Simple and Fast Parallel Algorithms for the Voronoi Maps and the Euclidean Distance Map, with GPU implementations

Takumi Honda, Shinnosuke Yamamoto, Hiroaki Honda, Koji Nakano, Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—The complete Voronoi map of a binary image with black and white pixels is a matrix of the same size such that each element is the closest black pixel of the corresponding pixel. The complete Voronoi map visualizes the influence region of each black pixel. However, each region may not be connected due to exclave pixels. The connected Voronoi map is a modification of the complete Voronoi map so that all regions are connected. The Euclidean distance map of a binary image is a matrix, in which each element is the distance to the closest black pixel. It has many applications of image processing such as dilation, erosion, deblurring effects, skeletonizing and matching. The main contribution of this paper is to present simple and fast parallel algorithms for computing the complete/connected Voronoi maps and the Euclidean distance map and implement them in the GPU. Our parallel algorithm first computes the mixed Voronoi map, which is a mixture of the complete and connected Voronoi maps, and then converts it into the complete/connected Voronoi by exposing/hiding all exclave pixels. After that, the complete Voronoi map is converted into the Euclidean distance map by computing the distance to the closest black pixel for every pixel in an obvious way. The experimental results on GeForce GTX 1080 GPU show that the computing time for these conversions is relatively small. The throughput of our GPU implementation for computing the Euclidean distance maps of $2K \times 2K$ binary images is up to 2.08 times larger than the previously published best GPU implementation, and up to 172 times larger than CPU implementation using Intel Core i7-4790.**

I. Introduction

The GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [1]–[3]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [1]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [4], [5], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [6], since they have thousands of processor cores and very high memory bandwidth. However, memory access to the global memory by threads must be *coalesced* to fully utilize high memory bandwidth.

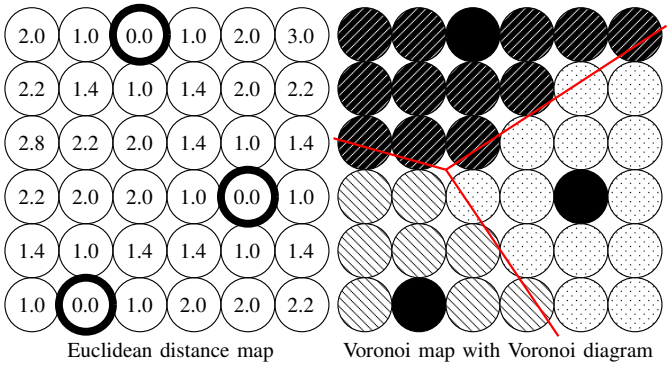

Fig. 1. The Euclidean distance map and the Voronoi map with the Voronoi diagram for a 6×6 binary image with three black pixels

Suppose that a binary image of $\sqrt{n} \times \sqrt{n}$ pixels, each of which takes 0 (white) or 1 (black), is given. *The Euclidean distance map* is a matrix of the same size such that each element is the distance to the closest black pixel. It has many applications in the area of image processing such as dilation, erosion, deblurring effects, skeletonizing and matching. Figure 1 shows the Euclidean distance map of a 6×6 binary image with three black pixels.

The Voronoi diagram of a set P of points in a plain is a partitioning of a plain into regions called *Voronoi cells*, each of which consists of all points closest to a point p ($\in P$) over all points in P. *The Voronoi map* (or *the complete Voronoi map*) of a binary image is a projection of the Voronoi diagram, that is, a matrix of the same size, in which each element is the closest black pixel. Hence, the complete Voronoi map visualizes regions of influence and adjacency relationship of black pixels. However, the complete Voronoi map may have an exclave pixel as illustrated in Figure 2. Although the closest pixel of the exclave pixel in the figure is black pixel b, it is not adjacent to the other pixels in the Voronoi cell of black pixel b. A cluster of multiple exclave pixels may appear if the size of an image is large. Since exclave pixels seem to be noise, we should hide all exclave pixels if such noise is unfavorable. For this purpose, elements in the complete Voronoi map corresponding to exclave pixels are

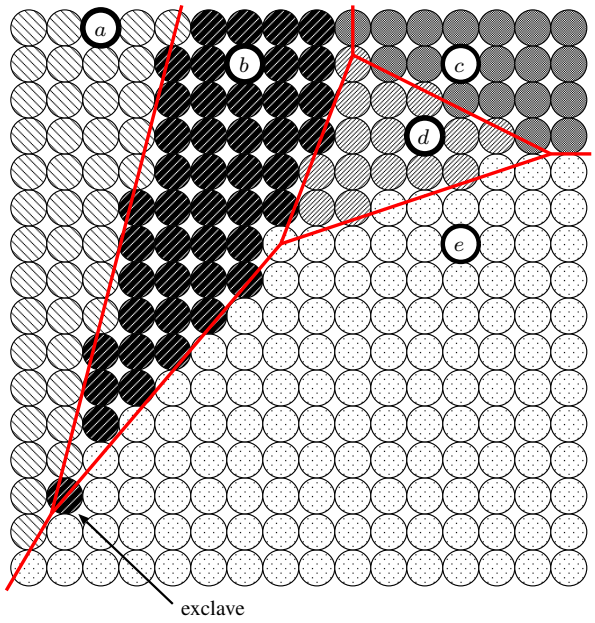

Fig. 2. The Voronoi diagram and the complete Voronoi map of a 16×16 binary image

replaced by those of neighboring pixels so that all pixels in the same Voronoi cell are connected. We call such Voronoi map *the connected Voronoi map*. For example, the element of the exclave pixel in Figure 2 is change to black pixel a or e.

The main contribution of this paper is to present simple and fast parallel algorithms for computing the complete/connected Voronoi maps and the Euclidean distance map, and implement it in the GPU. Our parallel algorithm first computes the mixed Voronoi map, which are a mixture of the complete and connected Voronoi maps by a simple operation. After that, it is converted it into the complete/connected Voronoi maps by exposing/hiding all exclave pixels. In practice, the number of exclave pixels is very small. Hence, the computation cost of these conversions is very small. Once the complete Voronoi map is obtained, the Euclidean distance map can be computed in an obvious way.

Many algorithms for computing the Euclidean distance map have been proposed in the past, such as sequential algorithms [7]–[10] and parallel algorithms [11]–[15]. Breu *et al.* [7] and Chen *et al.* [8], [9] have presented $O(n)$-time sequential algorithm for computing the Euclidean distance map. Since all pixels must be read at least once, these sequential algorithms with time complexity of $O(n)$ is optimal. Roughly at the same time, Hirata [10] presented a simpler $O(n)$-time sequential algorithm to compute the distance map for various distance metrics including Euclidean, four-neighbor, eight-neighbor, chamfer, and octagonal. On the other hand, for accelerating sequential ones, numerous parallel algorithms have been developed for various parallel model. Lee *et al.* [16] presented an $O(\log^2 n)$-time algorithm using n processors on the EREW PRAM. Pavel and Akl [13] presented an algorithm running in $O(\log n)$ time and using n^2 processors on the EREW PRAM. Clearly, these two algorithms are not work-optimal. Fujiwara *et al.* [11] have presented a work-optimal algorithm running in $O(\log n)$ time using $\frac{n}{\log n}$ processors on the EREW PRAM and in $O(\frac{\log n}{\log \log n})$ time using $\frac{n \log \log n}{\log n}$ processors on the CRCW PRAM. Later, Hayashi *et al.* [12] have exhibited a more efficient algorithm running in $O(\log n)$ time using $\frac{n}{\log n}$ processors on the EREW PRAM and in $O(\log \log n)$ time using $\frac{n}{\log \log n}$ processors on the PRAM. Since the product of the computing time and the number of processors is $O(n)$, these algorithms are work optimal. Also, it was proved that the computing time cannot be improved as long as work optimality is satisfied, these algorithms are also work optimal. Our parallel algorithm takes a different new approach to compute the Euclidean distance map.

