Published in final edited form as:

Parallel Comput. 2013 April 1; 39(4-5): 189–211. doi:10.1016/j.parco.2013.03.001.

Efficient Irregular Wavefront Propagation Algorithms on Hybrid CPU-GPU Machines

George Teodoro*, Tony Pan, Tahsin Kurc, Jun Kong, Lee Cooper, and Joel Saltz

Center for Comprehensive Informatics and Biomedical Informatics Department, Emory University, Atlanta, GA 30322

Abstract

We address the problem of efficient execution of a computation pattern, referred to here as the irregular wavefront propagation pattern (IWPP), on hybrid systems with multiple CPUs and GPUs. The IWPP is common in several image processing operations. In the IWPP, data elements in the wavefront propagate waves to their neighboring elements on a grid if a propagation condition is satisfied. Elements receiving the propagated waves become part of the wavefront. This pattern results in irregular data accesses and computations. We develop and evaluate strategies for efficient computation and propagation of wavefronts using a multi-level queue structure. This queue structure improves the utilization of fast memories in a GPU and reduces synchronization overheads. We also develop a tile-based parallelization strategy to support execution on multiple CPUs and GPUs. We evaluate our approaches on a state-of-the-art GPU accelerated machine (equipped with 3 GPUs and 2 multicore CPUs) using the IWPP implementations of two widely used image processing operations: morphological reconstruction and euclidean distance transform. Our results show significant performance improvements on GPUs. The use of multiple CPUs and GPUs cooperatively attains speedups of 50× and 85× with respect to single core CPU executions for morphological reconstruction and euclidean distance transform, respectively.

Keywords

Irregular Wavefront Propagation Pattern; GPGPU; Cooperative CPU-GPU Execution; Heterogeneous Environments; Morphological Reconstruction; Euclidean Distance Transform

1. Introduction

We investigate the efficient parallelization on hybrid CPU-GPU systems of a computation structure that we refer to as the *irregular wavefront propagation pattern* (IWPP) (see Algorithm 1). We target CPU-GPU equipped machines, because they are emerging as viable high performance computing platforms for scientific computation [59]. Contemporary GPUs provide extremely fast memories and massive multi-processing capabilities, exceeding those of multi-core CPUs. The application and performance benefits of GPUs for general purpose processing have been demonstrated for a wide range of applications [15, 45, 55, 48, 6, 39].

Our work is motivated by the requirements of analysis of whole slide tissue images in biomedical research. We are interested in enabling more effective use of large datasets of high resolution tissue slide images in research and patient care. A typical image from state-

*Corresponding author george.teodoro@emory.edu (George Teodoro), tony.pan@emory.edu (Tony Pan), tkurc@emory.edu (Tahsin Kurc), jun.kong@emory.edu (Jun Kong), lee.cooper@emory.edu (Lee Cooper), jhsaltz@emory.edu (Joel Saltz).

of-the-art scanners is about $50K \times 50K$ to $100K \times 100K$ pixels in resolution; with rapid improvements in sensor technologies and scanner instruments, it is becoming feasible for research projects and healthcare organizations to gather large volumes of microscopy images. Microscopy images are, however, underutilized in research and clinical settings, because processing a high resolution image on a single CPU system can take hours. A whole slide tissue image is typically analyzed through a cascade of image normalization, object segmentation, object feature computation, and image classification stages. In our current implementation, segmentation is one of the most expensive stages and composed of a pipeline of substages. The most expensive substages are built on several low-level operations, notably morphological reconstruction [62] and distance transform [60]. Efficient implementation of these operations is necessary to reduce the cost of image analysis.

The processing structures of the morphological reconstruction and distance transform operations bear similarities and include the IWPP on a grid. The IWPP is characterized by one or more source grid points from which waves originate and the irregular shape and expansion of the wave fronts. The composition of the waves is dynamic, data dependent, and computed during execution as the waves are expanded. Elements in the front of the waves work as the sources of wave propagations to neighbor elements. A propagation occurs only when a given *propagation condition*, determined based on the value of a wavefront element and the values of its neighbors, is satisfied. In practice, each element in the propagation front represents an independent wave propagation; interaction between waves may even change the direction of the propagation. In the IWPP only those elements in the wavefront are the ones effectively contributing for the output results. Because of this property, an efficient implementation of irregular wavefront propagation can be accomplished using an auxiliary container structure, e.g., a queue, set, or stack, to keep track of active elements forming the wavefront. The basic components of the IWPP are shown in Algorithm 1.

In this algorithm, a set of elements in a multi-dimensional grid space (D) are selected to form the initial wavefront (S). These active elements then act as wave propagation sources in the wavefront propagation phase. During propagation phase, a single element (e_i) is extracted from the wavefront and its neighbors ($Q \leftarrow N_G(e_i)$) are identified. The neighborhood of an element e_i is defined by a discrete grid G , also referred to as the structuring element. The element e_i tries to propagate the wavefront to each neighbor $e_j \in Q$. If the propagation condition (*PropagationCondition*), based on the values of e_i and e_j is satisfied, the value of the element e_j ($D(e_j)$) is updated, and e_j is inserted in the container S . The wave expansion operations are expected to be commutative and atomic. That is, the order in which the wavefront computations are carried out should not impact the algorithm results. The wavefront propagation process continues until stability is reached; i.e., until the wavefront container is empty.

The IWPP is not unique to morphological reconstruction and distance transform. Core computations in several other image processing methods contain a similar structure: Watershed [63], Euclidean skeletons [32], and skeletons by influence zones [26]. Additionally, Delaunay triangulations [38], Gabriel graphs [16] and relative neighborhood graphs [57] can be derived from these methods. Another example of the IWPP is the shortest-path computations on a grid which contains obstacles between source points and destination points (e.g., shortest path calculations in integrated circuit designs). A high performance implementation of the IWPP can benefit these methods and applications.

The traditional wavefront computation is common in many scientific applications [64, 3, 2]. It is also a well-known parallelization pattern in high performance computing. In the classic wavefront pattern, data elements are laid out in a multi-dimensional grid. The computation of an element on the wavefront depends on the computation of a set of neighbor points. The

classic wavefront pattern has a regular data access and computation structure in that the wavefront starts from a corner point of the grid and sweeps the grid diagonally from one region to another. The morphological reconstruction and distance transform operations could potentially be implemented in the form of iterative traditional wavefront computations. The IWPP offers a more efficient execution structure, since it avoids touching and computing on data points that do not contribute to the output.

The IWPP implementation on a hybrid machine consisting of multiple CPUs and GPUs is a challenging problem. The difficulties with IWPP parallelization are accentuated by the irregularity of the computation that is spread across the input domain and evolves during the computation as active elements change. In this paper, we propose, implement, and evaluate parallelization strategies for efficient execution of IWPP computations on large grids (e.g., high resolution images) on a machine with multiple CPUs and GPUs. The proposed strategies can take full advantage this computation pattern characteristics to achieve efficient execution on multicore CPU and GPU systems. The contributions of our work can be summarized as follows:

- We identify a computation pattern commonly found in several analysis operations: the irregular wavefront propagation pattern.
- We develop an efficient GPU implementation of the IWPP algorithm using a multi-level queue structure. The multi-level queue structure improves the utilization of fast memories in a GPU and reduces synchronization overheads among GPU threads. To the best of our knowledge this is the first work to implement this pattern for GPU accelerated environments.
- We develop efficient implementations of the morphological reconstruction and distance transform operations on a GPU using the GPU-enabled IWPP algorithm. These are the first GPU-enabled IWPP-based implementations of morphological reconstruction and distance transform. The morphological reconstruction implementation achieves much better performance than a previous implementation based on raster/anti-raster scans of the image. The output results for morphological reconstruction and distance transform are exact regarding their sequential counterparts.
- We extend the GPU implementation of the IWPP to support processing of images that do not fit in GPU memory through coordinated use of multiple CPU cores and multiple GPUs on the machine.

We perform a performance evaluation of the IWPP implementations using the morphological reconstruction and distance transform operations on a state-of-the-art hybrid machine with 12 CPU cores and 3 NVIDIA Tesla GPUs. Significant performance improvements were observed in our experiments. Speedups of 50× and 85× were achieved, as compared to the single CPU core execution, respectively, for morphological reconstruction and euclidean distance transform when all the CPU cores and GPUs were used in a coordinated manner. We were able to compute morphological reconstruction on a 96K×96K-pixel image in 21 seconds and distance transform on a 64K×64K-pixel image in 4.1 seconds. These performances make it feasible to analyze very high resolution images rapidly and conduct large scale studies.

The manuscript is organized as follows. Section 2 presents the use of the IWPP on image analysis, including preliminary definitions and the implementation of use case algorithms: morphological reconstruction and euclidean distance transform. The IWPP parallelization strategy and its support in GPUs are discussed in Section 3. The extensions for multiple GPUs and cooperative CPU-GPU execution are described in Section 4. The experimental

evaluation is presented in Section 5. Finally, Sections 6 and 7, respectively, present the related work and conclude the paper.

2. Irregular Wavefront Propagation Pattern in Image Analysis

We describe two morphological algorithms, *Morphological Reconstruction* and *Euclidean Distance Transform*, from the image analysis domain to illustrate the IWPP. Morphological operators are basic operations used by a broad set of image processing algorithms. These operators are applied to individual pixels and are computed based on the current value of a pixel and the values of pixels in its neighborhood. A pixel p is a neighbor of pixel q if $(p, q) \in G$. G is usually a 4-connected or 8-connected square grid. $N_G(p)$ refers to the set of pixels that are neighbors of $p \in \mathbb{Z}^n$ according to G ($N_G(p) = \{q \in \mathbb{Z}^n \mid (p, q) \in G\}$). Input output images are defined in a rectangular domain $D_I \in \mathbb{Z}^n \rightarrow \mathbb{Z}$. The value $I(p)$ of each image pixel p assumes 0 or 1 for binary images. For gray scale images, the value of a pixel comes from a set $\{0, \dots, L-1\}$ of gray levels from discrete or continuous domain.

2.1. Morphological Reconstruction Algorithms

Morphological reconstruction is one of the elementary operations in image segmentation [62]. Figure 1 illustrates the process of gray scale morphological reconstruction in 1-dimension. The marker intensity profile is propagated spatially but is bounded by the mask image's intensity profile.

The morphological reconstruction $\rho_I(J)$ of mask I from marker image J can be done by performing elementary dilations (i.e., dilations of size 1) in J by G , the structuring element. An elementary dilation from a pixel p corresponds to propagation from p to its immediate neighbors in G . The basic algorithm carries out elementary dilations successively over the entire image J , updates each pixel in J with the pixelwise minimum of the dilation's result and the corresponding pixel in I (i.e., $J(p) \leftarrow (\max\{J(q), q \in N_G(p) \cup \{p\}\}) \wedge I(p)$, where $\wedge$ is the pixelwise minimum operator), and stops when *stability* is reached, i.e., when no more pixel values are modified. Vincent [62] has developed morphological reconstruction algorithms for gray scale images based on this core technique. We present brief descriptions of these algorithms below and refer the reader to the original paper for more details.

