over time due to our analysis of the temporal and spatial derivatives of the level set field. With the GPU narrow band algorithm, the number of active voxels monotonically increased over time (Figure 8(a)). We observed that after the number of active voxels had peaked, the speed of our algorithm increased over time in contrast to the GPU narrow band algorithm (Figure 8(b)).

Based on the observed linear relationships between the number of active voxels per iteration and the computation time per iteration (Figure 8(c)), we conclude that the computational domain would need to be roughly 12% active before unconditionally updating every voxel would provide a performance benefit over our algorithm. At first glance it may seem as though the GPU narrow band algorithm would provide a performance benefit over our algorithm after the computational domain is roughly 9% active. However due to the GPU narrow band algorithm processing data in 2D tiles of size g^2, an active computational domain of size n using our algorithm will result in a larger active computational domain of size q where $n \leq q \leq g^2 n$ using the GPU narrow band algorithm. We observed that the computational domain remained less than 2% active during each iteration of our algorithm in all experiments.

We observed that tracking the active computational domain using our algorithm accounted for 77% of the total computation time and updating the level set field accounted for the remaining 23% of the total computation time (Figure 8(d)). We conclude that although most of our computation time goes into tracking the active computational domain, we leverage this cost to avoid the bigger downstream cost of unconditionally updating the entire level set field.

7. Conclusions

We have presented a new GPU level set segmentation algorithm with immediate applications in computer vision and medical imaging. Our algorithm is the first and only GPU level set segmentation algorithm to be presented in the literature with linear work-complexity and logarithmic step-complexity. Moreover our algorithm makes use of a novel

