Published in final edited form as:
Proc Symp Comput Archit High Perform Comput. 2015 October ; 2015: 25–32. doi:10.1109/SBAC-PAD.2015.13.

Efficient irregular wavefront propagation algorithms on Intel® Xeon Phi™

Jeremias M. Gomes,
University of Brasília, Brasília, DF, Brazil

George Teodoro,
University of Brasília, Brasília, DF, Brazil

Alba de Melo,
University of Brasília, Brasília, DF, Brazil

Jun Kong,
Emory University, Atlanta, GA, USA

Tahsin Kurc, and
Stony Brook University

Joel H. Saltz
Stony Brook University

Jeremias M. Gomes: jeremias@aluno.unb.br; George Teodoro: teodoro@cic.unb.br; Alba de Melo: albamm@cic.unb.br; Jun Kong: jun.kong@emory.edu; Tahsin Kurc: tahsin.kurc@stonybrook.edu; Joel H. Saltz: joel.saltz@stonybrookmedicine.edu

Abstract

We investigate the execution of the Irregular Wavefront Propagation Pattern (IWPP), a fundamental computing structure used in several image analysis operations, on the Intel® Xeon Phi™ co-processor. An efficient implementation of IWPP on the Xeon Phi is a challenging problem because of IWPP's irregularity and the use of atomic instructions in the original IWPP algorithm to resolve race conditions. On the Xeon Phi, the use of SIMD and vectorization instructions is critical to attain high performance. However, SIMD atomic instructions are not supported. Therefore, we propose a new IWPP algorithm that can take advantage of the supported SIMD instruction set. We also evaluate an alternate storage container (priority queue) to track active elements in the wavefront in an effort to improve the parallel algorithm efficiency. The new IWPP algorithm is evaluated with Morphological Reconstruction and Imfill operations as use cases. Our results show performance improvements of up to 5.63× on top of the original IWPP due to vectorization. Moreover, the new IWPP achieves speedups of 45.7× and 1.62×, respectively, as compared to efficient CPU and GPU implementations.

I. Introduction

This work investigates the efficient execution of a computation structure called Irregular Wavefront Propagation Pattern (IWPP) [1] on the Intel® Xeon Phi™. This research is motivated by the high computational requirements of whole slide tissue image processing in biomedical image analysis [2]. Analyses of whole slide tissue images (WSIs) can lead to a better understanding of disease mechanisms at the sub-cellular levels and improvements in patient diagnosis and care. A modern microscopy scanner is able to capture images at 100K×100K pixel resolutions and can scan many slides rapidly. This enables research groups and healthcare organizations to collect large volumes of image data. An analysis of a WSI using a single CPU core may take hours; as a consequence, WSIs are underutilized in research and clinical settings.

A typical WSI analysis pipeline consists of multiple computing stages such as normalization, object segmentation, feature computation, and image classification. The object segmentation stage is often the most costly stage. It is generally implemented as a pipeline of several finer-grain operations that include Morphological Reconstruction [3], Imfill (fill image regions and holes) [4], H-minima [4], H-maxima [4], Watershed [5], and distance transform [6]. Therefore, efficient implementations on multi-core and many-core systems of these operations are critical to reducing the overall execution time of an image analysis pipeline. The processing structures of these operations bear similarities and include the IWPP on a grid.

The IWPP is characterized by waves originating from one or more source grid points and by the irregular shape and expansion of wavefronts. 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 sources of wave propagations to neighbor elements. A wave propagation occurs when a given *propagation condition*, determined based on the value of a wavefront element and the values of its neighbors, is satisfied. Each element in the wavefront represents an independent wave propagation; interaction between waves may even change their direction. In the IWPP only elements in the wave front contribute to output results. Because of this property, an efficient implementation of the IWPP 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 (Section II).

The implementation of IWPP on the Intel Xeon Phi is a challenging problem. The parallel versions of the IWPP algorithm we developed in an earlier work [1], [7] requires the use of atomic updates to avoid races conditions during propagations between neighbor elements in a grid (Section II, Algorithm 1, line 10). However, the Intel Xeon Phi does not support atomic SIMD instructions, but their use is a key factor to attain high performance on this co-processor. In this paper we address this problem with a new execution strategy. The contributions of our work can be summarized as follows:

- We propose a novel IWPP strategy that allows for the use of SIMD instructions supported by the Intel Xeon Phi.
- We develop efficient implementations of the Morphological Reconstruction and Imfill operations using this strategy. These are the first vectorized versions of these operations for the Intel Xeon Phi. Both operations significantly improve upon the performance of their non-vectorized counterparts.
- We extend IWPP to evaluate the use of different containers to store elements in an active wavefront.

- We experimentally evaluate our propositions using two state-of-the-art Intel Xeon Phi co-processors and compare the performance of the new IWPP implementation to a fast GPU-based implementation. The new IWPP implementation attains speedups of up to 45.7 × and 1.62× as compared to the sequential CPU and GPU versions, respectively. These performance gains demonstrate the feasibility of using the Intel Xeon Phi to rapidly execute image analysis operations and be able to scale analysis pipelines to large image datasets.

The rest of this paper is organized as follows. Section II presents the use of the IWPP on image analysis, including the of use case operations: Morphological Reconstruction and Imfill. The novel IWPP parallelization strategy for the Intel Phi is detailed in Section III. The experimental evaluation is presented in Section IV. Sections V and VI, respectively, present the related work and conclude the paper.

II. Irregular Wavefront Propagation Pattern

The IWPP is characterized by one or multiple points in grid that act as sources for waves with irregular shape of the fronts. The waves are dynamically created during the execution as a result of the computation, which makes them data dependent. A set of elements forming the front of waves (active elements) act as sources of propagations to their neighbor elements. The propagation between elements is carried out when a *propagation condition*, computed using the value of a wavefront element and the values of its neighbors, is satisfied. Thus, each element of the propagation front represents an independent wave, which may merge with other waves or even change direction.

An important characteristic of this structure that allows for its fast execution is that only the computation of elements within the wavefront contribute for changes in the output results. As such, instead of sweeping over the entire data grid until the propagations converge, we are able to utilize an auxiliary container structure, e.g., queue, sets, to keep track of active elements (those in the front of the wave) and touch only parts of the grid that are relevant. The original IWPP [1] is presented in Algorithm 1.