Sequential Reconstruction (SR)—Pixel value propagation in the marker image is computed by alternating raster and anti-raster scans. A raster scan starts from the pixel at $(0, 0)$ and proceeds to the pixel at $(N-1, M-1)$ in a row-wise manner, while an anti-raster starts from the pixel at $(N-1, M-1)$ and moves to the pixel at $(0, 0)$ in a row-wise manner. Here, N and M are the resolution of the image in x and y dimensions, respectively. In each scan, values from pixels in the upper left or the lower right half neighborhood are propagated to the current pixel in raster or anti-raster fashion, respectively. The SR method iterates until stability is reached, i.e., no more changes in pixels are computed.

Fast Hybrid Reconstruction (FH)—This algorithm makes one pass using the raster and anti-raster orders as in SR. It then continues the computation using a first-in first-out (FIFO) queue. The queue is initialized with pixels that satisfies the propagation condition, and the computation proceeds by removing a pixel from the queue, scanning the pixel's neighborhood, and queuing the neighbor pixels whose values have been changed as consequence of the wavefront computation. The overall process continues until the queue is empty. A pseudo-code implementation of FH is presented in Algorithm 2, N_G^+ and N_G^- denote the set of neighbors in $N_G(p)$ that are reached before and after touching pixel p during a raster scan.

2.2. Euclidean Distance Transform

The distances transform (DT) operation computes a distance map M from a binary input image I , where for each pixel $p \in I$ the pixel's value in M , $M(p)$, is the smallest distance from p to a background pixel. DT is a fundamental operator in a number of image computations, including shape analysis [63, 43], morphological operations [8], and the computation of Voronoi diagrams and Delaunay triangulation [60]. The definition of DT is simple, but historically it has been hard to achieve good precision and efficiency for this operation. For a comprehensive discussion of algorithms for calculation of distance transform, we refer the reader to the survey by Fabbri et al. [13].

Our implementation uses an algorithm proposed by Danielsson [11] to compute Euclidean DT that propagates information of the nearest background pixel using neighborhood operators. Intrinsically, the algorithm builds Voronoi diagrams where regions are formed by pixels with the same nearest background pixel. Danielsson's algorithm is not an exact algorithm. Nevertheless, the approximation errors introduced by the algorithm are small and bound by a mathematical framework. They have been found to be sufficiently small in practice for several applications [9, 14, 56]. Moreover, some of the exact euclidean distance transform algorithms [8, 46, 34] are implemented as a post-processing phase to Danielsson's algorithm, in which approximation errors are resolved.

Our implementation, shown in Algorithm 3, computes the single neighborhood approximation of DT using the IWPP. In the *initialization phase* a pass on the input is performed to identify and queue pixels forming the initial wavefront. In Lines 2 and 3, the Voronoi diagram (VR) that holds the current nearest background pixel for each image pixel is initialized. Each background pixel will itself be its nearest pixel, while the foreground pixels will initially point to a pixel virtually at infinite distance (*inf*). The *inf* pixel is such that for each pixel p , p is closer to any other pixel in the image domain than it is to the *inf* pixel. At the end of the initialization phase, background pixels with a foreground neighbor (contour pixels) are added to a queue for propagation (Lines 4 and 5).

The *wavefront propagation phase* of the EDT is carried out in Lines 7 to 12. During each iteration, a pixel p is removed from the queue. The pixel's nearest background pixel is propagated to the neighbors through a neighborhood operator. For each neighbor pixel q , if the distance of q to the current assigned nearest background pixel ($\text{DIST}(q, \text{VR}(q))$) is greater than the distance of q to the nearest background pixel of p ($\text{DIST}(q, \text{VR}(p))$), the propagation occurs. In that case, the nearest background pixel of q is updated with that of p (Line 11), and q is added to the queue for propagation. After the propagation phase is completed, the distance map is computed from the Voronoi diagram.

3. Implementation of Irregular Wavefront Propagation Pattern on a GPU

We first describe the strategy for parallel execution of the IWPP on a GPU. We then present a parallel queue data structure that is fundamental to efficiently tracking data elements in the wavefront. In the last two sections, we describe the GPU-enabled implementations of the two operations presented in Section 2.

3.1. Parallelization Strategy

The overview of our parallelization strategy, including the mapping of work to GPU thread blocks, is presented in Figure 2. The first step in the IWPP execution identifies the active elements forming the initial wavefront. It is necessary to evaluate the propagation condition for each element of the input data domain in order to identify these elements. The elements for which the propagation condition evaluates *true* are added to the set S of initial active elements, also referred to here as seeds. After the initialization stage, the set of seeds S is

divided into Z partitions ($P_1, \dots, P_Z$). Each of the seed partitions is mapped to a GPU thread block for parallel computation, therefore, Z thread blocks are used in the GPU execution. Each thread block i creates an independent instance of the hierarchical queue (Queue i in Figure 2), described in Section 3.2, which is initialized with elements from the seed partition P_i assigned to the thread block i .

This partitioning and mapping of seeds to different thread blocks eliminates communication and synchronization among the queue instances assigned to different thread blocks. We should note that propagations performed by the IWPP must be commutative for this to be possible. That is, the order in which propagations are computed should not change the final results. By using this partitioning approach, we avoid the maintenance of a queue shared among all thread blocks – a shared queue would adversely impact scalability because of inter-block synchronizations [65] needed to ensure queue consistency across GPU multi-processors. A drawback of our solution is the fact that work never migrates between stream multi-processors. In other words, propagations originated from partition P_i are computed by thread block i only. This may lead to load imbalance among stream multi-processors. However, since our framework allows for the instantiation of a large number of thread blocks, we can create thread blocks with smaller number of seeds and, consequently, reduce the effects of load imbalance.

Algorithm 4 presents the GPU-based propagation computations carried out in each thread block, starting with the seed partition assigned to it. Each block of threads enters into a loop in which data elements are dequeued in parallel and processed, and new data elements may be added to that block local (`queue[blockId]`) queue as needed. This process continues until the queue for that block is empty. At that point, the particular block with no more propagations to compute leaves the main loop and finishes its execution. The GPU will then schedule another block of threads for computation, if one exists.

The work done in each iteration of the main loop (Lines 1–14) consists of processing each element queued in the last iteration, or in the initialization phase if it is the first iteration. The data elements are dequeued in parallel and each GPU thread is responsible for carrying out propagations associated to a given element p it dequeued. Because the number of neighbors visited by each GPU thread to check the propagation condition is the same, defined by the structuring element N_G , the size of the neighborhood to be visited by each thread is the same.

The data element p is compared to every data element q in its neighborhood. If the *PropagationCondition* is true, the value of p should propagate to its neighbor q . Since several threads are executed concurrently and some of them may be computing elements with overlapping neighborhoods, the propagation test and the update of the neighbor element must be atomic.

The implementation of the atomic propagation uses the atomic compare and swap (*atomicCAS*) operation supported by CUDA. The value of the element q , used to compute the propagation condition, is stored in *curValueQ*. The atomicCAS operation uses *curValueQ* to ensure that the propagation only occurs if the value of q ($I(q)$) did not change in the interval the propagation condition was computed and the data update happens (Lines 5 to 7). The atomicCAS operation returns the value of the element q before the execution of compare and swap (*oldval*). This value is used to determine whether the value of q has really been changed (Line 8 in Algorithm 4). In other words, if the *oldval* is equals to *curValueQ*, we know that the update occurred. If atomicCAS performed by the current thread has changed the value of q , q is added to the queue for processing in the next iteration of the loop. Even with this control, it is possible that between the test at Line 8 and the addition to

the queue, the value of q may have been modified again. In this case, q is enqueued multiple times. This increases the computation cost, but the correctness of the algorithm is not affected because the update operations replace the value of q via commutative and atomic propagations. In the case the *atomicCAS* fails to update q , the propagation condition must be recalculated with the new value of q to make sure it still is true.

After computing the propagations from the last iteration, data elements that are added to the queue in the current iteration are made available for the next iteration. Elements used as input in one iteration and those queued as a result of propagations are stored in different regions of the queue structure: (i) the *input* region stores elements to be computed in the current iteration; (ii) the *output* region holds elements queued for computation during the current iteration, which will be computed in the next iteration. The choice of employing separate regions, instead of using a single region and make the queued elements immediately available for computation, was made in our design to implement a queue with very efficient read performance (dequeue performance) and with low synchronization costs to provide consistency among threads when multiple threads concurrently read from and write to the queue. Using this design, the algorithm processes the elements in rounds, where after processing all elements from the input region, the output or queued elements are made available for the next round by swapping the input and output regions (`queue_swap_in_out()`; Lines 14). Our dequeue operation is very simple, but efficient, as it guarantees that in each iteration the number of elements forming the wavefront are distributed evenly among the threads in each thread block, and the access to the queue is a memory coalesced operation. The details of the dequeue operation are presented at the end of Section 3.2. In summary, the elements stored in the queue are put into the *output* region and placed in consecutive positions in an array. Thus, after swapping the *input* and *output* regions, the new *input* region will have the elements queued in the last iteration and the new *output* region will be empty. Since the elements to be computed in the current iteration are stored into an array, they can be divided evenly among the threads in that block as follows: the elements assigned to a thread warp are determined based on the warp offset on the thread block with increments equal to the number of threads in the block. This is similar to a tiled parallel loop with the tile size equal to the warp size. After each iteration in the main loop of Algorithm 4, the elements ready for computation in that iteration are re-balanced among threads without incurring expensive overheads.

The execution of each neighborhood operator is performed considering the value of the current data element being processed (p) and that of each data element in the neighborhood ($q \in N_G(p)$). Since multiple GPU threads may be concurrently performing computation on the same or neighbor elements, it is necessary to guarantee atomicity in updates on neighbor data elements (q). This can only be achieved at the cost of extra synchronization overheads. Fortunately, the use of *atomicCAS* operations is sufficient to solve this race condition. The efficiency of atomic operations in GPUs has been significantly improved in the last generation of NVIDIA GPUs (Fermi) [35] because of the use of cache memory. Atomic operations obviously still are more efficient in cases where threads do not concurrently try to update the same memory address. When multiple threads attempt to update the same memory location, they are serialized in order to ensure correct execution. As a result, the number of operations successfully performed per cycle might be drastically reduced [35]. To lessen the impact of potential serialization, our GPU parallelization employs data element level parallelism which allows for data elements queued for computation to be independently processed. In addition, when maximum and minimum operations are used in the propagation condition test, *atomicCAS* may be replaced with an *atomicMax* or *atomicMin* operation to improve performance.