Rong *et al.* [17] has been presented *the jumping flooding algorithm (JFA)*, that computes the Voronoi map and the Euclidean distance map on the GPU. The jumping flooding algorithm diffuses the closest black pixel in $\log \sqrt{n}$ steps such that each step i $(1 \leq i \leq \log \sqrt{n})$ diffuses the closest pixel to 8 pixels in relative positions $(\{\pm\frac{\sqrt{n}}{2^i}, 0\}, \{\pm\frac{\sqrt{n}}{2^i}, 0\})$. Since memory access is regular and coalesced, the jump flooding algorithm attains very high memory bandwidth. However, it totally performs $O(n \log n)$ memory access operations, it runs fast only if the image size is small, say at most 512×512 pixels. Also, the resulting Voronoi map and the Euclidean distance map have unexpected errors and they are just approximated results. Schneider *et al.* [18] has presented a sweep line algorithm for computing the Euclidean distance map and implemented in a GPU. Since it performs $O(n)$ memory access operations, it is optimal. However, the resulting Euclidean distance map still has errors. Man *et al.* [19], [20] have presented a GPU implementation of the work-time optimal algorithm for computing the Euclidean distance map shown in [12]. In their implementation, each of $\sqrt{n}$ threads is assigned to a column and it computes the column-wise closest black pixel within the same column. After that, a thread is assigned a row and the Euclidean distance map in the assigned row is computed using the column-wise closest black pixels computed in the previous step. This implementation has several drawbacks. First, for coalesced memory access in the row-wise computation, matrix transposing must be performed before and after the row-wise computation. Also, a large stack is necessary for the row-wise Euclidean distance map computation by a thread. More specifically, $\sqrt{n}$ stacks must be used and memory access to the stacks by $\sqrt{n}$ threads is not coalesced. Cao *et al.* [21] has presented a more sophisticated GPU implementation called *Parallel Banding Algorithm (PBA)*, which is also based on the work-time optimal algorithm in [12]. Basically, the PBA partitions computation into bands. The band-wise computation is performed, and the results are combined to use more threads. Thus, the PBA has larger parallelism and runs faster than a simple GPU implementation [19], [20]. However, it still performs complicated stack operations and non-coalesced access to the global memory.

We have implemented our parallel algorithm for computing

the Voronoi maps and the Euclidean distance map and evaluated the performance on GeForce GTX 1080 GPU. For a single binary image of size $16K \times 16K$, our implementation is 1.12-1.54 times faster than the PBA on the same GPU and 118-132 times faster than a sequential algorithm running on Intel Core i7-4790 CPU. For computing the Euclidean distance map, the throughput of our GPU implementation is 1.43- 2.08 times larger than the PBA, and 121-172 times faster than CPU implementation. Thus, our implementation achieves better performance than the previously published best GPU implementation, although it is much simpler. Further, *the warp-wise asynchronous sweep*, which we have developed has few synchronization overhead and contributes high acceleration of the computation.

This paper is organized as follows. In Section II, we define the Voronoi maps and the Euclidean distance map, and discuss condition to have exclave pixels. We then go on to show parallel algorithms for computing the Voronoi maps and the Euclidean distance map in Section III. Section IV presents our GPU implementations for computing the Voronoi maps and the Euclidean distance map. Finally, we show experimental results using the GPU in Section V. Section VI concludes our work.

II. Voronoi maps and Euclidean distance map

Let $\mathrm{dist}(p, q)$ denote the Euclidean distance of two points p and q in a plane. For a set P of points in a plain, *the Voronoi diagram* V is a partitioning of a plain into regions $V(p)$ ($p \in P$) called *Voronoi cells*, each of which consists of all points closest to p over all points in P, that is,

$$V(p) = \{q \in Q \mid \mathrm{dist}(p, q) \leq \mathrm{dist}(q', q) \text{ for all } q' \in P\},$$

where Q is a set of all points in a plain. The boundary of Voronoi cell $V(p)$ for each point p is determined line segments, each of which is a perpendicular bisector of p and a neighbor point in P. We call a line segment representing the boundary of a Voronoi cell *a Voronoi edge* and a point connecting two or more Voronoi edges *a Voronoi vertex*.

The Voronoi map (or *the complete Voronoi map*) is a projection of the Voronoi diagram in a binary image. Let B be a binary image of size $\sqrt{n} \times \sqrt{n}$ such that the value $B(i, j)$ of each pixel (i, j) ($0 \leq i, j \leq \sqrt{n} - 1$) takes 0 (white) or 1 (black). *The complete Voronoi map* C of a binary image B is determined by the Voronoi diagram V of a set of black points, $P = \{(i, j) \mid (i, j) \text{ is black}\}$. More specifically, each pixel (i, j) is assigned *a label* $C(i, j)$, which is the closest black pixel of pixel (i, j). Thus, $(i, j) \in V(C(i, j))$ is satisfied for all pixels (i, j) ($0 \leq i, j \leq \sqrt{n}-1$). We assume that, if a pixel has two or more equidistant closest black pixels, the earliest black pixel among them in row major order is the closest. Figure 2 shows the complete Voronoi map C of a 16×16 binary image, which has five black pixels a, b, c, d, and e and the other pixels are white. Pixels are partitioned into five regions, each of which corresponds to the Voronoi cell of a black pixel. It also illustrates the Voronoi diagram V of the five points. *The Euclidean distance map* E of a binary image B is a matrix E such that each $E(i, j)$ is the distance to the closest black pixel of (i, j). Since $C(i, j)$ is the closest black pixel of (i, j), we have $E(i, j) = \mathrm{dist}((i, j), C(i, j))$ and E can be computed from the complete Voronoi map C directly.