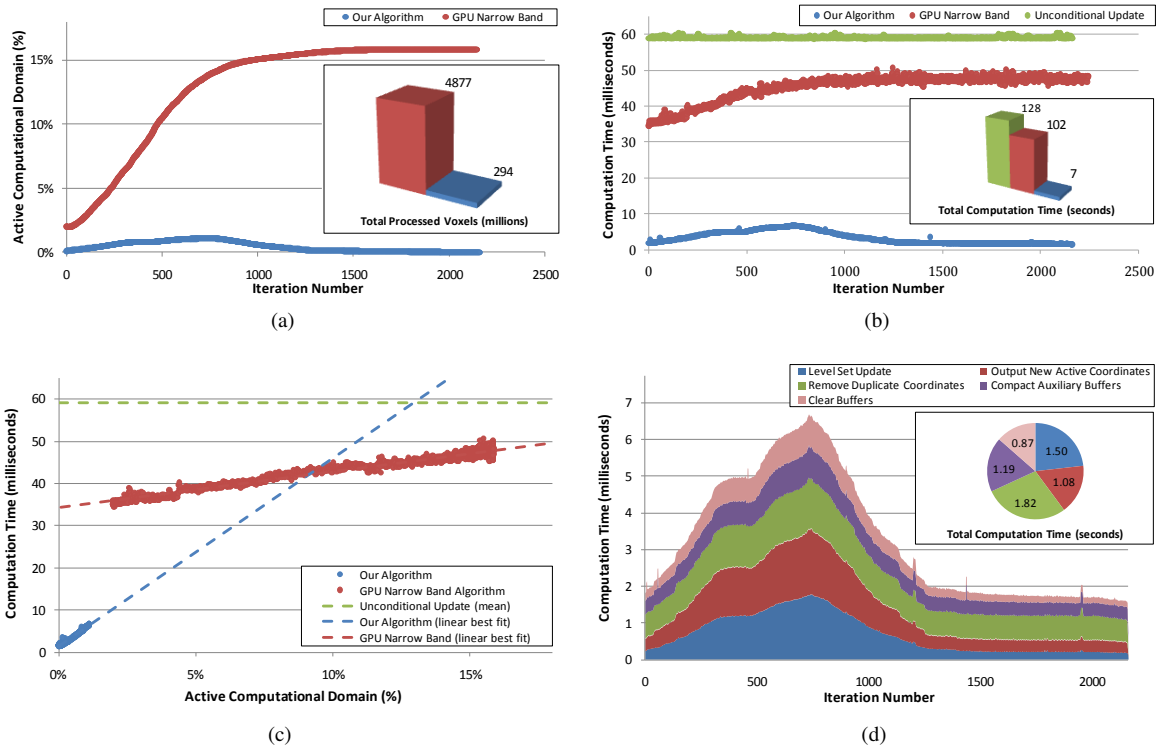


Figure 8: *Performance of our algorithm and the GPU narrow band algorithm while segmenting the white and grey matter in a 256^3 head MRI. We measured the size of the active computational domain per iteration (a), computation time per iteration (b), computation time as a function of active computational domain size (c), and the computation time per iteration for each subroutine in our algorithm (d). In (c) we overlay the lines of best fit for each algorithm. Lower is better for all graphs.*

condition on the temporal derivatives of the level set field to limit the active computational domain to the minimal set of changing elements. These innovations improve computational efficiency without affecting segmentation accuracy and create new possibilities for clinical application where speed and interactivity are critical.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions. This research was supported by the iCORE/Calgary Scientific Inc. Industrial Research Chair in Medical Imaging Informatics, Alberta Innovates – Health Solutions, the Alberta Heritage Foundation for Medical Research Endowment Fund, the iCORE/Foundation CMG Industrial Research Chair in Scalable Reservoir Visualization, and the Discovery Grants Program from the Natural Sciences and Engineering Research Council of Canada.

References

[Ada95] ADALSTEINSSON D.: A fast level set method for propagating interfaces. *Journal of Computational Physics 118*, 2 (1995), 269–277. 2

[AF98] ATALLAH M. J., FOX S. (Eds.): *Algorithms and Theory of Computation Handbook*. CRC Press, Inc., Boca Raton, FL, USA, 1998. Produced By-Lassandro, Suzanne. 1, 2

[CKK*97] COCOSCO C., KOLLOKIAN V., KWAN R., PIKE B., EVANS A.: Brainweb: Online interface to a 3D MRI simulated brain database. *NeuroImage 5*, 4 (1997), 425. 1, 7

[CLW04] CATES J., LEFOHN A., WHITAKER R.: GIST: an interactive, GPU-based level set segmentation tool for 3D medical images. *Medical Image Analysis 8*, 3 (2004), 217–231. 1, 2, 7, 8

[CUD] CUDPP: CUDA data parallel primitives library. http://gpgpu.org/developer/cudpp. 7

[CZK*98] COLLINS D. L., ZIJDENBOS A. P., KOLLOKIAN V., SLED J. G., KABANI N. J., HOLMES C. J., EVANS A. C.: Design and construction of a realistic digital brain phantom. *IEEE Transactions on Medical Imaging 17*, 3 (1998), 463–468. 1, 7

[EHK*06] ENGEL K., HADWIGER M., KNISS J., REZK-SALAMA C., WEISKOPF D.: *Real-Time Volume Graphics*. A K Peters, 2006. 6

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel prefix sum (scan) with CUDA. In *GPU Gems 3*, Nguyen H., (Ed.). Addison Wesley, Aug. 2007. 5, 6

[JBH*09] JEONG W.-K., BEYER J., HADWIGER M., VAZQUEZ A., PFISTER H., WHITAKER R. T.: Scalable and interactive segmentation and visualization of neural processes in EM datasets.

IEEE Transactions on Visualization and Computer Graphics 15, 6 (2009), 1505–1514. 2, 7

[KEP96] KWAN R. K.-S., EVANS A. C., PIKE G. B.: An extensible MRI simulator for post-processing evaluation. In *Proc. of 4th Intl. Conference on Visualization in Biomedical Computing (VBC '96)* (1996), pp. 135–140. 1, 7

[KEP99] KWAN R. K. S., EVANS A. C., PIKE G. B.: MRI simulation-based evaluation of image-processing and classification methods. *IEEE Transactions on Medical Imaging 18*, 11 (1999), 1085–1097. 1, 7

[LCW03] LEFOHN A., CATES J. E., WHITAKER R. T.: Interactive, GPU-based level sets for 3D segmentation. In *Proc. of Medical Image Computing and Computer Assisted Intervention (MICCAI '03* (2003), pp. 564–572. 1, 2, 7, 8