Since elements in the queue are divided among threads for parallel computation, the order in which the elements will be processed cannot be pre-determined. Thus it is necessary that wavefront propagation operations in an application be *commutative* and *atomic* for correct execution of the parallel computation. In practice, the order is not important for most analysis methods based on the IWPP, including the example operations presented in this paper.

3.2. Parallel Queue Implementation and Management

A parallel queue is used to keep track of elements in the wavefront. The implementation of an efficient parallel queue for GPUs is a challenging problem [18, 21, 30]. A straight forward implementation of a queue, as presented by Hong et al. [18], could be done by employing an array to store items in sequence and using atomic additions to calculate the position where the item should be inserted, as presented in the code below.

```
AddQueue(int* q_idx, type_t* q, type_t item) {
    int old_idx = AtomicAdd(q_idx, 1);
    q[old_idx] = item; }
```

Hong et al. stated that this solution worked well for their use case. The use of atomic operations, however, is very inefficient when the queue is heavily employed as in our case. Moreover, a single queue that is shared among all thread blocks introduces additional overheads to guarantee data consistency across the entire device.

We have developed a customized queue structure that exploits the characteristics of the IWPP to achieve the necessary efficiency and scalability. As presented in the previous section, propagations computed by the IWPP can be carried out independently on disjoint partitions of the data elements – thanks to the commutative property of the propagations. This is an essential IWPP property that enables our queue structure to be built using a set of queues, one per thread block, which operate independently and do not communicate or synchronize with each other. Since there is no synchronization between the thread block queues, the multi-queue implementation scales with the number of thread blocks. Nevertheless, optimizing the performance of queue operations in each queue instance remains a problem. The queue operations should minimize the synchronization costs among threads in the corresponding thread block to achieve maximum intra-block scalability and exploit the GPU memory hierarchy to take advantage the fast memories and speed up the queue operations.

Another important aspect of our queue design is that elements added to the queue and those being currently consumed are stored into two different memory regions, called *input* and *output*, and these regions are implemented as an array of elements. This design results in very efficient read performance (dequeue performance) and low synchronization costs. However, once the input region is empty, the application must swap the input and output regions, such that elements stored in the queue or in the output region are made available for computation. This swap operation is executed by the `queue.swap_in_out` operation, which simply change the queue pointers referencing the *input* and *output* regions. The rest of this section describes the *enqueue* and *dequeue* operations.

3.2.1. Enqueue Operation—To allow for efficient addition of elements to the queue, we have implemented a multiple level queue structure (as depicted in Figure 3), which consists of: (i) Per-Thread queues (TQ) which are very small queues private to each thread, residing in the shared memory; (ii) Block Level Queue (BQ) which is also in the shared memory, but

is larger than TQ. Write operations to the block level queue are performed in thread warp-basis; and (iii) Global Memory Block Level Queue (GBQ) which is the largest queue and uses the global memory of the GPU to accumulate data stored in BQ when the size of BQ exceeds the shared memory size.

In our queue implementation, each thread maintains an independent queue (TQ) that does not require any synchronization to be performed. In this way threads can efficiently store data elements from a given neighborhood in the queue for computation. Whenever a TQ is full, it is necessary to perform a warp level communication to aggregate items stored in individual TQs and write them to BQ. In this phase, a parallel thread warp prefix-sum is performed to compute the total number of items queued in individual TQs. A single shared memory atomic operation is performed at the Block Level Queue to identify the position where the warp of threads should write their data. This position is returned, and the warp of threads write their local queues to BQ.

Whenever BQ is full, the threads in the corresponding thread block are synchronized. The current size of the queue is used in a single operation to calculate the offset in GBQ from which the BQ should be stored. After this step, all the threads in the thread block work collaboratively to copy in parallel the contents of QB to GBQ. This data transfer operation is able to achieve high throughput, because the memory accesses are coalesced. It should be noted that, in all levels of the parallel queue, an array is used to hold the queue content. While the contents of TQ and BQ are copied to the lower level queues when full, GBQ does not have a lower level queue to which its content can be copied. In the current implementation, the size of GBQ is initialized with a fixed tunable memory space. If the limit of GBQ is reached during an iteration of the algorithm, excess data elements are dropped and not stored. The GPU method returns a boolean value to CPU to indicate that some data elements have been dropped during the execution of the method kernel. In that case, the wavefront propagation algorithm has to be re-executed, but using the output of the previous execution as input, because this output already holds a partial solution of the problem. Recomputing using the output of the previous step as input will result in the same solution as if the execution was carried out in a single step.

3.2.2. Dequeue Operation—While adding new items to the queue requires a number of operations to guarantee consistency, the read operations can be performed without introducing synchronization. This is possible because we can partition input data elements statically among threads, which keep consuming the data elements until all items are computed. A high level description of the *dequeue* function is presented bellow. In this function, the value of the item is initialized to a default code corresponding to an empty queue. Then, the queue index corresponding to the next item to be returned for that particular thread is calculated, based on the size of the thread block (*block_size*), the number of iterations already computed by this thread warp (*warp_iter*), and the offset of that thread in the block (*tid_in_block*). As might be noticed, the access to the data are coalesced and executed in a warp-basis, allowing for efficient reading operations.

```
type_t DeQueue(int q_size, type_t* q, int warp_iter) {
    type_t item = QUEUE_EMPTY;
    queue_idx = tid_in_block + warp_iter * block_size;
    if(queue_idx < q_size){
        type_t item = in_queue[queue_idx];
    }
    return item; }
```

3.3. GPU-enabled Fast Hybrid Morphological Reconstruction—In this section, we present how the GPU-based version of the Fast Hybrid Reconstruction algorithm, referred to in this paper as FH_GPU, is implemented (see Algorithm 5).

The first stage of FH, as is described in Section 2.1, consists of the raster and anti-raster scans of the image. This stage has a regular computation pattern that operates on all pixels of the image. An efficient GPU implementation of the SR algorithm, including the raster and anti-raster scan stage, has been done by Pavel Karas [21]. This implementation is referred to as SR_GPU in this paper. We employ the SR_GPU implementation to perform the first phase of FH on a GPU. Note that the raster scan is decomposed into scans along each dimension of the image. The neighborhoods are defined as $N_G^{+,0}$ and $N_G^{0,+}$ for the sets of neighbors for the row-wise and column-wise scans, respectively. Similarly, for the anti-raster scan, $N_G^{-,0}$ and $N_G^{0,-}$ are defined for the respective axis-aligned component scans.

The transition from the raster scan phase to the wavefront propagation phase is slightly different from the sequential FH. The SR_GPU does not guarantee consistency when a pixel value is updated in parallel. Therefore, after the raster scan phase, pixels inserted into the set of elements forming the initial wavefront (S) for computation in the next phase are not only those from the anti-raster neighborhood (N_G^-) as in the sequential algorithm, but also pixels from the entire neighborhood (N_G) satisfying the propagation condition (See Algorithm 5, Lines 16–18).

When two queued pixels (e.g., p' and p'') are computed in parallel, it is possible that they have common neighbors ($N_G(p') \cap N_G(p'') \neq \emptyset$). This can potentially create a race condition when updating the same neighbor q . q is updated via a maximum operation, which can be efficiently implemented to avoid race conditions by employing atomicCAS based instructions (*atomicMax* for GPUs). The old value of pixel q (*oldval*) is used in our algorithm to test if q was updated due to the computation of the current pixel p' . If *oldval* is not smaller than $\min\{J(p'), I(q)\}$, it means that the processing of another pixel p'' has updated the value of q to a value greater than or equal to $\min\{J(p'), I(q)\}$. In this case, q has already been queued due to the p' computation and does not need to be queued again. Otherwise, q is queued for computation. The overall process continues until the queue is empty. Note that the update operation is a maximum, hence an *atomicMax* is used instead of the *atomicCAS* operation in the generic wavefront propagation skeleton. It leads to better performance as the repeat-until loop in Algorithm 4 (Lines 4–13) is avoided. We also refer the reader to a previous technical report [52] with additional details on our GPU-based implementation of the Morphological Reconstruction.

3.4. GPU-enabled Euclidean Distance Transform

The GPU implementation of distance transform is shown in Algorithm 6. The implementation is very similar to the sequential version. The initialization phase assigns initial values to the Voronoi diagram and adds contour pixels (i.e., the background pixels with foreground neighbors) to the set S of propagation seeds (Lines 2 to 5).

In the wavefront propagation phase, the set of propagation seeds is divided into Z independent partitions and are assigned to different GPU threads blocks. During the wavefront propagation phase, for each pixel p queued for computation, its neighbor pixels q are checked to verify if using $VR(p)$ as path to a background pixel would result in a shorter distance as compared to the current path stored in $VR(q)$. If it does, $VR(q)$ is updated and q is enqueued for computation. Once again, concurrent computation of pixels in the queue may create a race condition if multiple threads try to update the same $VR(q)$. To prevent this

condition, a compare and swap operation is employed when updating VR . VR is only updated if the value used to calculate distance ($curVRQ$) is still the value stored in $VR(q)$. Otherwise, the algorithm reads the new value stored in $VR(q)$ and calculates the distance again. The final step, as in the sequential case, computes the distance map from VR . It is performed in parallel for all points in the input image.

A given VR diagram may have multiple valid solutions. If two pixels, p' and p'' , are equidistant and the closest background pixels to a foreground pixel q , either of the pixels can be used as the closest background pixel of q ($VR(q) = p'$ or $VR(q) = p''$). In other words, a foreground pixel could initially be assigned one of multiple background pixels, as long as the distance from the foreground pixel to those background pixels is the same. In our algorithm, the order in which the wave propagations are computed defines whether p' or p'' are used in such a case. However, since both pixels are equidistant to q , using either of them would result in the same distance map. Thus, the results achieved by the parallel distance transform are the same as those of the sequential version of the algorithm.

4. Parallel Execution on Multiple GPUs and CPUs

Our parallelization strategy, shown in Figure 4, for multiple GPUs and CPUs divides the input data domain into non-overlapping partitions (tiles). Local wavefront propagations are computed independently for each partition. After the local propagations are computed, border information is exchanged to allow for inter-partition propagations. This process continues until there are no more propagations within a partition and between partitions.

Inter-processor communication is necessary because the irregular wavefront propagation pattern involves wavefront propagations that are data dependent. The shape of the wavefront and the amount of wave propagation cannot be estimated prior to execution. As a result, a partitioning of the input data domain across processing units can incur inter-partition propagations and inter-processor communication, which cannot be eliminated by padding each domain partition. As noted earlier, propagation computations in our framework are commutative. This guarantees that the final results of the tile based computation in parallel are the same as the results from a single processor execution.