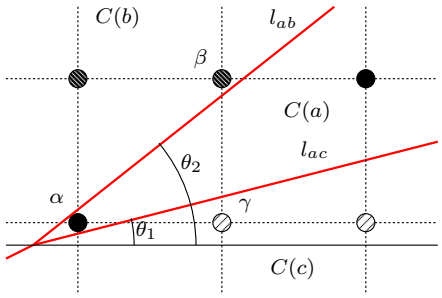

Fig. 3. Necessary condition for having exclave pixels

As we can see in Figure 2, the complete Voronoi map may have *an exclave pixel*, which is not included in the connected region of a Voronoi cell. More formally, an exclave pixel is defined as follows. We assume that two pixels (i, j) and (i', j') are *neighbors* if $|i - i'| \leq 1$ and $|j - j'| \leq 1$. Thus, each pixel has at most eight neighbors. *A path* of pixels is a sequence of pixels $p_0, p_1, \ldots, p_k$ such that p_i and p_{i+1} ($0 \leq i \leq k-1$) are neighbors. Let C be the complete Voronoi map of an image B. A pixel p is *a connected pixel* if there exists a path of pixels $p_0 (= p), p_1, p_2, \ldots, p_k$ such that p_k is a black pixel and $C(p_i) = p_k$ for all i ($0 \leq i \leq k$). A pixel p is *an exclave pixel* if it is not a connected pixel.

Let us see a condition to have exclave pixels. Exclave pixels may appear in the complete Voronoi map C around Voronoi points. Figure 3 illustrates an example of such three Voronoi cells $C(a)$, $C(b)$, and $C(c)$. In this figure, two Voronoi edges l_{ab} and l_{ac} lie between pixels β ($\in C(b)$) and γ ($\in C(c)$) and so α ($\in C(a)$) is an exclave pixel. Clearly, both $\theta_1 > 0$ and $\theta_2 < \frac{\pi}{4}$ are satisfied. In general, for some integer i ($0 \leq i \leq 7$), both $\theta_1 > \frac{\pi}{4} i$ and $\theta_2 < \frac{\pi}{4}(i + 1)$ must be satisfied to have an exclave pixel. Note that, even if this condition is satisfied, exclave pixels may not exist. These conditions are necessary to have exclave pixels. Also, it is possible to have two or more exclave pixels if the angle $\theta_2 - \theta_1$ is very small.

In some applications, *a connected Voronoi map* F, which hides all exclave pixels, are expected. *The connected Voronoi map* F of a binary image B is obtained by modifying the complete Voronoi map C such that, every exclave pixel is replaced by a non-exclave neighbor pixel. In other words, $F(i, j) = C(i, j)$ if pixel (i, j) is not an exclave pixel. Otherwise, $F(i, j) = C(i', j')$ such that (i', j') is a non-exclave neighbor pixel of (i, j). For example, an exclave pixel in Figure 2 is labeled as a or e. *The mixed Voronoi map* M is a mixture of the complete Voronoi map C and the connected Voronoi map F. It is a map satisfying $M(i, j) = C(i, j)$ or $F(i, j)$ for all pixels (i, j).

III. Parallel algorithms for Voronoi maps and Euclidean distance map

We first show a very simple parallel algorithms for computing the mixed Voronoi map of a binary image. We will show that, the mixed Voronoi map can be converted into the connected Voronoi map with small computational cost.

A. Simple parallel algorithm for the mixed Voronoi map

Our algorithm basically performs sweep operations presented in [18]. The parallel algorithm performs four sweep operations, *down sweep*, *up sweep*, *left sweep*, and *right sweep*. These operations are almost the same and the difference is the direction of sweep. Let closest$(p, q_1, q_2, \ldots)$ be a function for pixels $p, q_1, q_2, \ldots$ such that it finds and returns the closest pixel of p in the remaining pixels. In other words, if it returns q_i, then $\mathrm{dist}(p, q_i) \leq \mathrm{dist}(p, q_j)$ is satisfied for every $j \geq 1$. The down sweep computes the angle restricted closest black pixel of every pixel with angle upward 90 degree. For this computation, angle restricted close black pixels are propagated downward 90 degree. A parallel algorithm for the down sweep is described below. It stores the angle restricted closest black pixel of (i, j) in $M_d(i, j)$ when it terminates. We use M_d's outside of index range such as $M_d(-1, 0)$ to avoid special handling for boundary pixels. We assume that all $M_d(i, j)$'s including them store pixel $(+\infty, +\infty)$, initially.

[Down sweep]
for $i \leftarrow 0$ to $\sqrt{n} - 1$ do
  for $j \leftarrow 0$ to $\sqrt{n} - 1$ do in parallel
    if $B(i, j) = 1$ then $M_d(i, j) \leftarrow (i, j)$
    else $M_d(i, j) \leftarrow \mathrm{closest}((i, j), M_d(i - 1, j - 1),$
      $M_d(i - 1, j), M_d(i - 1, j + 1))$

Figure 4 shows the pixels stored in $M_d(i, j)$ after the down sweep terminates. We can see that each black pixel is propagated downward 90 degree.

Let M_u, M_l and M_r be arrays obtained by the up sweep, the left sweep and the right sweep, respectively. The mixed Voronoi map M can be obtained by pixel-wise minimum of the four sweep operations. More specifically, by performing $M(i, j) \leftarrow \mathrm{closest}((i, j), M_d(i, j), M_u(i, j), M_l(i, j), M_r(i, j))$ for all pixels (i, j), we can obtain the mixed Voronoi map M. We call this algorithm *the 4-directional sweep*.

In the 4-directional sweep, $M_d(i, j)$, $M_u(i, j)$, $M_l(i, j)$, and $M_r(i, j)$ store the angle restricted closest black pixels for four directions of 90 degree, respectively, if (i, j) is not an exclave pixel. Thus, the 4-directional sweep computes the closest black pixel of every non-exclave pixel correctly. We will show that exclave pixels may or may not be hidden in the resulting Voronoi map by the 4-directional sweep. In other words, $M(i, j)$ for an exclave pixel (i, j) stores the closest pixel of (i, j) or $M(i', j')$ of a non-exclave neighbor pixel (i', j') of (i, j). Suppose that the 4-directional sweep is executed for the binary image in Figure 2. Figure 4 shows the resulting black pixels stored in M_d computed by the down sweep. The value of M_d for the exclave pixel is black pixel b.

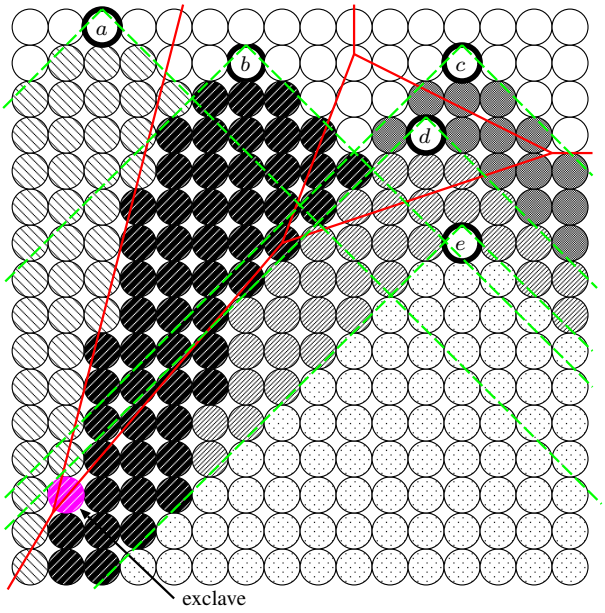

Fig. 4. Pixels stored in $M_d(i, j)$ by the down sweep for the same binary image of Figure 2