Algorithm 1

Irregular Wavefront Propagation Pattern (IWPP)

```
Input: D: data elements in a multi-dimensional space
Output: D: stable set with all propagations reached
1: {Initialization Phase}
2: $S \leftarrow$ subset active elements from $D$
3: {Wavefront Propagation Phase}
4: while $S \neq \emptyset$ do
5:     Extract $e_i$ from $S$
6:     $Q \leftarrow N_G(e_i)$
7:     while $Q \neq \emptyset$ do
8:         Extract $e_j$ from $Q$
```

```
9:      if $PropagationCondition(D(e_i), D(e_j))$ = true then
10:         $D(e_j) \leftarrow max/min(D(e_i), D(e_j))$
11:         Insert $e_j$ into $S$
```

In this algorithm, a set of elements in a multi-dimensional grid space (D) is selected to form the initial wavefront (S). These active elements act as wave propagation sources in the wavefront propagation phase. During the 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 propagation 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.

We describe two morphological algorithms, Morphological Reconstruction and Imfill, from the image analysis domain to illustrate the IWPP. Input and 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 $\{0, \ldots, L - 1\}$ of gray levels from a discrete or continuous domain.

A. Morphological Reconstruction

Morphological reconstruction is one of the elementary operations in image segmentation [3]. 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 efficient algorithm for this operation [3] makes one pass using the raster and anti-raster orders. It then continues the computation using a First-In, First-Out (FIFO) queue. The queue is initialized with pixels satisfying 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 due the wavefront computation. The overall process continues until the queue is empty. A pseudo-code implementation is presented in Algorithm 2, where 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. As presented, it computes raster and anti-raster scans during the initialization phase, before it proceeds to the IWPP phase.

B. Imfill

Imfill is a technique used to fill holes or areas in binary or gray scale images [4]. In binary, it changes values of connected background elements to foreground values with a flood-fill operation that stops when object boundaries are reached. In gray scale images, using the same flood-fill scheme, it modifies the intensity value of dark areas surrounded by lighter areas to the same intensity level of the surrounding area. As such, it removes regional minima not connected to image borders.

Algorithm 2

Morphological Reconstruction Algorithm

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

Imfill is used in image processing to homogenizing images in order to avoid false segmentation, to remove lighting intensity differences or to facilitate other types of processing. The Figure 2 is an example of one of these applications where the objects or connected components have different shades inside and their content is homogenized with the use of Imfill. The Imfill algorithm is very similar to the of Morphological Reconstruction, except that in the initialization phase it inverts the input marker image to use it as a mask for the propagation.

III. Efficient parallel IWPP on an Intel Phi

This section presents our approach to efficiently execute the IWPP on the Intel Phi. We have redesigned the IWPP algorithm to create a new execution scheme (Section III-A) that does not require atomic instructions and, as such, can leverage SIMD instructions. Section III-B describes the SIMD version of this algorithm. Section III-C presents the version of the algorithm that exploits thread-level parallelism. Further, in Section III-D, we discuss the benefits of using a heap container to store wavefront elements in IWPP algorithms.

A. Redesigned IWPP algorithm

The new IWPP is presented in Algorithm 3. The main difference from the new to the original algorithm (Algorithm 1) refers to the propagation scheme. In the original algorithm, the active elements forming the wavefront propagate information to neighbors elements immediately after the *PropagationCondition* between pixels is evaluated true. This creates a race condition when multiple active elements try to update a third element in parallel. This problem is addressed with a two stage propagation strategy in which elements receiving propagation are first identified and, in a second stage, they become active in the computation to find the correct information that should be received from the neighborhood.

Algorithm 3

New IWPP algorithm - sequential non-vectorized

```
Input: D: data elements in a multi-dimensional space
Output: D: stable set with all propagations reached
 1: {Initialization Phase}
 2: …
 3: {Wavefront Propagation Phase}
 4: while $currWave.empty() = false$ do
 5:   {Identification of elements receiving propagation}
 6:   for all $p \in currWave$ do
 7:     for all $q \in N_G(p)$ do
 8:       if $PropagationCondition(D(q), D(p)) = true$ then
 9:         $nextWave.insert(q)$
10:   {Propagation}
11:   for all $p \in nextWave$ do
12:     for all $q \in N_G(p)$ do
13:       $D(p) \leftarrow max/min(D(q), D(p))$
14:   $currWave \leftarrow nextWave$
15:   $nextWave \leftarrow \emptyset$
```

The first stage of the propagation in the new IWPP is presented in lines 6 to 9 of Algorithm 3. It computes the propagation condition for each active element in the current wavefront set (*currWave*) and their respective neighbors. However, instead of directly performing the propagation, it inserts elements receiving propagation into another set *nextWave*. This container coincides with the elements actively propagating information in the next iteration of the algorithm, since they are receiving propagation in the current step. The second stage of the algorithm computes the actual information propagation (lines 11 to 13). In this stage, each element receiving a propagation searches its neighborhood for the element from which the information should be received, and performs the propagation from the correct neighbor.

B. Vectorized IWPP

The vectorized version of the redesigned IWPP is presented in Algorithm 4. The first for loop in the wavefront propagation phase (lines 9 to 15) refers to the identification of elements receiving propagation, whereas the second loop performs the propagations. These phases are detailed below.

1) Identification of elements receiving propagation—This phase is implemented using 3 operations that are vectorized in our code: element neighborhood reading (lines 10 to 12), computation of the propagation condition (line 13), and insertion of elements that receive propagation in *nextWave* (lines 14 and 15).

The *neighborhood reading* is designed based on the observation that neighbor elements are stored within constant shifts from an active element. Theses shifts are precomputed based on the size of the grid and are stored into a vector register (vec_{shift} — line 1 of Algorithm 4). The address of active elements are loaded into $/N_G(p)/$ consecutive positions of vec_p (line 10). The constant shifts are then added to vec_p to compute the addresses of the neighbor elements (line 11), which are read using a *gather* instruction in line 12. The neighborhood reading process is illustrated in Figure 3. To fully exploit the SIMD instructions, we may process multiple active elements p into a single iteration of the algorithm. For 4-connected and 8-connected neighborhoods typically used in our algorithms, respectively, 4 and 2 elements are processed per loop iteration in the Intel Phi using 32-bit integer data.