Figure 5 shows an example in which an input data domain is partitioned between two processors. The morphological reconstruction algorithm is used in this example. The input data domain, described in a 1-dimensional space in the example, is equally divided between processors 1 and 2 without any replication at the beginning of the execution (Figure 5(a)). Each processor independently performs the wavefront propagation in their respective partitions by identifying seeds and computing the usual propagation as required by the algorithm. When the stability is reached (i.e., the queue is empty) in the partitions, there may be propagations crossing partition boundaries that have not been computed. Figure 5(b) presents the result of propagations within a partition. To compute inter-partition propagation effects correctly, processors must exchange border information to identify inter-partition propagations. After border information is exchanged, propagations originating from partition borders are initialized and another round of local propagations is executed. This process continues until no more propagations are possible within and between partitions. The final result of the morphological reconstruction for the 1-dimensional example is presented in Figure 5(c). The area which is recomputed because of the inter-partition propagations is represented in dark green.

The parallel implementation consists of a pipeline of two stages (Figure 6): Tile Propagation (TP) and Border Propagation (BP). The TP stage performs local propagations in tiles. An instance of this stage is created for each tile. The BP stage, on the other hand, computes propagations from data elements at tile borders. In essence, BP transmits propagations from

one tile to another. When there is a need for propagation between tiles, BP will re-instantiate the pipeline. In that case, a TP instance is created for each tile receiving propagation and scheduled for execution. It computes propagations within the tile that are initiated from the borders. A new instance of BP is also created with a dependency on the new TP instances. That is, the BP instance is not scheduled for execution until all of the new TP instances finish their propagation computations. This cycle is repeated until there are no more intra- and inter-tile propagations.

The instances of the two stages are dispatched for execution on available CPUs and GPUs, as long as a stage has an implementation for those devices. We employ the concept of function variant, which is a group of functions with same name, arguments, and result types [28, 33]. In our implementation a function variant for a task is the CPU and GPU implementations of the operation. Binding to a function variant enables the runtime system to choose the appropriate function or functions during execution, allowing multiple computing devices to be used concurrently and in a coordinated manner. We should note that a function variant provides opportunities for runtime optimizations, but an application is not required to provide CPU and GPU implementations for all of the stages.

Figure 7 presents an overview of our system support [53, 51] to schedule pipeline applications on machines equipped with multiple CPUs and GPUs. Tasks (TP and BP instances) dispatched for execution by the wavefront propagation pipeline are either queued as ready to execute or inserted into a queue of pending tasks, respectively, depending on whether their dependencies are resolved or not. Tasks ready for execution are consumed by threads responsible for managing CPU cores and GPUs. A computing thread is assigned to each CPU-core or GPU available in the default mode. Assignment of tasks to computing devices is done in a demand-driven manner. When a CPU core or GPU remains idle, one of the tasks is assigned to the idle device. The default scheduling policy for choosing the task to be executed into a device requesting work is FCFS (first come, first served). In the case of our application, since the BP instance will be dependent on all TP instances, the BP is only dispatched for execution after all the TPs have finished.

When a TP instance is scheduled for GPU execution, the tile to be processed is copied to the GPU memory. The results are transferred back to the CPU memory at the end of processing. The TP instances are dynamically assigned to processing devices using a demand-driven approach to alleviate load imbalance that may arise due to different computation requirements among tiles. Moreover, with a partitioning of input data into tiles, GPUs can be used when the input data does not fit into GPU memory but individual tiles do.

Our experimental evaluation shows that the maximum GPU performance for the IWPP is achieved when large input domain partitions are used. This is also observed for other types of GPU computations, because larger inputs usually lead to higher computation times that better offset the overheads of using GPUs. However, when CPUs and GPUs are used together, large tiles tend to reduce the performance gain because of the greater gap in execution times on a CPU core vs on a GPU. This increases load imbalance between the CPU and the GPU. To reduce the impact of load imbalance, our implementation allows for parallel CPU processing variants to be used. In this approach, instead of scheduling a single partition per CPU core, a single partition is scheduled to a group of CPU cores, if a parallel variant of the CPU processing operation is available. Consequently, a partition assigned to a CPU will have an execution time closer to that of a partition assigned to a GPU.

As presented at the bottom of Figure 6, a partition assigned for parallel CPU computation is broken into smaller micro-partitions that are processed by multiple CPU cores in parallel. Border communication between partitions assigned to different CPU cores is carried out for

inter-partition propagations. The runtime system (Figure 7) is responsible for setting the number of CPU cores available for a given partition. The multi-core version of the propagation stages is implemented on top of OpenMP [10].

The partitioning of an input data domain into tiles for multi-processor execution has been employed before in a number of applications and systems [37, 55, 23]. In this work, the tiling is employed on top of a runtime system which allows for dynamic mapping of tile partitions to a heterogeneous collection of processing units to enable the use of a GPU for processing large input data domains that, otherwise, would not fit in the GPU memory. A key aspect that allows domain partition in IWPP is that propagation computations are commutative.

5. Experimental Results

We evaluate the performance of the irregular wavefront propagation implementation using the morphological reconstruction and distance transform operations. These operations were applied on high-resolution RGB images from datasets collected by research studies in the In Silico Brain Tumor Research Center (ISBTRC) [43] at Emory University. The experiments were carried out on a machine (a single computation node of the Keeneland cluster [59]) with Intel® Xeon® X5660 2.8 GHz CPUs, 3 NVIDIA Tesla M2090 (Fermi) GPUs, and 24GB of DDR3 RAM (See Figure 8). A total of 12 CPU computing cores are available because hyper-threading mechanism is not enabled. Codes used in our evaluation were compiled with “gcc 4.2.1”, “-O3” optimization flag, and NVidia CUDA SDK 4.0. The experiments were repeated 3 times, and, unless stated, the standard deviation was not observed to be higher than 1%. The speedup values presented in the following sections are calculated based on the performance of single CPU-core versions of the operations.

5.1. Multi-core CPU Parallelization

Two different multi-core CPU parallelization strategies were implemented using OpenMP [10]: (i) *Tile-based parallelization* that partitions an input image into regions, iteratively solves propagations local to each partition, and performs inter-region propagations, as described in Section 4; (ii) *Non-Tiled parallelization* is an alternative parallelization in which points in the initial wavefront are distributed in round-robin among CPU threads at the beginning of the execution. Each thread then independently computes propagations associated with the points assigned to it. To avoid inter-thread interference in the propagation computations, it is necessary to use *atomic* compare-and-swap operations during data updates.

The performances of both parallelization strategies are presented in Figure 9. The non-tiled parallelization of the morphological reconstruction resulted in slowdown as compared to the sequential algorithm for most of the configurations; only a slight improvement was observed on 8 CPU cores. In order to understand this poor performance, the algorithm was profiled using *perf* profiling tool [22] to measure the effects of increasing the number of threads to the cache and the bus cycles. Additional bus cycles are used for data read and write operations as well as for data transfers to ensure cache coherence among threads. An increase in the number of threads from 1 to 2 in the non-tiled parallelization results in twice more cache misses and three times more bus cycles. Similar behavior is observed as the number of threads is further increased, preventing the algorithm from having performance gains as more CPU cores are used. For this parallelization, threads may be concurrently computing in the same region of the input domain, and updates from one thread will invalidate cache lines of all the other threads processing the same region. This makes the cost of maintaining cache coherence very high.

The performance of the tile-based parallelization is much better with a speedup of $7.5\times$ in the configuration with 12 threads. This is a result of avoiding inter-thread interference. Because threads are assigned to independent partitions, they do not modify the same regions and, consequently, do not influence each other. Nevertheless, the speedup performance of the algorithm is sub-linear. This is an inevitable consequence of the threads' having to share the processor cache space and the bus subsystem. Since the algorithm is very data intensive, larger cache and fast access to memory are primordial to achieve maximum performance, and contention on these resources limits gains in CPU multicore systems.

5.2. GPU Parallelization

The experiments in this section examine the impact on performance of the queue design, the choice of tile size, and input data characteristics for the GPU-enabled implementations of irregular wavefront propagation computations.

5.2.1. Cost and Performance of Elementary Queue Operations—This section assesses the performance of our queue structure in the execution of elementary enqueue and dequeue operations. For the sake of this analysis, we implemented a *kernel* application that iteratively dequeues all elements in the queue and enqueues them again. In our evaluation, this application iterates $100\times$ and enqueues and dequeues all the elements in each iteration. The queue is initialized in each run with a different number of elements that are grouped into same size chunks and distributed among the queue instances assigned to different thread blocks. This set of experiments evaluates the performance of three different parallel queue approaches described in Section 3.2: (i) GBQ. This is the queue implementation based on the atomic add operations only; the queue storage uses the global memory in the GPU; (ii) GBQ+BQ (BQ). BQ performs a prefix-sum among GPU threads to calculate the positions where items are stored in the queue, reducing the synchronization cost due to the atomic operations, and uses shared memory to build the BQ layer on top of the GBQ that uses the slower global memory; and (iii) BGQ+BQ+TQ (TQ). TQ employs local queues in addition to BQ. All queue versions are instantiated for each GPU thread block, and there is no synchronization between the queue instances used by different thread blocks.

The results are presented in Figure 10. Figure 10(a) presents the execution times of the queue versions for different number of operations (enqueue+dequeue) on the queue. The BQ queue was able to strongly improve the performance of the queue structure. This is the result of the smaller number of atomic operations because of the use of prefix sum calculation at the block level to aggregate the number of elements to be queued. In addition, the use of a thread local queue (TQ) improved the performance of the BQ by $1.27\times$, because it prevents the threads from accessing BQ until TQ is full. The average cost to enqueue and dequeue an element, using the TQ configuration, was only $0.64\mu s$.

5.2.2. Impact of Queue Design Choices—This set of experiments evaluates the impact of different queue versions (GBQ, BQ, and TQ) on the performance of the IWPP based morphological reconstruction implementation – we do not show the results for the distance transform because there is no significant difference between these results and those from the distance transform implementation. The different queue versions are compared using a 4-connected grid and a $4K\times 4K$ input image tile with 100% tissue coverage. As a strategy to initialize the queue with different number of pixels and evaluate its performance under various scenarios, the number of raster and anti-raster scans performed in the initialization phase is varied. These experiments use 3 threads blocks per Stream Multiprocessor (SM) of the GPU, as this is sufficient for fully utilizing the GPU. Increasing or decreasing the number of thread blocks does not significantly affect the relative performance of the different

queue versions. The execution times reported in Table 1 correspond to the wavefront propagation phase only.

The results show that the GBQ version has the highest execution times in all the experiments. The BQ implementation was on average able to improve the performance by $2.25\times$ compared to the GBQ version. The relative performance gains with BQ for morphological reconstruction are smaller than those achieved in the previous section by the synthetic application that only performs dequeue and enqueue operations. This was expected because the morphological reconstruction algorithm performs computations on dequeued elements, thus the queue operations will correspond to a smaller fraction of the overall execution. These performance improvements with BQ indicate that atomic operations, while they are much faster in modern GPUs [35], still introduce significant overhead and that the heavy use of atomic operations should be considered carefully. The results in the table also show that the TQ version, which uses local queues to further reduce synchronization, achieved performance improvements of $1.2\times$ on top of BQ. These gains with the different versions of the queue demonstrate the benefits of employing prefix-sum to reduce the number of atomic operations and local queues to minimize synchronization among threads.