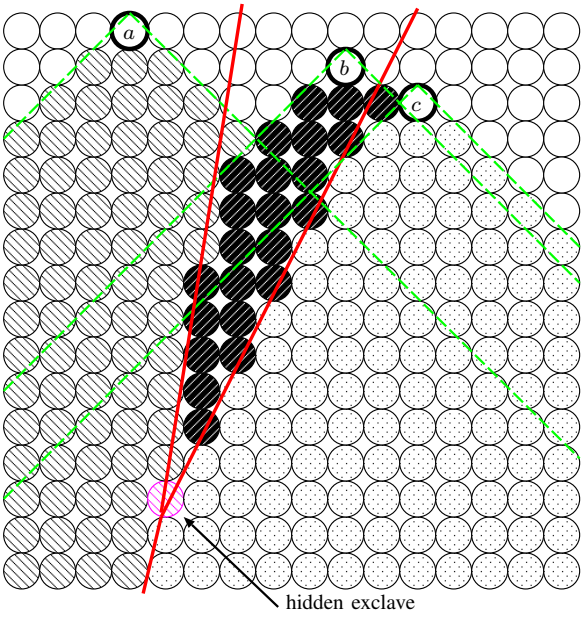

Fig. 5. The resulting Voronoi map by the down sweep: an exclave pixel disappears

Since black pixel b is the closest pixel over all black pixels, the value of M for the exclave pixel is also b. Thus, an exclave pixel is not hidden in M.

Next, we will show an example, in which an exclave pixel is hidden. Figure 5 shows the resulting Voronoi map by the down sweep of a binary image with black pixels a, b, and c. Although the closest black pixel of the exclave pixel is b, black pixel a is stored as the closest pixel for it by the down sweep. Also, the 90 degree angles emitted from three black

pixels by the up sweep, the left sweep, and the right sweep do not include the exclave pixel and M_u, M_l, and M_r of it are $(+\infty, +\infty)$. Thus, in the resulting Voronoi map by the 4-directional sweep, black pixel a is assigned as the closest pixel of the exclave pixel. Therefore, the exclave pixel in Figure 5 is hidden by the 4-directional sweep.

Let us evaluate the performance of the 4-directional sweep. For simplicity, we assume the EREW PRAM. Each sweep including the down sweep runs $O(\sqrt{n})$ time using $\sqrt{n}$ processors. After that, the closest pixel of $M_d(i,j)$, $M_u(i,j)$, $M_l(i,j)$, and $M_r(i,j)$ for every (i,j) can also be computed in $O(\sqrt{n})$ time using $\sqrt{n}$ processors. Thus, we have,

Theorem 1: The 4-directional sweep computes the mixed Voronoi map in $O(\sqrt{n})$ time using $\sqrt{n}$ processors on the EREW PRAM.

B. Mixed-to-complete conversion of Voronoi maps

Exclave pixels in a mixed Voronoi map M may be hidden around Voronoi points. We find all hidden exclave pixels and assigns the correct closest black pixels for them to obtain the complete Voronoi map C. We first find Voronoi points by reading M for every set $S(i,j) = \{M(i,j), M(i,j+1), M(i+1,j), M(i+1,j+1)\}$ $(0 \leq i,j \leq \sqrt{n}-2)$ of 2×2 pixels. If a set $S(i,j)$ includes three or four black pixels, the corresponding three or four Voronoi cells are adjacent and the Voronoi points for them can be computed. For example, suppose that $S(i,j)$ has three pixels a, b, and c as illustrated in Figure 6. Let l_{ab} and l_{ac} be Voronoi edges. In this figure, two Voronoi edges l_{ab} and l_{ac} lie between pixels $(i,j+1)$ and $(i+1,j+1)$. In Case 1, $(i,j+2)$ the Voronoi point connecting l_{ab} and l_{ac} is in the right side of column $j+2$. If this is the case, pixel $(i,j+2)$ may be a hidden exclave pixel. Further, if the angle of l_{ab} and l_{ac} is very small, multiple exclave pixels may be hidden. In Case 2, $(i+1,j)$ is an appearing exclave pixels. Thus, $(i+2,j)$ is also an appearing exclave pixel by the left sweep. So, if the Voronoi point is in the left side of column j, no hidden exclave pixel exists. Since we have positions of three black pixels a, b, and c, we can compute line segments l_{ab} and l_{ac}, and the Voronoi point connecting them in an obvious way. If $S(i,j)$ includes four black pixels, we can find all hidden exclave pixels for them by executing all four sets of 3 pixels in $S(i,j)$.

To find all hidden exclave pixels in parallel, we assign one processor for each $S(i,j)$ $(0 \leq i,j \leq \sqrt{n}-2)$. If $S(i,j)$ has less than 3 black pixels, the processor terminates. Otherwise, it finds hidden exclave pixels and assigns the correct black pixel to it one by one. If it finds an appearing exclave pixel, it terminates. Since a processor assigned to $S(i,j)$ that includes the appearing exclave pixel finds remaining hidden exclave pixels, a processor that finds an appearing exclave pixel should be terminated to avoid redundant computation.

In this parallel algorithm, $(\sqrt{n}-1)^2$ processors are used. Let h be the maximum number of consecutive hidden exclave pixels. In practice, h is very small. Since the total number of exclave pixels is less than n, we have,

Lemma 2: The mixed Voronoi map of size $\sqrt{n}\times\sqrt{n}$ can be converted into the complete Voronoi map in $O(h)$ time and $O(n)$ total work using $(\sqrt{n}-1)^2$ processors.

C. Mixed-to-connected conversion of Voronoi maps

We will show how we convert a mixed Voronoi map into the connected Voronoi map. For this purpose, we hide all exclave pixels in the mixed Voronoi map. We can use the same technique in the mixed-to-complete conversion to find all appearing exclave pixels. However, the computation of finding exclave pixels is a bit costly. We will show a simpler method for mixed-to-connected conversion.

We use n processors and each processor is assigned to a pixel (i,j) $(0 \leq i,j \leq \sqrt{n}-1)$ and works for determining if (i,j) is an appearing exclave pixel. We use Figure 7 to show how it can be determined. Let (i,j) be a pixel with $M(i,j) = (i',j')$, that is, the 4-directional sweep assigns black pixel (i',j') to pixel (i,j). As illustrated in Figure 7, we assume that (i',j') is in the upper left of (i,j), that is, $i' < i$ and $j' < j$. A processor reads $M(i-1,j-1)$, $M(i-1,j)$, and $M(i,j-1)$ and if at least one of them is (i',j') then the processor terminates. Otherwise, (i,j) is an appearing exclave pixel that should be hidden. If this is the case, the processor chooses the closest pixel of $M(i-1,j-1)$, $M(i-1,j)$, and $M(i,j-1)$, and writes it in $M(i,j)$. Further, the processor reads $M(i,j+1)$, $M(i+1,j+1)$, and $M(i,j+1)$ and if one of them is (i',j'), then it is also appearing exclave pixel that should be hidden. For example, if $M(i,j+1) = (i',j')$ as illustrated in Figure 7, then $(i,j+1)$ is a exclave pixel. Note that the processor assigned to $(i,j+1)$ has terminated, because $M(i,j) = (i',j')$. Thus, the processor assigned to (i,j) performs the same procedure for hiding exclave pixel $(i,j+1)$. We repeat the same procedure for three neighbor pixels in the lower right of $(i,j+1)$, until all appearing exclave pixels from (i,j) are hidden.