[LKHW03] LEFOHN A. E., KNISS J. M., HANSEN C. D., WHITAKER R. T.: Interactive deformation and visualization of level set surfaces using graphics hardware. In *Proc. of 14th IEEE Visualization 2003 (VIS '03)* (2003), p. 11. 1, 2, 3, 7

[LKHW04] LEFOHN A. E., KNISS J. M., HANSEN C. D., WHITAKER R. T.: A streaming narrow-band algorithm: Interactive computation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics 10*, 4 (2004), 422–433. 1, 2, 3, 4, 7

[NPGM00] NYLAND L. S., PRINS J. F., GOLDBERG A., MILLS P. H.: A design methodology for data-parallel applications. *IEEE Transactions on Software Engineering 26*, 4 (2000), 293–314. 1, 2

[NVI10] NVIDIA: NVIDIA CUDA programming guide, January 2010. 7

[OP03] OSHER S., PARAGIOS N.: *Geometric Level Set Methods in Imaging, Vision, and Graphics*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. 1, 2

[PMO*99] PENG D., MERRIMAN B., OSHER S., ZHAO H., KANG M.: A PDE-based fast local level set method. *Journal of Computational Physics 155*, 2 (1999), 410–438. 2

[Ros06] ROST R.: *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2006. 7

[SBD] SBD: Brainweb: Simulated brain database. http://www.bic.mni.mcgill.ca/brainweb/. 1, 7

[Set99] SETHIAN J. A.: *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press, 1999. 2

[SHG08] SENGUPTA S., HARRIS M., GARLAND M.: Efficient parallel scan algorithms for GPUs. NVIDIA Technical Report, September 2008. 5, 6

[SHZO07] SENGUPTA S., HARRIS M., ZHANG Y., OWENS J. D.: Scan primitives for GPU computing. In *Graphics Hardware 2007* (Aug. 2007), ACM, pp. 97–106. 5, 6

[SPM*09] SHATTUCK D. W., PRASAD G., MIRZA M., NARR K. L., TOGA A. W.: Online resource for validation of brain segmentation methods. *NeuroImage 45*, 2 (2009), 431–439. 1

[SWND05] SHREINER D., WOO M., NEIDER J., DAVIS T.: *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, 2005. 7

[Whi94] WHITAKER R. T.: Volumetric deformable models: Active blobs. In *Visualization in Biomedical Computing* (1994), pp. 122–134. 1

[Whi98] WHITAKER R. T.: A level-set approach to 3D reconstruction from range data. *International Journal of Computer Vision 29*, 3 (1998), 203–231. 2

Appendix A: Derivation of the Active Set Condition ς_2

We define the set of all user-specified parameters to the speed function as H and the user-specified image as I. We define $\eta(\mathbf{x}) = \{\mathbf{x}, \mathbf{n}_0^{\mathbf{x}}, \mathbf{n}_1^{\mathbf{x}}, \mathbf{n}_2^{\mathbf{x}}, \ldots, \mathbf{n}_k^{\mathbf{x}}\}$ to be the set of coordinates in the immediate neighborhood of some voxel $\mathbf{x}$. We define the set of all of level set field values in the immediate neighborhood of $\mathbf{x}$ at time t as $\Phi(\mathbf{x},t) = \{\phi(\mathbf{x},t), \phi(\mathbf{n}_0^{\mathbf{x}},t), \phi(\mathbf{n}_1^{\mathbf{x}},t), \phi(\mathbf{n}_2^{\mathbf{x}},t), \ldots, \phi(\mathbf{n}_k^{\mathbf{x}},t)\}$. We assume that the speed function $F(\mathbf{x},t)$ is a function of the level set values around $\mathbf{x}$ during the previous iteration $\Phi(\mathbf{x},t-\Delta t)$, the image I, and the set of user-specified parameters H. We assume without loss of generality that $\Delta t \neq 0$.

We want to prove that $\forall_{\mathbf{n}\in\eta(\mathbf{x})} : \phi(\mathbf{n},t-\Delta t) = \phi(\mathbf{n},t-2\Delta t)$ implies $\phi(\mathbf{x},t) = \phi(\mathbf{x},t-\Delta t)$. If this claim is true, it means that we can exclude $\mathbf{x}$ from our active set at time t if $\forall_{\mathbf{n}\in\eta(\mathbf{x})} : \phi(\mathbf{n},t-\Delta t) = \phi(\mathbf{n},t-2\Delta t)$. We begin by proving a useful lemma.