5.2.3. Use of *atomicMax* vs *atomicCAS* in Morphological Reconstruction—As discussed in Section 3.1, some algorithms based on the IWPP may be able to replace the propagation condition test and update of the input data domain via *atomicCAS* with a CUDA built-in atomic operation as *atomicMax* or *atomicMin*. It is, however, necessary that the propagation condition operation employed by the application correspond to an operation supported by an atomic built-in operation provided by CUDA. The advantage of using a built-in operation, as opposed to *atomicCAS*, is that the propagation condition test and the data update are guaranteed to be computed atomically. When *atomicCAS* is used, the propagation condition test and the data update are done in separate steps. However, the data update can occur only if the data element being updated in the input data domain has not changed its value since the calculation of the propagation condition. Thus, the use of *atomicCAS* makes it necessary to loop over the computation of the propagation condition and the data update via *atomicCAS* to guarantee the atomicity of the propagation (See Algorithm 4 in Section 3.1).

In this section, we measure the benefits of using the CUDA built-in *atomicMax* instead of *atomicCAS* in the morphological reconstruction implementation. The input data used by our algorithm was a $4K\times 4K$ data tile with 100% tissue coverage. The experimental results show that the *atomicMax* based version of the application achieved a speedup of $1.15\times$ that of the implementation using *atomicCAS*, representing a considerable improvement in performance. We expect that more atomic built-in operations will be made available with CUDA and that more data types, such as floating point numbers, will be supported. The rest of the experiments carried out in this manuscript use the most efficient *atomicMax* based version of the morphological reconstruction implementation.

The same evaluation is not possible for the distance transform, because CUDA does not offer an appropriate atomic built-in operation required for the propagation condition test in the distance transform implementation. In the distance transform algorithm, the value stored in the input data domain (VR diagram) is the address (“coordinates”) of the current nearest background pixel. However, the propagation condition is computed based on the euclidean distance between a pixel element and its current nearest background. The required built-in operation would be complex as it would have to decode the nearest background coordinates, compute the euclidean distance based on these coordinates, and compare the computed distance to the distance of a candidate for the new nearest background element.

5.2.4. Comparison to a Task-Parallel Model of Execution on GPUs—This section compares the performance of our implementation of the IWPP to the Task-Parallel execution model with dependency resolution (referred to as the *Task model*/here) proposed by Tzeng et. al [58]. The Task model was proposed as a way to deal with irregular workloads that both are composed of tasks with uneven computation demands and dispatch new tasks for computation during execution as dependencies are resolved. In the Task model, a global queue is employed to maintain the tasks to be computed, and a set of Workers consume tasks from that queue. In addition, as new tasks are created or dependencies are resolved, the new tasks are inserted into the global queue for computation. Each Worker consists of a CUDA thread block with the size of a single thread warp (32 threads).

We implemented the Task model algorithm proposed by Tzeng et. al to compare with our implementation of the IWPP. The global queue in the Task model algorithm is protected using a spinlock, implemented via an atomic compare-and-swap intrinsic operation (`while(atomicCAS(lock,0,1)==1)`), which guarantees coherence under concurrent Worker accesses. We used the morphological reconstruction algorithm as the use case. The implementation of this algorithm using the Task model employs the global queue to store elements that form the wavefront, and the Workers access the queue to retrieve wavefront elements for computation. Once these elements are processed, neighbor elements may be pushed to the queue as new tasks. In order to ensure that all threads in a Worker are used, each task request to the queue returns a set of 32 (Warp Size) wavefront elements, one for each thread in the respective Worker. Each thread in the Worker then computes the propagation on the neighborhood of its assigned wavefront element. Note that the morphological reconstruction does not have the dependency resolution phase, because all active task (elements forming the wavefront) can be computed in parallel and new tasks inserted to the queue are the result of computation of wavefront elements.

The performance evaluation of morphological reconstruction implementations using the IWPP and the Task model was done using 4K×4K image tiles with varying amounts of tissue coverage and, consequently, computation demands. The results, presented in Table 2, show that the execution times of the IWPP version of the morphological reconstruction were smaller than those of the Task model version for all configurations. The speedups achieved by the IWPP based morphological reconstruction on top of the Task model version were upto 80×. The large difference in performance is a consequence of the IWPP's scalable queue structure that can handle heavy loads. The lock mechanism used in the Task model is inefficient for cases where queue is intensively accessed [47]. It is worth noting, however, that the Task model [58] is an interesting abstraction on top of data parallel CUDA. It performs better under circumstance where the computation cost of a task is much higher than the overhead of accessing the queue in order to dequeue and enqueue tasks.

5.2.5. Effects of Tile Size—The experiments in this section evaluate the impact of the tile size on the performance of the morphological reconstruction and distance transform algorithms. Figure 11 presents the morphological reconstruction algorithm performance using an 8-connected *G* grid and an input image of 64K×64K pixels with 100% tissue coverage. Two versions of this algorithm were evaluated: FH_GPU that is implemented using the irregular wavefront propagation framework; and, SR_GPU [21] that is based on raster and anti-raster scans. SR_GPU was the fastest GPU implementation of morphological reconstruction available before our work. In addition, SR_GPU operates by sweeping over the input data domain using raster and anti-raster scans. This is equivalent to performing wavefront propagations without tracking the active elements forming the wavefront. Thus, the comparison of SR_GPU to our algorithm is important in order to understand the

performance of different strategies to execute wavefront propagations on GPUs. All speedup values are calculated using the fast FH single core CPU version as a reference.

The results show that the performances of both GPU algorithms improve as the tile size is increased until 16K×16K pixels; beyond this size no significant additional gains are observed. The better speedup values with larger tiles are the result of (i) better amortization of initialization of computations with the GPU, specially in the raster scan pass (SR_GPU and the initialization phase of FH_GPU), which launches four GPU *kernels* per iteration; and (ii) the larger amount of parallelism available with big tiles. The experimental results show that FH_GPU performs better than SR_GPU [21] in all the configurations, on average about 2.6× better. Further, FH_GPU achieves a performance improvement of 14.3× compared to the single core CPU version. It is important to highlight, however, that the FH_GPU is a two phase algorithm that includes raster scans in the initialization phase. The initial scan is responsible for about 43% of the execution time, reducing performance gains from the wavefront propagation phase.

The performance of the distance transform implementation is presented in Table 3. The initialization phase in this algorithm is inexpensive. Most of the execution time is spent in the wavefront propagation phase, which is less sensitive to the tile size because of its lower overheads – all of the computation in this phase is performed within a single GPU *kernel*.

5.2.6. Impact of Input Data Characteristics—These experiments employed four different images with 64K×64K pixels, shown in Figure 12, that have different percentage of tissue coverage. The 25%, 50%, 75%, and 100% of the image area is covered with tissue (in contrast to background) in images 1, 2, 3, and 4, respectively. The images are partitioned into tiles of size 16K×16K to fit into the GPU memory. Inter-partition propagations are performed as discussed in Section 4.

The speedup values of FH_GPU and SR_GPU with respect to the single core CPU version are presented in Figure 13. The performances of the GPU implementations get better with images with larger tissue coverage. Larger tissue coverage implies more work that can be done in parallel, which amortizes the overhead of starting the GPU computations. In addition, the higher parallelism for images with larger tissue coverage increases the utilization of GPU threads and reduces the chances of having conflicting atomic operations between threads.

The comparison of FH_GPU and SR_GPU shows a consistent performance gap between the implementations for all variations of the input images. In our experiments, FH_GPU achieved performance improvements of up to 3.1× on top of SR_GPU. The performance gains of FH_GPU are higher for images with less tissue coverage. For those images, SR_GPU passes over the entire input image domain irrespective of the number of pixels being modified, resulting in more unnecessary computation that does not contribute to the output.

The performance of the distance transform implementation as the input data is varied is presented in Figure 14. As in the morphological reconstruction implementation, larger tissue coverage results in higher speedup values. Once again, this is a consequence of more computation and parallelism available with images with more tissue coverage. More tissue coverage means there are more foreground pixels from which to calculate distances to a background pixel.

5.2.7. Impact of Exceeding Queue Storage Limit—This set of experiments vary the memory space available for the queue to store pixels. When the queue storage limit is

exceeded, our implementation throws away excess pixels. Then it is necessary to perform another round of computations to carry out any missing wavefront propagations. To stress the algorithm, we reduced the space available for the queue until we created two scenarios. In the first scenario, the queue storage limit is exceeded once, and the wavefront propagation execution phase has to be re-launched to fix any missing propagations. In the second scenario, the queue storage limit is exceeded in the second iteration of the wavefront propagation phase as well. As a result, a third iteration of this phase is needed to finish computation correctly.

The experimental results were obtained using the morphological reconstruction implementation and a single input tile of 4K×4K pixels. The results show that performance penalty due to exceeding the queue storage limit is small. For instance, the performance hit is 6% when the queue is exceeded once during the execution and 9% when it is exceeded twice. In general, we observed that setting a storage limit that is 10% larger than the initial queue size was sufficient.

5.3. Cooperative Multi-GPU Multi-CPU Execution

These experiments evaluate the implementations when multiple GPUs and CPUs are used cooperatively for execution. The input images have 100% tissue coverage. The resolutions of the images are 96K×96K pixels and 64K×64K pixels for the morphological reconstruction implementation and the distance transform implementation, respectively; these are the maximum size images that can fit in the main CPU memory for each algorithm.

The multi-GPU scalability of the morphological reconstruction algorithm is presented in Figure 15(a). The speedups achieved for one, two, and three GPUs, in comparison to the CPU single core version, are about 16×, 30× and 43×. The overall performance improvement is good, but the 3-GPUs execution achieves 2.67× speedup with respect to the 1-GPU version which is below linear. To investigate the reasons, we calculated the average cost of the wavefront propagation phase by breaking it down into three categories: (i) *Computation*, which is the time spent by the application *kernels*; (ii) *Download*, which corresponds to the time to download the results from GPU to CPU memory; and (iii) *Upload*, which includes data copy from CPU to GPU. The results, presented in Figure 15(b), show that there is an increase in data transfer costs as more GPUs are used, limiting the scalability. We should note that we efficiently utilized the architecture of the node, which is built with two I/O Hubs connecting CPUs to GPUs (see Figure 8). In our approach, the CPU thread managing a GPU is mapped to a CPU closer to the corresponding GPU. This allows for maximum performance to be attained during data transfers. When this placement is not employed, the performance of the multi-GPU scalability is degraded and only small performance improvements are observed when more than two GPUs are utilized.