Let e be the number of appearing exclave pixels. In practice, e is very small. Since $O(1)$ operation is performed for each appearing exclave pixel, we have,

Lemma 3: The mixed Voronoi map of size $\sqrt{n}\times\sqrt{n}$ can be converted into the complete Voronoi map in $O(e)$ time and $O(n)$ total work using n processors.

IV. GPU implementations for the Voronoi maps and the Euclidean distance map

We assume that input binary images are stored in the global memory of the GPU. The pixel values are *bit-packed*, that is, the binary values of 32 consecutive pixels in a row is stored in a 32-bit word. Thus, a binary image of size $\sqrt{n}\times\sqrt{n}$ is stored in an array of 32-bit words of size $\frac{n}{32}$. For efficient global memory access, we assume that words are arranged in column major order as illustrated in Figure 8. A square block of 32×32 pixels are in consecutive addresses of the array, global memory access to it is coalesced.

A. The 4-directional sweep

We will show three implementation methods for the 4-directional sweep called *single-block (synchronous) sweep*,

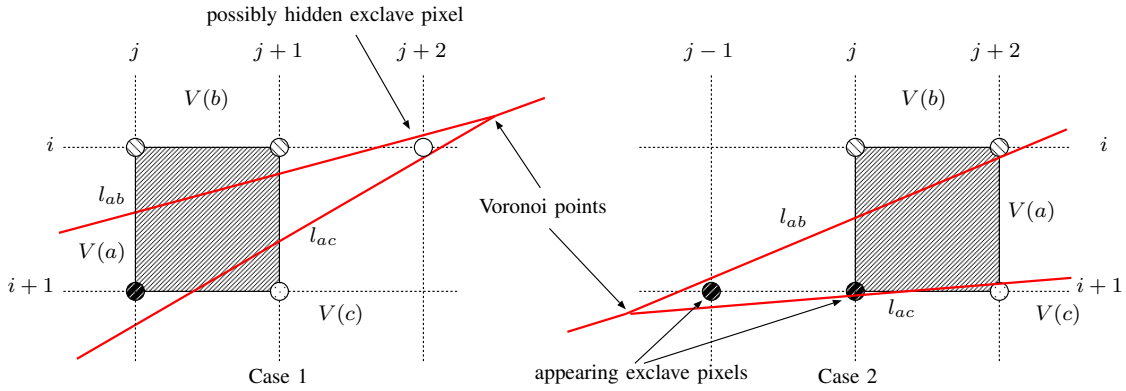

Fig. 6. Finding hidden exclave pixels for three Voronoi cells $V(a)$, $V(b)$, and $V(C)$

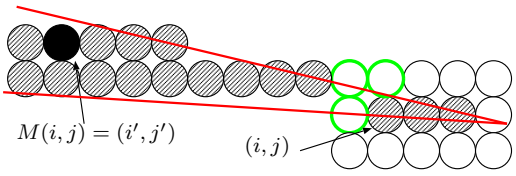

Fig. 7. Hide an exclave pixel to obtain the connected Voronoi map

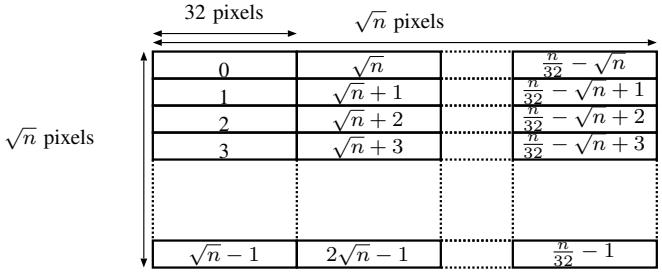

Fig. 8. The column major order bit-packed arrangement of a $\sqrt{n} \times \sqrt{n}$ binary image

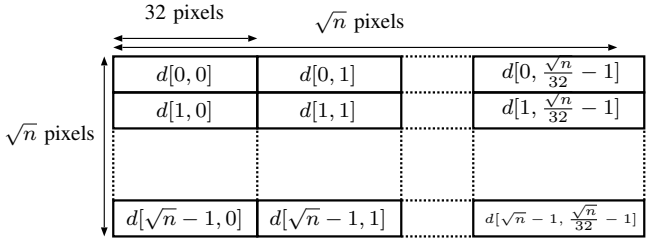

Fig. 9. $\sqrt{n} \times \frac{\sqrt{n}}{32}$ tasks for the down sweep

multiple-block (synchronous) sweep, and *warp-wise (asynchronous) sweep*. The single-block sweep uses one CUDA block for each of the down, up, left, and right sweeps. Since barrier synchronization is necessary after each row (or each column) is computed, __syncthreads(), which synchronizes all threads in the same CUDA block, is executed. The 4-directional sweep uses only four CUDA blocks for a binary image. Since only four streaming processors are used for a binary image and the other streaming processors are idle, we can not get enough acceleration performance.

The multiple-block sweep uses multiple CUDA blocks for each of the down, up, left, and right sweeps. Since a lot of CUDA blocks can be invoked, all streaming processors are assigned CUDA blocks and cores in them are fully used. However, separated CUDA kernel must be called for each row (or each column). Hence, non-negligible large overhead for calling a CUDA kernel is imposed many times.

The warp-wise sweep assigns a warp of 32 threads to compute the closest black pixels of 32 pixels in a row (or a column). We call the computation for consecutive 32 pixels *a task*. Clearly, the 4-directional sweep involves $\frac{n}{32} \cdot 4 = \frac{n}{8}$ tasks. We have developed *the dynamic soft synchronization technique*, which uses a counter in the global memory to arrange a warp to a task. Every invoked warp performs atomic-increment for the counter and performs the task corresponding to the value of the counter. It repeats this operation until no more unexecuted tasks exist. Since barrier synchronization using __syncthreads() or separated kernel calls is not necessary, the warp-wise sweep has no barrier synchronization overhead.

We will explain more details for these three implementations for the 4-directional sweep. For later reference, let $d[i,j]$ denote the task to compute $M_d(i,32j), M_d(i,32j+1), \ldots, M_d(i,32j+31)$ in the down sweep as illustrated in Figure 9. Similarly, we define $u[i,j], l[i,j]$, and $r[i,j]$ for the up/left/right sweeps, respectively. Clearly, to start task $\alpha[i,j]$, ($\alpha \in \{d,u,l,r\}$), tasks $\alpha[i-1,j-1], \alpha[i-1,j]$, and $\alpha[i-1,j+1]$ (if exist) must be completed.

B. Single-block (synchronous) sweep

We show how the down sweep is implemented by the single-block sweep. The other sweeps can be implemented in the same way.