Algorithm 4

Vectorized IWPP algorithm

```
Input: D:data elements in a multi-dimensional space
Output: D:stable set with all propagations reached
1:  $vec_{shift} \leftarrow$ Constant address distance from neighborhood
2:  {Initialization Phase}
3:  …
4:  {Scan Phase}
5:  …
6:  {Wavefront Propagation Phase}
7:  while $currWave.empty() == false$ do
8:    {Identification of elements receiving propagation}
9:    for all $p \in currWave$ do
10:     $vec_p \leftarrow$ Extract active elements
11:     $vec_{addr} \leftarrow VecAdd(vec_p, vec_{shift})$
12:     $vec_{neigh} \leftarrow Gather(D, vec_{addr})$
13:     $mask_{cond} \leftarrow VectorPropagationCondition(vec_p, vec_{neigh})$
14:     $vec_{prefixSum} \leftarrow PrefixSum(mask_{cond})$
15:     $nextWave.Insert(vec_{addr}, mask_{cond}, vec_{prefixSum})$
16:   {Propagation}
17:   for all $q \in nextWave$ do
18:     $vec_q \leftarrow$Extract elements
19:     $vec_{addr} \leftarrow VecAdd(vec_q, vec_{shift})$
20:     $vec_{neigh} \leftarrow Gather(D, vec_{addr})$
21:     $D(q) \leftarrow Max/MinReduce(vec_q, vec_{neigh})$
```

```
22:     $currWave \leftarrow nextWave$
23:     $nextWave \leftarrow \emptyset$
```

The computation of the propagation condition is usually a comparison of values stored in the active element and its neighbors. This operation is performed for all pixels in the neighborhood, resulting into another vector register ($mask_{cond}$) that stores the value 1 in positions in which the propagation is evaluated true, and 0 otherwise.

The insertion of elements receiving propagations in the *nextWave* container is the next operation (lines 14 and 15), which is implemented in the vectorized form here for a regular FIFO queue. This operation computes the number of elements to be inserted and their relative position to the end of the *nextWave* queue (line 14). This is performed using an inclusive prefix-sum of the values in $mask_{cond}$, which stores 1 in the positions of neighbor elements that need to be stored. The result of this operation is a vector register $vec_{prefixSum}$ with the relative position to the end of the queue in which each element receiving propagation should stored. The prefix-sum consists of a series of elements shifting in the vector registers followed by addition operations. It is worth noticing that our prefix-sum is performed inside a vector register and, for this setting in which the width of the vector instruction is sufficient large to store all the data, this prefix-sum version is efficient. Finally, we perform the actual insertion of elements receiving propagation (line 15) using a *scatter* in which only elements with the vec_{cond} marked with 1 are inserted into the queue with the relative positions from the last element in the queue as calculated in the prefix-sum. The insertion is illustrated in Figure 4.

2) Propagation—This phase performs the actual propagation of information. The elements identified for receiving propagation search into their neighborhood for the element from which it should receive the information. Thus, elements receiving propagation are extracted from *nextWave* (line 18), their neighborhood addresses are calculated (line 19), and neighbor elements are read (line 20). Finally, we computed the actual element from which the propagation should be received in line 21. In the reduce operation, the neighbor elements are compared to the active elements, and the max/min resulting value is retrieved to update $D(q)$.

It is important to highlight that it may happen that the same element q is inserted multiple times into the *nextWave* container. As a consequence, it is also possible that q be processed multiple times into a single step of the propagation loop, which would lead to a data race in the update of $D(q)$. However, in this case, the computation of the q replicas would lead to the same result, because the value of the neighborhood for the copies of q is the same as they are read by the same instruction (gather in line 20). Thus, this causes a benign data race [8] and does not affect the application results.

C. Thread level parallelism in IWPP

The thread level parallelization strategy is employed on top of the vectorized IWPP presented in Algorithm 4. This algorithm is modified to have multiple threads that work independently in subsets of the active elements to perform propagations in parallel. In this scheme, the elements identified in the “Initialization Phase” as the seed set of active elements are stored into multiple container copies (one per thread), so that threads can process propagations starting in different seeds independently. The initialization of thread queues or containers is presented in Figure 5. The partitioning of seeds to multiple threads eliminates the need of communication and synchronization among threads, and it is only possible because IWPP operations are commutative. This reduction or elimination of synchronization is attained with the cost of a potential load imbalance among threads, which happens if propagations starting from seed elements of different threads do not have the same cost. This problem can be lessened with a good initial partitioning of seeds and we have not observed significant load imbalance in our experiments.

The “Wavefront Propagation Phase” is a slightly modified version of the vectorized IWPP (Algorithm 4) to execute in parallel. The modifications include the execution of the entire while loop in parallel, having each thread manipulating its own copy of the currWave and nextWave storage containers independently. Additionally, a barrier is inserted between identification of elements and propagation phases (after the first inner for loop), to guarantee that propagations from multiple steps do not overlap.

Propagations starting in seed elements from initial disjoint partitions (Figure 5) may cross partitions during their execution. As a consequence, it is possible that distinct threads happen to insert the same active element to their independent wavefront sets. In this case, a data race may exist in the update of element $D(q)$ by multiple threads. The consequences of this data race are exemplified in Figure 6 for the Morphological Reconstruction algorithm. Thread 1 has element e_{x-1} in its currWave container in *iteration i*, while thread 2 has e_{x+2} and e_{x+1}. The element in threads’ currWaves that satisfy the propagation condition to a common neighbor element is e_x. As such, e_x is inserted in the *nextWave* of both threads during the identification of elements receiving propagation. Further, if the neighborhood of e_x is not modified in the interval from which it is read by both threads, they will read the same neighborhood of e_x, select the same value to be stored in e_x, and write it twice to $D(q)$.