Figure 15(a) also presents the cooperative CPU–GPU execution performance. In this scenario, when the multi-core version of the CPU operation is used (parallel variant), the multi-CPU multi-GPU executions achieve extra performance improvements of about 1.17× with respect to the 3-GPU configuration, or a total of 50× speedup on top of the single core CPU execution. As is shown in the figure, the use of sequential CPU variants results in performance loss with respect to the multi-GPU only execution. This performance degradation is the result of load imbalance among devices, which is higher with the sequential variants due to the larger gap in execution times between a GPU execution and a single CPU core. The computation time for the 96K×96K pixel image in the cooperative execution mode using all the CPUs and GPUs was 21 seconds.

Finally, Figure 16 presents the cooperative multi-GPU/-CPU execution results for distance transform. Similar to that of the morphological reconstruction implementation, the multi-

GPU scalability is good, but it is sub-linear because of increasing data transfer costs. The cooperative CPU—GPU execution, in this case, resulted in a speedup of 1.07× on top of the execution with 3 GPUs. The smaller improvement percentage, as compared to the morphological reconstruction algorithm, is simply because of the better GPU acceleration achieved by distance transform. The overall speedups of the algorithm when all GPUs and CPU cores are employed together is about 85.6×, as compared to the single CPU core execution. For reference, the distance transform computation on the 64K×64K pixel image took only 4.1 seconds.

6. Related work

The irregular wavefront propagation pattern has similarities to some fundamental graph scan algorithms, such as Breadth-First Search (BFS), but modified to support multiple sources. Much effort has been put into the development of parallel implementations of BFS on multi-core CPUs [1] and GPUs [18, 30]. The GPU implementation proposed in [30] is based on a sequence of iterations until all nodes are visited. Each iteration visits all nodes on the graph and processes those that are on the propagation frontier. The work by Hong et al. [18] targeted new techniques to avoid load imbalance when nodes in a graph have a highly irregular cardinality (the number of edges per node). In image operations with the IWPP, the approach proposed by Hong et al. would not do any better than the approach suggested by Karas [21], since the degree of all nodes (pixels) is the same and defined by the structuring element G . Moreover, the irregular wavefront propagation based approach visits the nodes (data elements) that are modified – i.e., enqueued in the last iteration –, instead of all the nodes to compute only those that are in the frontier [18].

The queue in our case is heavily employed and hundreds of thousands to millions of nodes (pixels) may be queued per iteration. The solution introduced in the earlier work [30], if employed in the context of the IWPP, would require expensive inter-block communication at each iteration of the application. This synchronization is realized by synchronizing all thread blocks, stopping the *kernel* execution, and re-launching another round of computations. Unlike the previous work [30], we propose that a scalable hierarchical queue be employed in a per-block basis. This avoids communication among blocks and permits the entire computation to be efficiently carried out in a single *kernel* call. Moreover, since the connectivity of a pixel is known a priori, we can employ local queues more aggressively to provide very fast local storage, as well as use parallel prefix-sum based reductions to avoid expensive atomic operations when storing items into the queue. Additionally, in our solution, an arbitrarily large number of blocks may be used, which enables the implementation to leverage the GPU's dynamic block scheduler in order to alleviate the effects of inter-block load imbalance.

A variety of image processing algorithms have been ported to GPUs for efficient execution [15, 25, 45]. In addition, there is a growing set of libraries, such as the NVIDIA Performance Primitives (NPP) [36] and OpenCV [5], which encapsulate GPU-implementations of several image processing algorithms through high level and flexible APIs. Most of the algorithms currently supported, however, exhibit more regular computation patterns as compared to the IWPP.

Morphological reconstruction has been increasingly employed in several applications in the last decade. Efficient algorithms [62, 27] and a number of applications were presented by Vincent [60]. Other variants of Vincent's fast hybrid algorithm (FH) have been proposed, such as the downhill filter approach [40]. We have tested this approach on our datasets and found it to be slower than FH. The high computational demands of morphological reconstruction motivated other works that employed specialized processors to speedup this

algorithm. Jivet et al. [20] implemented a version using FPGAs, while Karas et al. [21] targeted GPUs. In both cases, however, the parallel algorithms were designed on top of a sequential baseline version that is intrinsically parallelizable, but about one order of magnitude slower than the fastest sequential version. In our solution, on the other hand, we extend the fastest baseline algorithm which exhibits the irregular wavefront propagation pattern. This approach has resulted in strong and consistent performance improvements over the existing GPU implementation by Karas [21].

The Euclidean distance transform is also important in image analysis and used in several tasks as watershed based segmentation. This algorithm is computationally very demanding, which has attracted considerable attention in the last few years regarding strategies for GPU acceleration [42, 41, 44, 7, 24]. Kong et. al [24] presented the acceleration of a crucial set of operations from the MAT-LAB image processing toolbox, including the exact euclidean distance transform. This set of operations was carefully classified according to the parallelism exhibited, which allowed for the authors to better understand and explain the differences in acceleration achieved by those operations. Since their euclidean distance transform implementation is not based on neighborhood propagation, it is not possible to directly compare it to the distance transform implemented in our framework. Among the proposed solutions, the work of Schneider et al. [44] is the most similar to our distance transform algorithm, because it also implements an output that is equivalent to that of Danielsson's distance transform [11]. Their GPU implementation, however, is based on sweeps over the entire dataset, instead of using propagations that process only the elements in the wavefront as in our case. Moreover Schneider's algorithm only performs propagation of pixels in one row at time, considering 2D problems, which limits GPU utilization. Unfortunately, this code is not available for comparison, but the levels of acceleration achieved in our case are higher than what Schneider's algorithm attained. No multi-CPU multi-GPU algorithms have been developed previously for the morphological reconstruction and euclidean distance transform operations; our work is the first one to use multiple GPUs and CPUs.

The use of CPUs and GPUs cooperatively has gained attention of the research community in the last few years. Several works have developed system and compiler techniques to support efficient execution on heterogeneous CPU-GPU equipped machines [28, 29, 54, 48, 50, 4, 39, 19, 17, 49, 31]. In this paper, we present a domain specific framework for the parallelization of irregular wavefront propagation based applications on CPU-GPU equipped machines. This framework makes the computation power of such environments available to a class of applications by exporting a high level set of tools, which can be used by a programmer to accelerate applications fitting in the irregular wavefront propagation pattern.

The cooperative CPU-GPU parallelization strategy used in this work leverages the use of multi-core CPU variants as a way to reduce the performance gap and, consequently, potential load imbalance between CPUs and GPUs. Another approach for reducing load imbalance between these devices would be the use of techniques such as dynamic partition of the input data domain into different tile sizes [12, 17]. With this strategy, tiles sizes for CPUs and GPUs would be calculated considering the relative performance of the devices so that tiles assigned to each of the devices would take approximately the same computation time. Though this strategy is interesting and have ample applicability, it would be very challenging to use with the IWPP, because the computation cost of regions in the input data domain are data dependent and typically irregularly distributed. Since this information is not available before execution to be used as a tiling metric, the tuning of the tile size would be error prone and unstable.

7. Conclusions

The efficiency of the *irregular wavefront propagation pattern* relies on tracking points in the wavefront and, consequently, avoiding unnecessary computation to visit and process parts of the domain that do not contribute to the output. This characteristic of the IWPP makes it an efficient computation structure in image analysis and other applications [62, 60, 63, 32, 26, 61, 38, 16, 57]. Nonetheless, the parallelization of the IWPP on hybrid systems with multi-core CPUs and GPUs is complicated because of the dynamic, irregular, and data dependent data access and processing pattern. Our work shows that a multi-level queue to track the wavefronts provides an efficient data structure for execution on a GPU. The multi-level queue structure enables the efficient use of fast memory hierarchies on a GPU. Inter- and intra-thread block scalability can be achieved by per block queues and parallel prefix-sum index calculation to avoid heavy use of atomic operations. Our work also shows that a tile based parallelization is critical to achieve scalability on multi-core and multi-GPU systems, since it reduces inter-thread interferences because of computation of the same regions of the domain without tiling. Inter-thread interferences cause inefficient use of cache.

In order to evaluate the proposed approaches we have developed the IWPP-based implementations of two widely used image processing operations, morphological reconstruction and euclidean distance transform, on a state-of-the-art hybrid machine. In our experimental evaluation, these implementations have achieved orders of magnitude performance improvement (of upto 85×) over the sequential CPU versions. Our results showed that coordinated use of CPUs and GPUs makes it feasible to process high resolution images in reasonable times, opening the way for large scale imaging studies.

One limitation of our current implementation is that it does not support the use of distributed multi-node machines. The parallelization strategy we employ for multiple GPUs and CPUs (Section 4), however, can be extended for the multi-node case. In this scenario, a message passing mechanism should be employed to exchange information necessary for propagations to cross partitions of the input domain assigned to different machines.

Acknowledgments

The authors would like to express their gratitude to the anonymous reviewers for their valuable comments, which helped us to improve the presentation of this paper. This research was funded, in part, by grants from the National Institutes of Health through contract HHSN261200800001E by the National Cancer Institute; and contracts 5R01LM009239-04 and 1R01LM011119-01 from the National Library of Medicine, R24HL085343 from the National Heart Lung and Blood Institute, NIH NIBIB BISTI P20EB000591, RC4MD005964 from National Institutes of Health, and PHS Grant UL1TR000454 from the Clinical and Translational Science Award Program, National Institutes of Health, National Center for Advancing Translational Sciences. This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH. We also want to thank Pavel Karas for releasing the SR_GPU implementation used in our comparative evaluation.

References

1. Agarwal, V.; Petrini, F.; Pasetto, D.; Bader, DA. Scalable Graph Exploration on Multicore Processors.. Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10; 2010;
2. Ancourt, C.; Irigoin, F. Proceedings of the Third ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPOPP '91. ACM; New York, NY, USA: 1991. Scanning polyhedra with DO loops.; p. 39-50.
3. Banerjee, U. Advances in Languages and Compilers for Parallel Computing. Pitman Publishing; London, UK: 1991. Unimodular Transformation of Double Loops.; p. 192-219.