We first assume that $\sqrt{n} \leq 1024$. The down sweep uses one CUDA block with $\sqrt{n}$ threads. Warp j ($0 \leq j < \frac{\sqrt{n}}{32}$) performs task $d[0,j], d[1,j], \ldots, d[\sqrt{n},j]$ in turn. When task $d[i,j]$ starts, tasks $d[i-1,j-1]$, $d[i-1,j]$, and $d[i-1,j+1]$

must be computed. Thus, when the CUDA block completes tasks $d[i,0]$, $d[i,1]$, ..., $d[i,\frac{\sqrt{n}}{32}-1]$ ($0 \le i \le \sqrt{n}-2$), all threads execute __syncthreads() for barrier synchronization. Each thread j ($0 \le j \le \sqrt{n}-1$) uses a 32-bit register m to store $M_d(i,j)$. Hence, to compute $M_d(i+1,j)$, thread j must read the values of register m of thread $j-1$ and $j+1$. This can be done by warp shuffle function __shfl(), which copies the value stored in a 32-bit register of a different thread in the same warp very efficiently. However, the first and the last threads in a warp must read the values of register m of a thread in different warp. This can be done through the shared memory. More specifically, threads 0, 32, 64, ... and 31, 63, 95, ... write the values of m to the shared memory after the angle-restricted closest black pixels for tasks of a particular row is computed. They are read by the first/last threads in a different warp for tasks of the following row.

We should note several implementation issues in terms of memory access both for the binary image B and for M_d, M_u, M_l, and M_r stored in the global memory. Since B is stored in the column major order bit-packed arrangement (Figure 8), 32 threads in a warp read only one 32-bit word for a task in the down/up sweep. Also, for a task in the left/right sweep, 32 threads in a warp read consecutive 32 32-bit word. Thus, the 4-directional sweep performs coalesced memory access to B. We assume that $M_d(i,j)$ and $M_u(i,j)$ stored in the $((i\cdot\frac{\sqrt{n}}{32}+j)$-th 32-bit word (i.e. row major order) in the global memory space for M_d and M_u, respectively. Hence, we can guarantee that memory access to M_d/M_u by each task is coalesced. On the other hand, $M_l(i,j)$ and $M_r(i,j)$ stored in the $(i+j\cdot\frac{\sqrt{n}}{32})$-th 32-bit word (i.e. column major order) in the global memory space for M_r and M_l, respectively. Similarly, memory access to M_d/M_u by each task is coalesced.

If $\sqrt{n} > 1024$, then each warp must be assigned multiple tasks in each row. We assign tasks $d[i,pj]$, $d[i,pj+1]$, ..., $d[i,pj+p-1]$ to warp j ($0 \le j \le 31$), where $p = \frac{\sqrt{n}}{1024}$. Since each thread works for p columns, it uses p 32-bit registers to store the closest black pixels. Each warp j performs tasks $d[i,pj]$, $d[i,pj+1]$, ..., $d[i,pj+p-1]$ in turn.

C. Multiple-block (synchronous) sweep

We should use multiple CUDA blocks for each sweep for full utilization of streaming multiprocessors. Let w be the number of warps of a CUDA block. Since each CUDA block has $32w$ threads, we use $p = \frac{\sqrt{n}}{32w}$ CUDA blocks for each sweep. For barrier synchronization, we need one CUDA kernel must be called for each row. Thus, each kernel call i ($0 \le i \le \sqrt{n}-1$) invokes p CUDA blocks for each sweep, and CUDA block j ($0 \le j \le p-1$) works for tasks $d[i,wj]$, $d[i,wj+1]$, ..., $d[i,wj+w-1]$. The resulting values of $M_d(i,32wj), M_d(i,32wj+1)$, ..., $M_d(i,32wj+32w-1)$ are written in the global memory.

D. Warp-wise (asynchronous) sweep

We assign serial numbers for all tasks of the 4-directional sweep, such that each task $d[i,j], u[i,j], r[i,j], l[i,j]$ is assigned serial numbers $4\frac{\sqrt{n}}{32}i+j$, $4\frac{\sqrt{n}}{32}i+\frac{\sqrt{n}}{32}+j$, $4\frac{\sqrt{n}}{32}i+2\frac{\sqrt{n}}{32}+j$, and $4\frac{\sqrt{n}}{32}i+3\frac{\sqrt{n}}{32}+j$, respectively. Let $t[k]$ ($0 \le i \le 4\frac{n}{32}-1$) denote the task with serial number k. The reader should refer to Figure 10 illustrating serial numbers assigned to tasks for a 128×128 binary image. We can see that $4\times\frac{128\times128}{32} = 2048$ tasks are assigned serial numbers so that tasks computed earlier have smaller serial numbers.

We use a zero-initialized counter c arranged in the global memory to assign a task to a warp. A CUDA kernel call invokes $4\cdot\frac{\sqrt{n}}{32}$ warps. The first thread of every warp performs atomicADD(&c,1), which increments the value of c exclavely and returns the old value before addition. If the return value of atomicADD is k is smaller than $4\cdot\frac{n}{32}$ (the total number of tasks), then the warp performs task $t[k]$. Note that, before performing task $t[k]$, it must be checked if all necessary angle-restricted closest pixel are computed by three tasks. After the warp completes task $t[k]$, it repeats the same procedure until the return value of atomicADD(&c,1) is larger than or equal to $4\cdot\frac{n}{32}$, which is the total number of tasks. Since atomicADD(&c,1) by the first threads of warps return 0, 1, 2, ..., all $4\cdot\frac{n}{32}$ tasks are performed in turn.

Clearly, only one CUDA kernel is called for the warp-wise sweep, and so it has no overhead by separated kernel calls. However, each warp repeatedly reads the the resulting values of three tasks to perform an assigned task if they are not finished. To minimize the overhead for this iteration, $4\frac{\sqrt{n}}{32}$ warps are invoked for the 4-directional sweep. For example, for a 128×128 binary image, we use 16 warps, say, 8 CUDA blocks with 2 warps each. Initially, they are arranged 16 tasks from 0 to 15. If one of the warps completes the assigned task, then it is rearranged task 16. It repeatedly reads the resulting values of tasks 0 and 1 which are necessary to perform task 16, until these tasks are completed. Since tasks 0 and 1 are assigned to warps at the beginning, with high probability, tasks 0 and 1 are completed when the warp start executing task 16.

The reader may think that such dynamic task allocation using the counter c is not necessary and fixed task allocation is more efficient. For example, for a 128×128 binary image in Figure 10, each warp i ($0 \le i \le 15$) performs tasks i, $i+16$, $i+32$, ... in turn. However, the algorithm using fixed task allocation may stall due to the deadlock. In CUDA, it is not guaranteed that all CUDA blocks are dispatched to streaming processors. If one of them is not dispatched, this algorithm never terminates. On the other hand, the dynamic task allocation using the counter c works correctly, even if less than $4\frac{\sqrt{n}}{32}$ warps are dispatched to streaming processors.