Lemma 1 $\Phi(\mathbf{x},t-\Delta t) = \Phi(\mathbf{x},t-2\Delta t)$ implies $F(\mathbf{x},t) = F(\mathbf{x},t-\Delta t)$.

Proof 1 We assume $\Phi(\mathbf{x},t-\Delta t) = \Phi(\mathbf{x},t-2\Delta t)$. By definition $F(\mathbf{x},t) = f(\Phi(\mathbf{x},t-\Delta t),I,H)$ for some function f. Therefore we get $F(\mathbf{x},t-\Delta t) = f(\Phi(\mathbf{x},t-2\Delta t),I,H) = f(\Phi(\mathbf{x},t-\Delta t),I,H) = F(\mathbf{x},t)$. □

Now we move onto our central proof.

Claim 1 $\forall_{\mathbf{n}\in\eta(\mathbf{x})} : \phi(\mathbf{n},t-\Delta t) = \phi(\mathbf{n},t-2\Delta t)$ implies $\phi(\mathbf{x},t) = \phi(\mathbf{x},t-\Delta t)$.

Proof 2 We prove by contradiction. We assume that $\forall_{\mathbf{n}\in\eta(\mathbf{x})} : \phi(\mathbf{n},t-\Delta t) = \phi(\mathbf{n},t-2\Delta t)$. From this expression and the definition of Φ we get $\Phi(\mathbf{x},t-\Delta t) = \Phi(\mathbf{x},t-2\Delta t)$. From Lemma 1 we get $F(\mathbf{x},t) = F(\mathbf{x},t-\Delta t)$. From the definition of $\nabla\phi$ we get $\nabla\phi(\mathbf{x},t-\Delta t) = \nabla\phi(\mathbf{x},t-2\Delta t)$.

We assume for the sake of contradiction that $\phi(\mathbf{x},t) \neq \phi(\mathbf{x},t-\Delta t)$. Substituting this inequality into Equation 1 we get $\phi(\mathbf{x},t) - \phi(\mathbf{x},t-\Delta t) = \Delta t F(\mathbf{x},t)|\nabla\phi(\mathbf{x},t-\Delta t)| \neq 0$. From the zero product rule we get $F(\mathbf{x},t) \neq 0$ and $|\nabla\phi(\mathbf{x},t-\Delta t)| \neq 0$.

From our initial assumption that $\forall_{\mathbf{n}\in\eta(\mathbf{x})} : \phi(\mathbf{n},t-\Delta t) = \phi(\mathbf{n},t-2\Delta t)$, and since $\mathbf{x} \in \eta(\mathbf{x})$, we get $\phi(\mathbf{x},t-\Delta t) = \phi(\mathbf{x},t-2\Delta t)$. Substituting the right hand side of this expression into Equation 1 we get $\phi(\mathbf{x},t-\Delta t) = \phi(\mathbf{x},t-\Delta t) + \Delta t F(\mathbf{x},t-\Delta t)|\nabla\phi(\mathbf{x},t-2\Delta t)|$, or equivalently $\Delta t F(\mathbf{x},t-\Delta t)|\nabla\phi(\mathbf{x},t-2\Delta t)| = 0$. From this expression and the zero product rule we get either $F(\mathbf{x},t-\Delta t) = 0$ or $|\nabla\phi(\mathbf{x},t-2\Delta t)| = 0$.

We assume for the moment that $F(\mathbf{x},t-\Delta t) = 0$. Since $F(\mathbf{x},t) = F(\mathbf{x},t-\Delta t)$ we get $F(\mathbf{x},t) = 0$ which leads to a contradiction. Now we assume for the moment that $|\nabla\phi(\mathbf{x},t-2\Delta t)| = 0$. From this expression, and since $\nabla\phi(\mathbf{x},t-\Delta t) = \nabla\phi(\mathbf{x},t-2\Delta t)$, we get $|\nabla\phi(\mathbf{x},t-\Delta t)| = 0$ which also leads to a contradiction. Therefore $\phi(\mathbf{x},t) = \phi(\mathbf{x},t-\Delta t)$. □