However, if the neighborhood of e_x is modified by propagations carried out by another element e_{x+2}, there is a chance of threads reading different neighborhood data versions and the update in $D(q)$ could result in different intermediate results (O.1 or O.2) in iteration i. If during the propagation phase thread 1 reads the neighborhood of e_x to registers, thread 2 processes propagations e_{x+1} and e_x, and after that thread 1 selects the value among those in its registers (value 7) to be store in e_x, the output will be O.1. In this case, however, the *nextWave i* of thread 2 would still contain e_{x+1} and e_x that are used in the next iteration (i+1) as *currWave i*+1 of the algorithm to identify elements receiving propagation. As such, the computation of e_{x+1} in iteration $i + 1$ would insert e_x to *nextWave* $i + 1$, and the execution of e_x in the propagation phase of iteration $i + 1$ would set e_x to 8 (Marker O.2).

The second execution order is the simplest one in which thread 2 processes propagations of e_{x+1} before thread 1 reads the neighborhood of e_x. In this case, the output O.2 would be reached in iteration i, as such iteration $i + 1$ would not change the results. As such, the same final results are reached independently of the execution order. Therefore, the data race observed in the thread-level parallel version is also benign, which is classified as a double check [8].

D. Improving the parallel efficiency with an alternate propagation order

In the parallel multi-threaded version of IWPP, threads are dispatched simultaneously to compute propagations that start in the partitions assigned to each one. As discussed, these propagations may cross partitions and threads could process intersecting areas in parallel during the execution. Despite the potential data races that are resolved in our approach with a 2-phase algorithm, the computation of the intersecting areas in parallel may lead to an increasing in the number of elements queued for computation during the execution. This occurs because the elements queued for computation in the parallel version are typically processed using a less updated version of the domain as compared to the sequential execution. As a consequence, an element may be inserted to the queue in the parallel version, but not in the sequential. The recomputation levels tend to increase with the level of parallelism exploited or number of computing threads used. Thus, the increasing in the throughput of the algorithm (elements processed per unit of time) due to parallelism may not reflect with the same intensity in execution times because of the intrinsic recomputation.

We approach this here problem by computing elements in the wavefront using an alternate order, which is less likely to result in the recomputation of an element due to multiple propagations. The main principle is to process elements with higher/smaller values first for max/min operations because they contain values that once propagated are unlikely to be overwritten by another propagation wave. Actually, in the sequential version, if a priority queue is used and we first process elements with higher values (for Morphological Reconstruction), every propagation is final, and no element in the domain will receive a propagation twice [9]. In order to employ this principle targeting to reduce the recomputation that is inherent of the parallel version, we have implemented the vectorized IWPP using an heap to maintain elements of the wavefront sorted according to their values. In this version (called heap), element propagations are still carried out using the vectorization, but container insertions are scalar as we employ a C++ Standard Template Library (STL) [10] to implement the priority queue.

IV. Experimental results

The IWPP was evaluated using Morphological Reconstruction and Imfill [11] algorithms. We used a machine with 2 Intel® Xeon™ 8-Core E5-processor, one coprocessor Intel® Xeon Phi™ SE10P, and 32GB of RAM, another compute node with the Intel® Xeon Phi™ 7120P, and K20 NVIDIA GPU for comparison purposes. The devices have been profiled for bandwidth performance, which is used to explain comparative performance of operations. The regular data access bandwidth used the STREAM benchmark [12], whereas random data access used a micro-benchmark we developed. It consists of 10 million data element accesses in parallel from random locations in a $4K \times 4K$ image. The results of the benchmark and other information about the processors are reported in Table I). Experiments were repeated at least 3 times and the coefficient of variation among experiments was not higher than 0.5%.

The input data used in our experimental evaluation have been collected in our brain tumor research studies [13]. The input images used vary both in terms of size (from $4K \times 4K$ to $16K \times 16K$ pixels) and tissue coverage, which refers to the approximate percentage of the image area that is covered with tissue. The CPU [3] and GPU [1], [14] versions used as baselines are to the best of our knowledge the most efficient implementations available.

A. Speedup

This section evaluates the speedups of the operations in a Intel Phi as the number of threads is varied. As presented in Figure 7, both operations attained good speedups and maximum performance was achieved when about 120 threads are used. The improvements over 60 threads are due to the hyperthreading. We reserved one core of the processor for the offload daemon in all experiments.

B. Effect of vectorization

This section evaluates the performance improvements attained with the vectorized IWPP proposed in this work as compared to the original non-vectorized version. The experimental evaluation was executed on the Intel Phi SE10P using the Morphological Reconstruction and Imfill with regular FIFO queue for 4K×4K input images with varying amount of tissue coverage.

The experimental results (Figure 8) show that the algorithms attained significant gains for all input data configurations and Imfill improvements with vectorization are slightly higher than those observed for Morphological Reconstruction. Although the performance gains for both algorithms is high (ranges from 4.29 to 5.63), the performances of the algorithms are still below more optimized performance that could result from the use of SIMD instructions. We used 32-bit integers to manipulate our data and, as such, 16 data elements may be processed per SIMD instruction. The main reason performance gains were smaller than 16 is the fact that IWPP is an irregular algorithm and the vectorized implementation of IWPP is not a direct translation of non-vector instructions to respective vector instructions. Thus, the number of instructions to perform the same operation in the vectorized code is higher than that in the non-vectorized code for some phases of the algorithm. For instance, the insertion of elements to a queue requires a prefix-sum that is performed in multiple instructions in the vectorized code, but it is only an increment in the non-vectorized code.

C. Impact of tissue coverage

In this section we evaluate the performance effects of tissue coverage (percentage of the image area that is covered with tissue). These experiments used 4K×4K tissue images with the SE10P. The speedups attained by the algorithms with regular FIFO queue as compared to the single CPU core versions are presented in Figure 9.

Both algorithms achieved significant performance improvements as compared to the sequential execution on a CPU. Higher tissue coverage resulted in better acceleration, because of increased computation cost that amortizes startup overheads and better exploits the co-processor. Additionally, Morphological Reconstruction achieved higher speedups than Imfill. The better gains of Morphological Reconstruction are due to the characteristics of the input used. Imfill computes an inverse of the input to use as a mask and, as such, it performs a larger number of propagations. This requires more passes on the data in the initialization phase. As such, performance gains in the vectorized wavefront propagation phase with respect to the entire execution of the algorithm are reduced.