E. Combining four matrix M_d, M_u, M_l, and M_r

After M_d, M_u, M_l, and M_r are computed, we combine them to obtain the mixed Voronoi map M by computing $M(i,j) \leftarrow \mathrm{closest}((i,j), M_d(i,j), M_u(i,j), M_l(i,j), M_r(i,j))$ for all (i,j). We partition M into $\frac{\sqrt{n}}{32}\times\frac{\sqrt{n}}{32}$ submatrices of size 32×32 each. A CUDA block with 32×32 threads is assigned to a submatrix and each thread computes an element. For this purpose, each thread (i,j) ($0 \le i,j \le 31$)

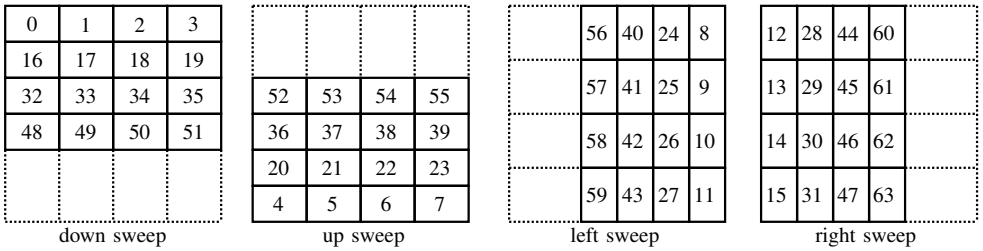

Fig. 10. Serial numbers assigned to tasks of the 4-directional sweep for a 128×128 binary image

read $M_d(i,j), M_u(i,j), M_l(i,j)$, and $M_r(i,j)$. Recall that M_d and M_u are row major order, while M_l and M_r are column major order. Thus, memory access to M_l and M_r is not coalesced. For coalesced memory access, we the diagonal arrangement of the shared memory [22]. For example, each $M_l(i,j)$ arranged in offset $i + j\sqrt{n}$ (column major order) is read by thread (j,i) (i.e. thread $32j + i$) and written in $s[i,(i+j) \bmod 32]$, where s is a 2-dimensional array of size 32×32 in the shared memory. After that, each thread (i,j) read $s[i,(i+j) \bmod 32]$, which stores $M_l(i,j)$. This transposing a submatrix is very efficient, because reading M_l by a warp of 32 threads $(j,0),(j,1),\ldots,(j,31)$ is coalesced and reading and writing by a warp of 32 threads $(i,0),(i,1),\ldots,(i,31)$ are conflict-free.

F. Mixed-to-complete and mixed-to-connected conversions

We simply use one thread for each set of 2×2 elements $\{M(i,j), M(i,j+1), M(i+1,j), M(i+1,j+1)\}$ in M for the mixed-to-complete conversion. If they have 1 or 2 black pixels, then the thread terminates immediately. Otherwise, it finds hidden exclave pixels and store the closest black pixel in $M(i,j)$. Reading 2×2 elements is coalesced, but memory access to find hidden exclave pixels is not coalesced. However, the mixed-to-complete conversion runs faster than the 4-directional sweep because the number of sets of 2×2 elements that have 3 or 4 black pixels is very small.

Similarly, we use one thread for each element in M for the mixed-to-connected conversion. A thread terminates, if the assigned pixel is not an appearing exclave pixel. If the assigned pixel is an appearing exclave pixel, then the thread hides it and continues to find neighboring exclave pixels. Since a binary image has very few appearing exclave pixels, most of threads terminate immediately. Also, the computation for determining if a pixel is an appearing exclave pixel is very light, the mixed-to-connected conversion runs faster than the mixed-to-complete conversion.

V. Experimental results

We have implemented *the jump flooding algorithm (JFA)* [17], *the parallel banding algorithm (PBA)* [21], and our algorithm for computing the Voronoi maps and the Euclidean distance map. We have used GeForce GTX 1080 GPU, which has 20 streaming multiprocessors with 128 cores each. In particular, for implementing the PBA, we have modified source codes provided by the authors of the paper [21] to attain the best performance on GeForce GTX 1080 GPU. For reference, we have implemented a sequential algorithm for the Euclidean distance map based on [12] and evaluated the running time on Intel Core i7-4790 CPU (3.6GHz).

Fig. 11. The binarized image of Lena by a threshold such that a half of the pixels are black

We have used four images, "Lena" (Figure 11), "50% random", "1% random", and "0.01% random." In "p% random" ($p = 50, 1, 0.01$), black pixels are placed at random such that p% out of all pixels are black. Hence, in "1% random" and "0.01% random", regions of sizes 10×10 and 100×100 have expected one black pixel, respectively. On the other hand, "50% random" has too many black pixels and any group of 32 consecutive pixels have at least one black pixel with probability $1 - 2^{-32}$. Although "Lena" also has 50% black pixels, they are clustered and all pixels in some groups of 32 consecutive pixels are all black or all white. Thus, we can expect that computation for "Lena" can be faster than "50% random" due to a smaller number of warp divergence.

Table I shows the running time for a single binary image of sizes from 512×512 (256K) to $16K \times 16K$ (256M). We have used the multiple-block (synchronous) sweep and the warp-wise (asynchronous) sweep to compute the mixed Voronoi map of a single image. In both implementations, CUDA blocks with 2 warps of 64 threads are used to attain 100% thread occupancy. The warp-wise (asynchronous) sweep always runs