4. Bosilca, G.; Bouteiller, A.; Herault, T.; Lemarinier, P.; Saengpatsa, N.; Tomov, S.; Dongarra, J. Performance Portability of a GPU Enabled Factorization with the DAGuE Framework.. 2011 IEEE International Conference on Cluster Computing (CLUSTER); sept. 2011; p. 395-402.
5. Bradski G. The OpenCV Library. Dr. Dobb's Journal of Software Tools. 2000
6. Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M, Hanrahan P. Brook for gpus: stream computing on graphics hardware. ACM Trans. Graph. 2004; 23(3):777-786.
7. Cao, T-T.; Tang, K.; Mohamed, A.; Tan, T-S. Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, I3D '10. ACM; New York, NY, USA: 2010. Parallel Banding Algorithm to compute exact distance transform with the GPU.; p. 83-90.
8. Cuisenaire O, Macq B. Fast euclidean distance transformation by propagation using multiple neighborhoods. Computer Vision and Image Understanding. 1999; 76(2):163-172.
9. Torres, R. da S.; Falcao, A.; Costa, L. da F. A graph-based approach for multiscale shape analysis. Pattern Recognition. 2004; 37(6):1163-1174.
10. Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming. IEEE Computational Science and Engineering. Jan-Mar;1998 5(1):46-55.
11. Danielsson P-E. Euclidean distance mapping. Computer Graphics and Image Processing. 1980; 14:227-248.
12. Di, P.; Xue, J. Proceedings of the 17th international conference on Parallel processing - Volume Part II, Euro-Par'11. Springer-Verlag; Berlin, Heidelberg: 2011. Model-driven tile size selection for doacross loops on gpus.; p. 401-412.
13. Fabbri R, Costa LDF, Torelli JC, Bruno OM. 2D Euclidean distance transform algorithms: A comparative survey. ACM Comput. Surv. Feb.2008 40(1):2:1-2:44.
14. Falcão AX, da Fontoura Costa L, Cunha BSSD. Multiscale skeletons by image foresting transform and its application to neuromorphometry. Pattern Recognition. 2002; 35(7):1571-1582.
15. Fialka O, Cadik M. FFT and Convolution Performance in Image Filtering on GPU. Proceedings of the conference on Information Visualization. 2006:609-614.
16. Gabriel RK, Sokal RR. A New Statistical Approach to Geographic Variation Analysis. Systematic Zoology. Sept.1969 18(3):259-278.
17. Hartley, TD.; Catalyurek, UV.; Ruiz, A.; Ujaldon, M.; Igual, F.; Mayo, R. Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores.. 22nd ACM Intl. Conference on Supercomputing; Dec 2008;
18. Hong, S.; Kim, SK.; Oguntebi, T.; Olukotun, K. Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming, PPOPP '11. ACM; 2011. Accelerating CUDA graph algorithms at maximum warp.; p. 267-276.
19. Huo, X.; Ravi, V.; Agrawal, G. Porting irregular reductions on heterogeneous CPU-GPU configurations.. 18th International Conference on High Performance Computing (HiPC); dec. 2011; p. 1-10.
20. Jivet I, Brindusescu A, Bogdanov I. Image contrast enhancement using morphological decomposition by reconstruction. WSEAS Trans. Cir. and Sys. Aug.2008 7:822-831.
21. Karas, P. *MEMICS*, volume 16 of *OASICS*. Schloss Dagstuhl -Leibniz-Zentrum fuer Informatik; Germany: 2010. Efficient Computation of Morphological Greyscale Reconstruction.; p. 54-61.
22. Kernel, L. Perf profiling tool. GNU; 2012.
23. Kohn, SR.; Baden, SB. Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM), Supercomputing '95. ACM; New York, NY, USA: 1995. A parallel software infrastructure for structured adaptive mesh methods..
24. Kong, J.; Dimitrov, M.; Yang, Y.; Liyanage, J.; Cao, L.; Staples, J.; Mantor, M.; Zhou, H. Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU '10. ACM; New York, NY, USA: 2010. Accelerating MATLAB Image Processing Toolbox functions on GPUs.; p. 75-85.
25. Körbes A, Vitor GB, de Alencar Lotufo R, Ferreira JV. Advances on watershed processing on GPU architecture. Proceedings of the 10th International Conference on Mathematical Morphology, ISMM'11. 2011:260-271.

26. Lantuéjoul C, Haralick R, Simon J-C. Skeletonization in quantitative metallography. *Issues in Digital Image Processing*, volume 34 of *NATO ASI Series E*. 1980:107–135.
27. Laurent, C.; Roman, J. Third International Conference on Vector and Parallel Processing, VECPAR '98. Springer-Verlag; London, UK, UK: 1999. Parallel implementation of morphological connected operators based on irregular data structures.; p. 579-592.
28. Linderman MD, Collins JD, Wang H, Meng TH. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.* 2008; 43(3):287–296.
29. Luk C-K, Hong S, Kim H. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. 42nd International Symposium on Microarchitecture (MICRO). 2009
30. Luo L, Wong M, Hwu W.-m. An effective GPU implementation of breadth-first search. *Proceedings of the 47th Design Automation Conference, DAC '10*. 2010:52–55.
31. Mariano, A.; Alves, R.; Barbosa, J.; Santos, L.; Proenca, A. A (ir)regularity-aware task scheduler for heterogeneous platforms.. *International Conference on High Performance Computing*; 2012;
32. Meyer F. Digital Euclidean Skeletons. *SPIE Vol. 1360, Visual Communications and Image Processing*. 1990:251–262.
33. Millstein T. Practical predicate dispatch. *SIGPLAN Not.* Oct.2004 39:345–364.
34. Mullikin JC. The vector distance transform in two and three dimensions. *CVGIP: Graphical Models and Image Processing*. 1992; 54(6):526–535.
35. Nvidia. Fermi Compute Architecture Whitepaper.
36. NVIDIA. NVIDIA Performance Primitives(NPP). Feb 11.2011
37. Pennycook, SJ.; Mudalige, GR.; Hammond, SD.; Stephen A, -J. 26th UK Performance Engineering Workshop (UKPEW10). Performance Computing and Visualisation, Department of Computer Science, University of Warwick; Jul. 2010 Parallelising wavefront applications on general-purpose GPU devices.; p. 111-118.
38. Preparata, FP.; Shamos, MI. Computational geometry: an introduction. Springer-Verlag New York, Inc.; New York, NY, USA: 1985.
39. Ravi, V.; Ma, W.; Chiu, D.; Agrawal, G. Proceedings of the 24th ACM International Conference on Supercomputing. ACM; 2010. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations.; p. 137146
40. Robinson K, Whelan PF. Efficient morphological reconstruction: a downhill filter. *Pattern Recogn. Lett.* Nov.2004 25:1759–1767.
41. Rong G, Tan T-S. Jump flooding in GPU with applications to Voronoi diagram and distance transform. *Proceedings of the 2006 symposium on Interactive 3D graphics and games, I3D '06*. 2006:109–116.
42. Rong G, Tan T-S. Variants of Jump Flooding Algorithm for Computing Discrete Voronoi Diagrams. *Proceedings of the 4th International Symposium on Voronoi Diagrams in Science and Engineering, ISVD '07*. 2007:176–181.
43. Saltz JH, et al. Multi-scale, integrative study of brain tumor: In silico brain tumor research center. Annual Symposium of American Medical Informatics Association 2010 Summit on Translational Bioinformatics (AMIA-TBI 2010). Mar.2010
44. Schneider, J.; Kraus, M.; Westermann, R. GPU-based Real-time Discrete Euclidean Distance Transforms with Precise Error Bounds.. In: Ranchordas, A.; Araújo, H., editors. *VISAPP (1)*. INSTICC Press; 2009. p. 435-442.
45. Scholl I, Aach T, Deserno TM, Kuhlen T. Challenges of medical image processing. *Comput. Sci.* Feb.2011 26:5–13.
46. Shih FY, Wu Y-TWY-T. The efficient algorithms for achieving euclidean distance transformation. *IEEE Transactions on Image Processing*. 2004; 13(8):1078–1091. [PubMed: 15326850]
47. Stuart JA, Owens JD. Efficient Synchronization Primitives for GPUs. *CoRR*. 2011 abs/1110.4623.
48. Sundaram N, Raghunathan A, Chakradhar ST. A framework for efficient and scalable execution of domain-specific templates on GPUs. *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*. 2009:1–12.
49. Teodoro G, Hartley T, Catalyurek U, Ferreira R. Optimizing dataflow applications on heterogeneous environments. *Cluster Computing*. 2012; 15:125–144.

50. Teodoro G, Hartley TDR, Catalyurek U, Ferreira R. Run-time optimizations for replicated dataflows on heterogeneous environments. Proc. of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC). 2010
51. Teodoro G, Kurc TM, Pan T, Cooper LA, Kong J, Widener P, Saltz JH. Accelerating Large Scale Image Analyses on Parallel, CPU-GPU Equipped Systems. IPDPS '12: Proceedings of the 2012 IEEE International Symposium on Parallel and Distributed Processing. 2012
52. Teodoro, G.; Pan, T.; Kurc, TM.; Kong, J.; Cooper, L.; Saltz, JH. Technical Report CCI-TR-2012-2. Emory University; Jan.. 2012 A Fast Parallel Implementation of Queue-based Morphological Reconstruction using GPUs..
53. Teodoro, G.; Pan, T.; Kurc, TM.; Kong, J.; Cooper, L.; Saltz, JH. Technical Report CCI-TR-2012-4. Emory University; May. 2012 High-throughput Execution of Hierarchical Analysis Pipelines on Hybrid Cluster Platforms..
54. Teodoro G, Sachetto R, Sertel O, Gurcan M, W. M. Catalyurek U, Ferreira R. Coordinating the use of GPU and CPU for improving performance of compute intensive applications. In IEEE Cluster. 2009
55. Tomov, S.; Nath, R.; Ltaief, H.; Dongarra, J. Dense Linear Algebra Solvers for Multicore with GPU Accelerators.. IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing; 2010;
56. Torres R, Falcao A, Costa L. Shape description by image foresting transform. 2002 14th International Conference on Digital Signal Processing, 2002. DSP 2002. 2002; 2:1089–1092.
57. Toussaint GT. The relative neighbourhood graph of a finite planar set. Pattern Recognition. 1980; 12(4):261–268.
58. Tzeng S, Lloyd B, Owens JD. A GPU Task-Parallel Model with Dependency Resolution. Computer. 2012; 45:34–41.
59. Vetter JS, Glassbrook R, Dongarra J, Schwan K, Loftis B, Mc-Nally S, Meredith J, Rogers J, Roth P, Spafford K, Yalamanchili S. Keeneland: Bringing Heterogeneous GPU Computing to the Computational Science Community. Computing in Science and Engineering. 13:2011.
60. Vincent L. Exact Euclidean Distance Function By Chain Propagations. IEEE Int. Computer Vision and Pattern Recog. Conference. 1991:520–525.
61. Vincent, L. Proc. of the Workshop "Shape in Picture". Springer; 1992. Morphological area openings and closings for grey-scale images.; p. 197-208.
62. Vincent L. Morphological grayscale reconstruction in image analysis: Applications and efficient algorithms. IEEE Transactions on Image Processing. 1993; 2:176–201. [PubMed: 18296207]
63. Vincent L, Soille P. Watersheds in Digital Spaces: An Efficient Algorithm Based on Immersion Simulations. IEEE Transactions on Pattern Analysis and Machine Intelligence. Jun; 1991 13(6): 583–598.
64. Wolfe M. Loops skewing: The wavefront method revisited. International Journal of Parallel Programming. 1986; 15:279–293. 10.1007/BF01407876.
65. Xiao S, chun Feng W. Inter-block GPU communication via fast barrier synchronization. IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing. 2010:1–12.