D. Varying data size

This section evaluates the performance impact of increasing input data size. The experiments were carried out using the Intel Phi SE10P and Morphological Reconstruction with FIFO queue only, because its performance is similar to that of Imfill as data sizes are varied.

The experimental results, presented in Figure 10, show an increase in speedups on the co-processor as the input data size grows. The average gain is 1.45×, when the sizes of images increase from 4K×4K to 16K×16K pixels. The most significant performance gain occurs when the image size is increased from 4K×4K to 8K×8K pixels. The performance improvements with large images is a result of the higher amount of parallelism, which leads to better use of the coprocessor and amortizes the cost of launching computations and transferring data between the CPU and the co-processor.

E. Performance of different coprocessors and storage types

This section compares the performance of IWPP on the Intel Phi SE10P, 7120P, and the NVIDIA K20 GPU. The Intel Phis are equipped with the same number of computing cores (61), but differ in the clock rate: 1.1GHz and 1.33GHz, respectively, for the SE10P and 7120P. The GPU implementation of IWPP was developed in an early work [1]. We also evaluated the impact of using a heap/priority queue to store wavefront elements on the Intel Phi. In the heap version wavefront elements are maintained in sorted order according to their values (i.e., their intensity levels).

The comparative performance of the coprocessors and storage types is presented in Figure 11, using the single core CPU version as a reference. As shown, all versions attained significant improvements on the coprocessors. The multicore CPU version attained speedups of up to 7.12× as compared to the single core version. The execution using the Intel Phi 7120P resulted in a speedup of 1.14× on top of the SE10P coprocessor, which is explained by their clock rate difference. The GPU version slightly improved on top of the best Intel Phi using regular FIFO queues, attaining speedups of up to 28.29 on top of the CPU sequential version. This may be explained by the different bandwidth attained by the processors. These operations are data intensive and they have regular data accesses in the initialization phase and irregular or random in the IWPP phase. While the Intel Phi devices are more efficient than the GPU for the regular data accesses, the GPU attains a bandwidth about 2.04× higher than the fastest Phi for irregular data accesses. Finally, the use of a heap to store wavefront elements (ordered by their intensity levels) resulted in significant performance improvement. The execution on 7120P with the heap and input images of 16K×16K pixels achieved a speedup of 1.62× on top of the GPU-based version. The strong gains with the use of a heap are a result of processing about 20× fewer elements during the wavefront propagation phase as compared to when a queue is used. In the queue, as we increase the number of threads used for the same input, more elements are processed in the wavefront propagation phase because of propagations that are computed multiple times due to propagations carried out in parallel. In the heap version, however, the sorting of elements by their values asserts that propagations carried by a thread are more likely to be final and, as a consequence, increases parallel efficiency. In an earlier effort, we implemented the IWPP using a state-of-the-art priority queue proposed by He et al. [15], but it was not able to improve the regular queue based GPU implementation because of the data management costs.

V. Related Work

The IWPP may be modeled as a graph scan algorithm with multiple sources. As such, recent efforts on efficient implementations of Breadth-First Search (BFS) [16] [17] are interesting for the sake of comparison with IWPP execution schemes and optimizations. The work of Hong et al. [16], for instance, provides techniques and optimizations to deal with load imbalance from irregular number of edges in vertices from real-world graphs for their GPU-based BFS algorithms. Although these techniques have shown to be effective to their work, it would have no impact in IWPP that has a regular and constant number of edges per vertex, represented by the fixed neighborhood. Tao et al. [17] is a closer related work that describes approaches to accelerate BFS using the Intel Phi. It develops reading and expansion operations using SIMD instructions, but it still uses atomic (non-vectorized) instructions to perform expansion of vertices. Also, Tao et al. use a bitmap to mark elements that should be processed in the next step of the algorithm and, as a consequence, it requires passing through all elements of the domain (represented by the bitmap) in each iteration to check for active elements. It makes their algorithm inefficient for IWPP, which touches only parts of the domain that contribute to the output in each step. Still, it served as an inspiration for our propositions.

The Morphological Reconstruction and Imfill algorithms have been used by several applications. Vincent [18] formally defined these operations and showed that their efficient execution was reached using a First-In, First-Out (FIFO) queue. A parallel CPU cluster-based version of Morphological Reconstruction was proposed by Laurent and Roman[19]. Karas [20] and Jivet et. al [21], respectively, focused on Field-Programmable Gate Arrays (FPGAs) and GPU based implementations of the Morphological Reconstruction. However, in both of the cases, an inefficient version of the algorithm that does not use queues to store the wavefront is employed in the parallelization. As such, the performance improvements as compared to the efficient queue based version are reduced. This work focuses on the implementation of the optimized version of the algorithm.

VI. Conclusion

This paper has proposed and implemented an efficient vectorized algorithm for the Irregular Wavefront Propagation pattern on the Intel® Xeon Phi™. The thread level parallel version of the algorithm and the use of a priority queue container have also been introduced to further improve performance. The experimental results using two types of operations (Morphological Reconstruction and Imfill), which are common in image analysis pipelines, show that the new IWPP algorithm improves the original non-vectorized IWPP up to 5.63×. Both operations attain significant performance gains on top of a state-of-the-art CPU implementations. The Morphological Reconstruction with vectorization can get speedups of up to 45.7× and 1.62× on top of the 1-CPU core [3] and GPU-based [1] versions. As a future work, we intend to extend the IWPP to execute on multiple Intel Phi co-processors.

Acknowledgments

This work was supported in part by 1U24CA180924-01A1 from the NCI, R01LM011119-01 and R01LM009239 from the NLM, CNPq, Capes, and NIH K25CA181503. This research used resources provided by the XSEDE Science Gateways program under grant TG-ASC130023.

References

1. Teodoro G, Pan T, Kurc T, Kong J, Cooper L, Saltz J. Efficient Irregular Wavefront Propagation Algorithms on Hybrid CPU-GPU Machines. Parallel Computing. 2013
2. Kong J, Cooper LA, Wang F, Gao J, Teodoro G, Scarpace L, Mikkelsen T, Schniederjan MJ, Moreno CS, Saltz JH, et al. Machine-Based Morphologic Analysis of Glioblastoma Using Whole-Slide Pathology Images Uncovers Clinically Relevant Molecular Correlates. PloS one. 2013; 8(11):e81049. [PubMed: 24236209]
3. Vincent L. Morphological grayscale reconstruction in image analysis: Applications and efficient algorithms. IEEE Transactions on Image Processing. 1993; 2:176–201. [PubMed: 18296207]
4. Soille, P. Morphological Image Analysis: Principles and Applications. 2. Secaucus, NJ, USA: Springer-Verlag New York, Inc; 2003.
5. 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)
6. Vincent L. Exact euclidean distance function by chain propagations. IEEE Int Comp Vision and Pattern Recog Conf. 1991:520–525.
7. Teodoro, G.; Kurc, T.; Kong, J.; Cooper, L.; Saltz, J. Comparative Performance Analysis of Intel (R) Xeon Phi (TM), GPU, and CPU: A Case Study from Microscopy Image Analysis. IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS); 2014;
8. Narayanasamy, S.; Wang, Z.; Tigani, J.; Edwards, A.; Calder, B. ACM SIGPLAN Notices. Vol. 42. ACM; 2007. Automatically classifying benign and harmful data races using replay analysis; p. 22-31.
9. Robinson K, Whelan PF. Efficient morphological reconstruction: a downhill filter. Pattern Recogn Lett. 2004; 25:1759–1767.
10. Musser, DR.; Derge, GJ.; Saini, A. STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library. Addison-Wesley Longman Publishing Co., Inc; 2001.
11. Gonzalez RC, Woods RE, Eddins SL. Morphological reconstruction. Digital Image Proc using MATLAB, MathWorks. 2010
12. McCalpin JD. Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture Newsletter. Dec.1995 : 19–25.
13. 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;
14. Teodoro G, Pan T, Kurc TM, Cooper L, Kong J, Saltz JH. A fast parallel implementation of queue-based morphological reconstruction using GPUs. Emory University, Center for Comprehensive Informatics Technical Report CCI-TR-2012-2. 2012
15. He, X.; Agarwal, D.; Prasad, SK. Design and implementation of a parallel priority queue on many-core architectures. 19th International Conference on High Performance Computing, HiPC 2012; Pune, India. December 18–22, 2012; 2012. p. 1-10.
16. Hong S, Kim SK, Oguntebi T, Olukotun K. Accelerating CUDA graph algorithms at maximum warp. ACM SIGPLAN Notices. 2011

17. Tao, G.; Yutong, L.; Guang, S. Using MIC to accelerate a typical data-intensive application: the breadth-first search. Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International; IEEE; 2013. p. 1117-1125.

18. Vincent L. Morphological algorithms. OPTICAL ENGINEERING-NEW YORK-MARCEL DEKKER INCORPORATED-. 1992; 34

19. Laurent, C.; Roman, J. Parallel implementation of morphological connected operators based on irregular data structures. Third International Conference on Vector and Parallel Processing, ser. VECPAR '98; London, UK, UK: Springer-Verlag; 1999. p. 579-592.

20. Karas, P. MEMICS, ser. OASICS. Vol. 16. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik; Germany: 2010. Efficient Computation of Morphological Greyscale Reconstruction; p. 54-61.

21. Jivet I, Brindusescu A, Bogdanov I. Image contrast enhancement using morphological decomposition by reconstruction. WSEAS Trans Cir and Sys. Aug.2008 7:822–831.

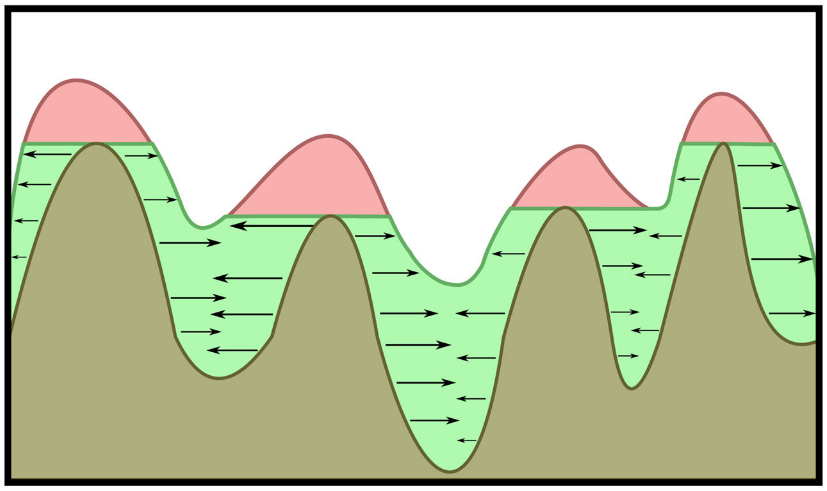

Fig. 1.

Gray scale morphological reconstruction in 1-dimension. The marker image intensity profile is represented as the brown line, and the mask image intensity profile is represented as the red 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.

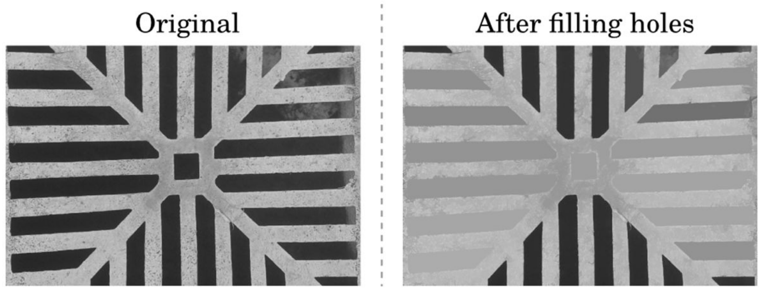

Fig. 2.
Imfill in a gray scale image. Connected components (those with surrounding border) are filled and noise is reduced.