Figure 1.

Gray scale morphological reconstruction in 1-dimension. The marker image intensity profile is represented as the red line, and the mask image intensity profile is represented as the blue line. The final image intensity profile is represented as the green line. The arrows show the direction of propagation from the marker intensity profile to the mask intensity profile. The green region shows the changes introduced by the morphological reconstruction process.

Figure 2.

Overview of the IWPP propagation phase on a GPU: (i) in the initialization phase the input data domain is processed, computing the propagation condition for each element, in order to identify the propagation seeds (i.e., the elements in the initial wavefront); (ii) the N propagation seeds selected in the previous stage are divided into Z independent partitions; (iii) a CUDA thread block is then created and mapped to each of the Z partitions. Each thread block instantiates an independent queue to perform the wavefront propagation starting from the seeds assigned to it. Therefore, there is no synchronization among queues managed by different blocks, since they are different queue instances.

Figure 3. Multi-level parallel queue scheme employed for efficient addition of elements (Add to queue operation).

Figure 4.
Tiling based parallelization (tiles are not necessarily square).

Figure 5.
Evaluating Scalability According to the Number of Blocks.

Figure 6.

Pipeline representation of the CPU-GPU wavefront propagation. Instances of Tile-based Propagation (TP) compute propagations in each tile as shown at the top. A Border Propagation (BP) stage, which depends on the TP instances for execution, facilitates propagation among the tiles. If the waves propagate between the neighbor tiles, BP re-instantiates the pipeline. This process continues until stability is reached. The multi-level tiling strategy, in which a tile is repartitioned into micro-tiles for execution on multiple CPU cores, is shown at the bottom. This strategy is implemented to reduce CPU vs GPU computational load imbalance.

Figure 7.
System support for execution of pipeline of tasks in machines equipped with multi-core CPUs and GPUs.

Figure 8.
Hybrid, Multi-GPU computing node.

Figure 9. Multicore performance of morphological reconstruction. Different parallelization strategies are compared.

Figure 10. Elementary performance of queue operations: the *kernel* application that dequeues and enqueues a specified number of data elements.

Figure 11. Morphological Reconstruction: performance of the IWPP based implementation (FH_GPU) and Pavas algorithm [21] (SR_GPU) that is implemented as iterative raster scanning according to the input data tile size.

Figure 12.

Raw image, marker, and mask for tiles with different amount of tissue coverage.

Figure 13.
Performance variation of the morphological reconstruction implementation with respect to input data characteristics.

Figure 14.
Performance variation of the distance transform implementation with respect to input data characteristics.

Figure 15.

Performance of the morphological reconstruction implementation on multiple GPUs and using multiple CPUs and GPUs together.

Figure 16. Performance of the distance transform implementation on multiple GPUs and using multiple CPUs and GPUs together.

Algorithm 1**Irregular Wavefront Propagation Pattern (IWPP)**

```

1:  $D \leftarrow$  data elements in a multi-dimensional space
2: {Initialization Phase}
3:  $S \leftarrow$  subset active elements from  $D$ 
4: {Wavefront Propagation Phase}
5: while  $S \neq \emptyset$  do
6:   Extract  $e_i$  from  $S$ 
7:    $Q \leftarrow N_G(e_i)$ 
8:   while  $Q \neq \emptyset$  do
9:     Extract  $e_j$  from  $Q$ 
10:    if  $\text{PropagationCondition}(D(e_i), D(e_j)) = \text{true}$  then
11:       $D(e_j) \leftarrow \text{Update}(D(e_j))$ 
12:      Insert  $e_j$  into  $S$ 

```

Algorithm 2**Fast Hybrid Gray scale Morphological Reconstruction Algorithm****Input***I: mask image**J: marker image.*

```

1:  {Initialization Phase}
2:  Scan I and J in raster order.
3:  Let p be the current pixel
4:   $J(p) \leftarrow \left( \max \{ J(q), q \in N_G^+(p) \cup \{p\} \} \right) \wedge I(p)$ 
5:  Scan I and J in anti-raster order.
6:  Let p be the current pixel
7:   $J(p) \leftarrow \left( \max \{ J(q), q \in N_G^-(p) \cup \{p\} \} \right) \wedge I(p)$ 
8:  if  $\exists q \in N_G^-(p) \mid J(q) < J(p) \text{ and } J(q) < I(q)$ 
9:    queue.enqueue(p)
10: {Wavefront Propagation Phase}
11: while queue.empty() = false do
12:   p ← queue.dequeue()
13:   for all q ∈ NG(p) do
14:     if  $J(q) < J(p)$  and  $J(q) < I(q)$  then
15:        $J(q) \leftarrow \min \{ J(p), I(q) \}$ 
16:       queue.enqueue(q)

```

Algorithm 3**Wavefront Propagation-based Distance Transform Algorithm**

Input: I : mask image
Output: M : distance map image
 FG : Foreground; BG : Background

```

1:  {Initialization Phase}
2:  for all  $p \in D_I$  do
3:     $VR(p) = (p == BG) ? p : inf$ 
4:    if  $I(p) == BG$  and  $\exists q \in N_G(p) \mid I(q) == FG$  then
5:      queue.enqueue(p)
6:  {Wavefront Propagation Phase}
7:  while queue.empty() = false do
8:     $p \leftarrow$  queue.dequeue()
9:    for all  $q \in N_G(p)$  do
10:     if  $DIST(q, VR(p)) < DIST(q, VR(q))$  then
11:        $VR(q) = VR(p)$ 
12:       queue.enqueue(q)
13:  for all  $p \in D_I$  do
14:     $M(p) = DIST(p, VR(p))$ 

```

Algorithm 4GPU Wavefront propagation *kernel*.

```

1:  while queue[blockId].empty() = false do
2:    while ( $p = \text{queue}[\text{blockId}].\text{dequeue}(\dots) \neq \emptyset$ ) do in parallel
3:      for all  $q \in N_G(p)$  do
4:        repeat
5:           $\text{curValueQ} = \mathbb{I}(q)$ 
6:          if  $\text{PropagationCondition}(\mathbb{I}(p), \text{curValueQ})$  then
7:             $\text{oldval} = \text{atomicCAS}(\&\mathbb{I}(q), \text{curValueQ}, \text{op}(\mathbb{I}(p)))$ 
8:            if  $\text{oldval} == \text{curValueQ}$  then
9:              queue[blockId].enqueue(q)
10:             break;
11:          else
12:            break;
13:        until True
14:    queue[blockId].swap_in_out()

```

Algorithm 5

Fast Parallel Hybrid Reconstruction

Input

I : mask image

J : marker image, defined on domain D_I , $J \leq I$.

```

1:  {Initialization phase}
2:  Scan  $D_I$  in raster order.
3:  for all  $rows \in D_I$  do in parallel
4:    Let  $p$  be the current pixel
5:     $J(p) \leftarrow \left( \max \{ J(q), q \in N_G^{+,0}(p) \cup \{p\} \} \right) \wedge I(p)$ 
6:  for all  $columns \in D_I$  do in parallel
7:    Let  $p$  be the current pixel
8:     $J(p) \leftarrow \left( \max \{ J(q), q \in N_G^{0,+}(p) \cup \{p\} \} \right) \wedge I(p)$ 
9:  Scan  $D_I$  in anti-raster order.
10: for all  $rows \in D_I$  do in parallel
11:   Let  $p$  be the current pixel
12:    $J(p) \leftarrow \left( \max \{ J(q), q \in N_G^{-,0}(p) \cup \{p\} \} \right) \wedge I(p)$ 
13: for all  $columns \in D_I$  do in parallel
14:   Let  $p$  be the current pixel
15:    $J(p) \leftarrow \left( \max \{ J(q), q \in N_G^{0,-}(p) \cup \{p\} \} \right) \wedge I(p)$ 
16: for all  $p \in D_I$  do in parallel
17:   if  $\exists q \in N_G(p) \mid J(q) < J(p)$  and  $J(q) < I(q)$ 
18:      $S.add(p)$ 
19: {Wavefront propagation phase}
20: Split  $S$  into  $Z$  partitions
21: while  $queue[blockId].empty() = false$  do
22:   while  $(p = queue[blockId].dequeue(...)) \neq \emptyset$  do in parallel
23:     for all  $q \in N_G(p)$  do
24:       if  $J(q) < J(p)$  and  $I(q) = J(q)$  then
25:          $oldval = atomicMax(\&J(q), min\{J(p), I(q)\})$ 
26:         if  $oldval < min\{J(p), I(q)\}$  then
27:            $queue[blockId].enqueue(q)$ 

```

Algorithm 6

Distance Transform Algorithm for GPUs

Input: I : mask image
Output: M : distance map image
FG: Foreground; *BG*: Background

```

1:  {Initialization phase}
2:  for all  $p \in D_I$  do in parallel
3:     $VR(p) = (I(p) == BG) ? p : inf$ 
4:    if  $I(p) == BG$  and  $\exists q \in N_G(p) \mid I(q) == FG$ 
5:       $S.add(p)$ 
6:  {Wavetront propagation phase}
7:  Split  $S$  into  $Z$  partitions
8:  while queue[blockId].empty() = false do
9:    while ( $p = queue[blockId].dequeue(...)$ )  $\neq \emptyset$  do in parallel
10:     for all  $q \in N_G(p)$  do
11:       repeat
12:          $curVRQ = VR(q)$ 
13:         if  $DIST(q, VR(p)) < DIST(q, curVRQ)$  then
14:            $old = atomicCAS(\&VR(q), curVRQ, VR(p))$ 
15:           if  $old == curVRQ$  then
16:             queue[blockId].enqueue(q)
17:             break;
18:         else
19:           break
20:       until True
21: for all  $p \in D_I$  do in parallel
22:    $M(p) = DIST(p, VR(p))$ 

```

Table 1

Execution times in milliseconds (ms) for GBQ, GBQ+BQ (BQ), and BGQ+BQ+TQ (TQ). The columns *Initial* and *Total* indicate the number of pixels initially queued before the wavefront propagation phase is executed and the total number of pixels enqueued/dequeued during the execution, respectively.

Initial	Total	#Raster Scans	GBQ	BQ	TQ
534K	30.4M	7	341	149	125
388K	22.2M	9	262	114	95
282K	16.7M	11	203	84	68
210K	12.7M	13	155	68	57
158K	9.9M	15	119	60	50

Table 2

Morphological Reconstruction: comparison of the IWPP performance with a Task-Parallel model on a GPU.

Image Types (% of tissue coverage)	25	50	75	100
Task model (time ms)	1,644	3,229	10,593	11,714
IWPP (time ms)	28	49	132	148

Table 3

Distance Transform: performance according to the input data tile size.

Tile Size	4K×4K	8K×8K	16K×16K	32K×32K
Speedup	31.4	36.8	37.5	37.1