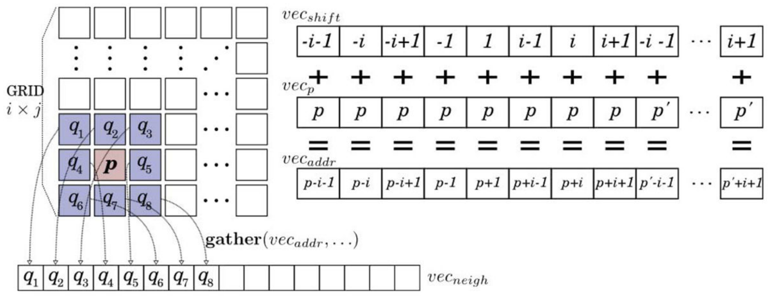

Fig. 3.

Loading an 8-connected neighborhood for two elements *p* and *p*′. *vec*$_{shift}$ contains shifts to neighbors of each element. Addresses of *p* and *p*′ are replicated in 8 consecutive positions of *vec*$_p$, and *vec*$_{shift}$ is added to *vec*$_p$ to calculate addresses of *p* and *p*′ neighbors. A gather with *vec*$_{addr}$ is used to read the neighbors (only *p*'s neighborhood is shown in *vec*$_{neigh}$).

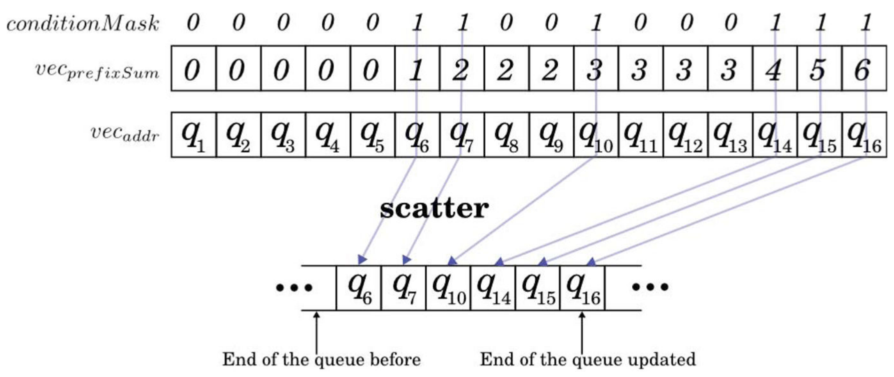
$conditionMask$ 0 0 0 0 0 1 1 0 0 1 0 0 0 1 1 1
$vec_{prefixSum}$ 0 0 0 0 0 1 2 2 2 3 3 3 3 4 5 6
vec_{addr} q_1 q_2 q_3 q_4 q_5 q_6 q_7 q_8 q_9 q_{10} q_{11} q_{12} q_{13} q_{14} q_{15} q_{16}
scatter
••• q_6 q_7 q_{10} q_{14} q_{15} q_{16} •••
End of the queue before
End of the queue updated

Fig. 4.

The $mask_{cond}$ marks the elements in vec_{addr} to be inserted in queue and $vec_{prefixSum}$ contains their position relative to the queue end.

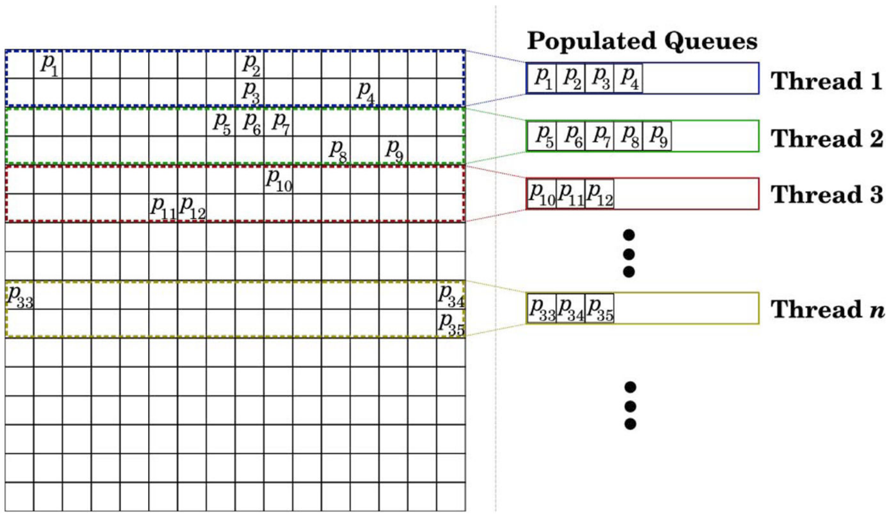

Fig. 5.

Input grid divided into subregions and seed elements from partitions are assigned to multiple thread.

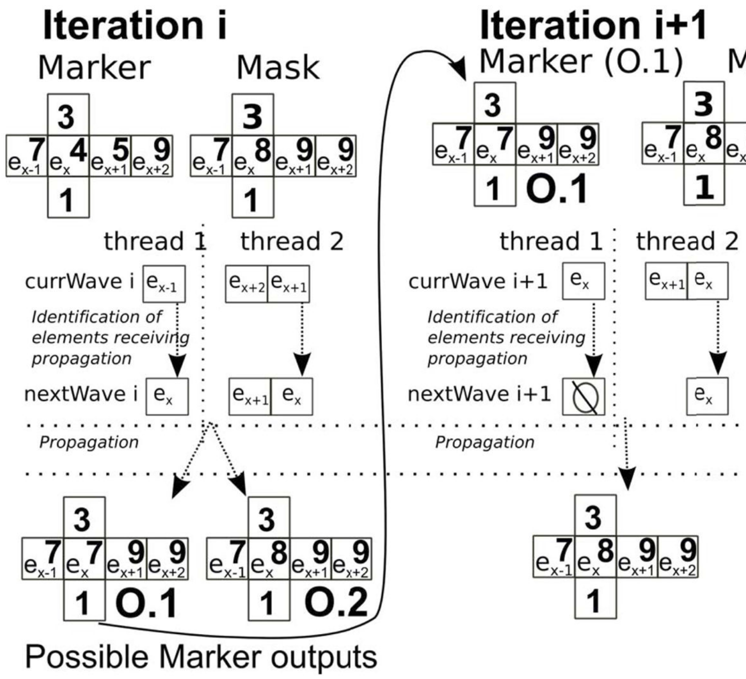

Fig. 6.
Data race exemplified using Morphological Reconstruction algorithm.

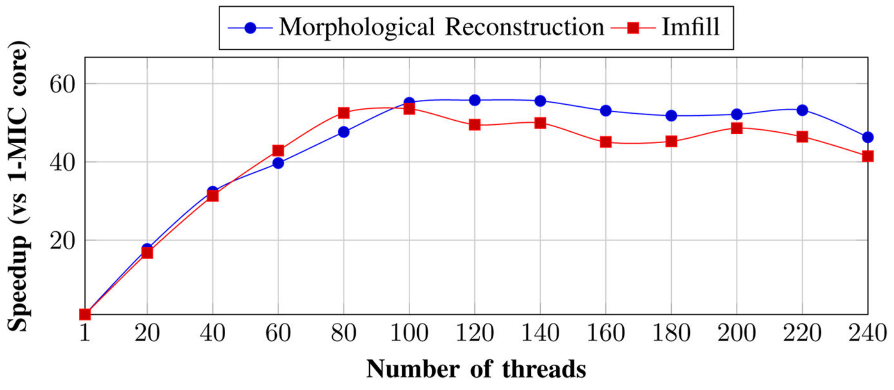

Fig. 7.

Scalability analysis for both algorithms varying the number of threads.

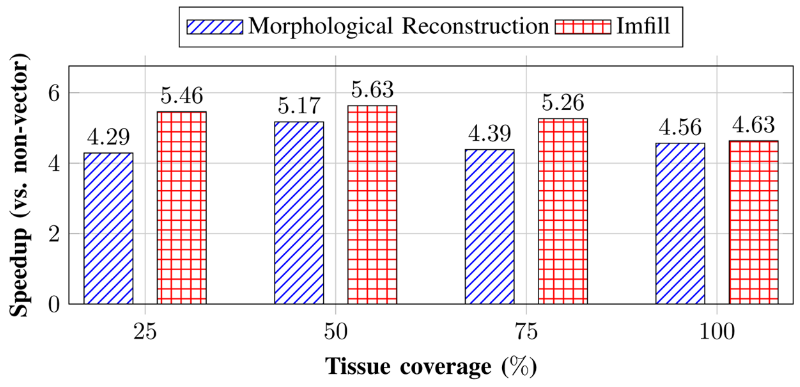

Fig. 8.
Performance improvement of vectorization for both algorithms using images with varying amount of tissue coverage.

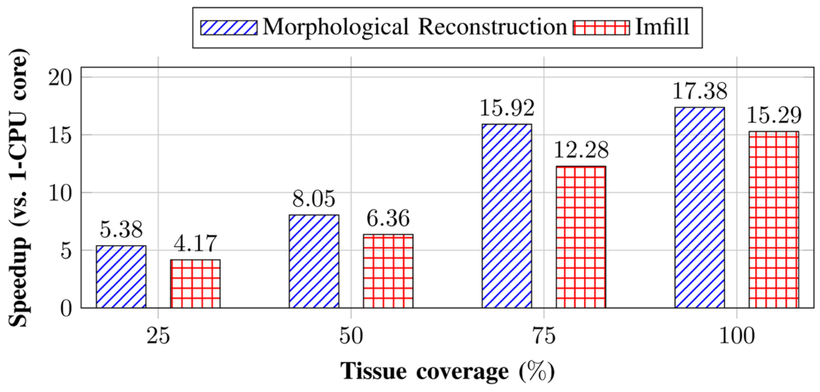

Fig. 9.

Acceleration attained in the SE10P Intel Phi as compared to 1-CPU core for input data with varying tissue coverage.

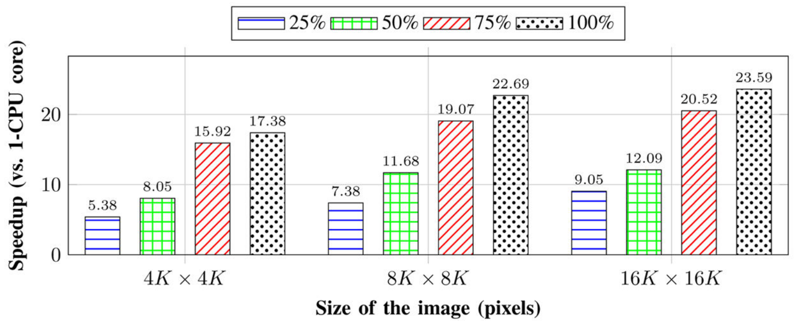

Fig. 10.

Performance improvements of the Morphological Reconstruction on Phi as the input data size is varied.

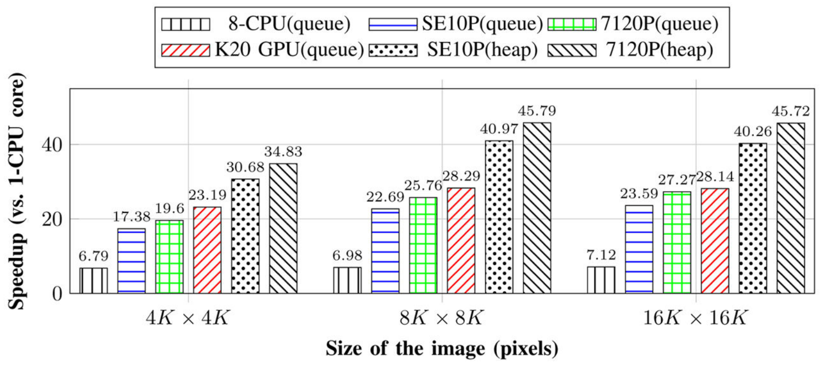

Fig. 11.

Evaluation of multiple coprocessors and wavefront storage containers with Morphological Reconstruction for 100% coverage input data.

TABLE I

PROCESSORS CHARACTERISTICS.

	K20 GPU	**SE10P**	**7120P**
Number of cores	2496	61	61
Processor core clock (MHz)	706	1100	1238
Bandwidth - Regular access (GB/s)	148	160	177
Bandwidth - Random access (MB/s)	895	399	438