TABLE I
THE RUNNING TIME FOR A SINGLE BINARY IMAGE IN MILLISECONDS

Input	Algorithm	512×512	$1K \times 1K$	$2K \times 2K$	$4K \times 4K$	$8K \times 8K$	$12K \times 12K$	$16K \times 16K$
Lena	CPU Euclidean distance map	6.449	25.06	122.6	591.0	2690	6259	11600
	JFA Euclidean distance map	**0.1936**	**0.8169**	3.168	13.53	58.23	137.2	252.9
	PBA Euclidean distance map	0.3059	0.8473	**2.400**	8.523	33.86	75.92	134.5
	mixed Voronoi map (multiple-block)	1.624	3.423	6.956	14.61	35.94	64.59	96.99
	mixed Voronoi map (warp-wise)	0.6288	1.258	2.629	6.502	19.98	47.80	79.47
	connected Voronoi map (warp-wise)	0.6369	1.287	2.725	6.861	21.35	50.51	84.26
	complete Voronoi map (warp-wise)	0.7003	1.441	3.098	7.772	23.74	55.03	90.44
	Euclidean distance map (warp-wise)	0.7081	1.463	3.171	**8.028**	**24.73**	**57.26**	**94.50**
50% random	CPU Euclidean distance map	7.463	31.70	157.5	748.4	3416	7950	14530
	JFA Euclidean distance map	**0.2095**	0.9719	3.977	18.15	82.00	197.3	367.1
	PBA Euclidean distance map	0.2714	**0.8552**	**2.700**	10.62	43.37	94.79	170.1
	mixed Voronoi map (multiple-block)	1.545	3.441	7.040	14.81	36.51	65.86	101.3
	mixed Voronoi map (warp-wise)	0.6155	1.212	2.682	6.786	20.79	49.52	84.07
	connected Voronoi map (warp-wise)	0.6251	1.241	2.781	7.166	22.30	53.04	89.47
	complete Voronoi map (warp-wise)	0.6425	1.308	3.022	8.068	25.81	60.49	104.4
	Euclidean distance map (warp-wise)	0.6517	1.343	3.120	**8.446**	**27.32**	**62.81**	**109.9**
1% random	CPU Euclidean distance map	5.815	25.07	128.7	628.8	2907	6808	12540
	JFA Euclidean distance map	0.2117	0.9366	3.966	18.00	81.49	194.8	364.9
	PBA Euclidean distance map	**0.2072**	**0.5816**	**2.006**	**7.644**	33.10	74.33	133.6
	mixed Voronoi map (multiple-block)	1.541	3.186	6.862	14.28	36.47	65.76	101.5
	mixed Voronoi map (warp-wise)	0.6394	1.183	2.685	6.773	20.71	49.29	83.73
	connected Voronoi map (warp-wise)	0.6503	1.213	2.790	7.180	22.32	53.23	89.61
	(complete) Voronoi map (warp-wise)	0.6649	1.240	2.858	7.433	23.29	55.01	93.66
	Euclidean distance map (warp-wise)	0.6758	1.276	2.965	7.840	**24.90**	**57.24**	**99.75**
0.01% random	CPU Euclidean distance map	4.656	20.41	109.4	551.3	2596	6098	11250
	JFA Euclidean distance map	0.2114	0.9257	3.967	18.00	81.48	193.6	364.8
	PBA Euclidean distance map	**0.1275**	**0.3737**	**1.396**	**5.904**	25.70	58.61	106.5
	mixed Voronoi map (multiple-block)	1.659	3.307	6.691	14.59	36.43	65.61	103.1
	mixed Voronoi map (warp-wise)	0.6404	1.172	2.690	6.772	20.68	49.09	83.49
	connected Voronoi map (warp-wise)	0.6510	1.203	2.795	7.181	22.28	52.43	89.27
	complete Voronoi map (warp-wise)	0.6490	1.202	2.794	7.163	22.24	52.97	89.26
	Euclidean distance map (warp-wise)	0.6600	1.233	2.900	7.567	**23.83**	**55.16**	**95.25**

TABLE II
THE RUNNING TIME FOR 2,4,8,16,32,64, AND 100 BINARY IMAGES WITH $2K \times 2K$ ($4M$) PIXELS

Input	Algorithm	2	4	8	16	32	64	100
Lena	CPU Euclidean distance map	262.9	514.3	1029	2111	4396	8903	12990
	JFA Euclidean distance map	6.288	12.54	25.02	50.04	100.1	194.3	314.0
	PBA Euclidean distance map	4.427	8.305	16.04	31.56	62.68	125.1	195.3
	mixed Voronoi map (single-block)	3.046	3.661	6.313	12.03	22.28	42.96	66.98
	connected Voronoi map (single-block)	3.221	4.000	6.987	13.37	24.94	48.28	75.29
	complete Voronoi map (single-block)	3.842	5.149	9.174	17.62	33.36	64.92	101.4
	Euclidean distance map (single-block)	**3.974**	**5.406**	**9.678**	**18.62**	**35.34**	**68.90**	**107.6**
50% random	CPU Euclidean distance map	322.6	645.3	1281	2589	5129	10310	17540
	JFA Euclidean distance map	8.000	15.96	31.87	64.22	129.9	253.7	408.4
	PBA Euclidean distance map	5.178	10.12	19.49	38.43	76.84	154.3	240.9
	mixed Voronoi map (single-block)	3.404	4.154	7.295	14.06	25.98	50.37	78.27
	connected Voronoi map (single-block)	3.600	4.542	8.062	15.59	28.98	55.91	86.62
	complete Voronoi map (single-block)	4.047	5.411	9.816	19.06	36.05	70.50	109.5
	Euclidean distance map (single-block)	**4.181**	**5.670**	**10.33**	**20.08**	**38.10**	**74.60**	**115.9**
1% random	CPU Euclidean distance map	264.7	519.4	1049	2097	4223	8415	13410
	JFA Euclidean distance map	7.957	15.87	31.69	64.13	130.1	249.2	410.3
	PBA Euclidean distance map	**3.784**	7.329	14.32	28.36	56.45	112.6	175.7
	mixed Voronoi map (single-block)	3.390	4.130	7.252	13.98	25.78	50.07	78.03
	connected Voronoi map(single-block)	3.614	4.561	8.109	15.68	28.87	56.25	87.47
	complete Voronoi map (single-block)	3.729	4.796	8.562	16.57	30.98	60.06	93.52
	Euclidean distance map (single-block)	3.862	**5.051**	**9.064**	**17.56**	**32.98**	**64.06**	**99.71**
0.01% random	CPU Euclidean distance map	227.7	444.4	892.4	1798	3594	7210	16000
	JFA Euclidean distance map	7.842	15.64	31.22	62.57	125.4	249.3	395.1
	PBA Euclidean distance map	**2.739**	5.375	10.70	21.33	42.53	84.93	132.6
	mixed Voronoi map (single-block)	3.372	4.114	7.234	13.95	25.74	49.98	77.79
	connected Voronoi map (single-block)	3.586	4.538	8.076	15.62	29.08	56.15	87.11
	complete Voronoi map (single-block)	3.574	4.510	8.014	15.50	28.84	55.83	86.86
	Euclidean distance map (single-block)	3.707	**4.763**	**8.508**	**16.48**	**30.81**	**59.77**	**92.98**

faster than the multiple-block (synchronous) sweep. Roughly speaking, we can think that the difference of the running time of the multiple-block sweep and the warp-wise sweep is the overhead of synchronization by iterative kernel calls by the multiple-block sweep. For example, the overhead ratio for Lena image of size $4K \times 4K$ is $\frac{14.61-6.502}{14.61} = 55\%$. The ratio of kernel call overhead ranges from 17% to 64.7% of the total computing time of the multiple-block sweep.

Table I also shows the running time of connected/complete Voronoi maps and Euclidean distance maps. They use the 4-directional sweep by the warp-wise sweep. From the table, we can see that the mixed-to-connected, the mixed-to-complete conversions and the complete-to-distance conversion to obtain the Euclidean distance map from the complete Voronoi map are not dominant. For example, the mixed Voronoi map for $16K \times 16K$ 50% random image is computed in 84.07ms. We can say that the mixed-to-connected and the mixed-to-complete conversions take $89.47 - 84.07 = 5.40$ ms and $104.4 - 84.07 = 20.3$ ms, respectively. Also, the complete-to-distance conversion is only in $109.9 - 104.4 = 5.5$ ms.

For each binary image of each size in Table I, the running time of the fastest implementation among the JFA, the PBA, and our warp-wise Euclidean distance map algorithm is bold-faced. The JFA can be the best only for images with 512×512 pixels. Our warp-wise Euclidean distance map algorithm is always faster than the others if the size of images is larger than or equal to $8K \times 8K$. For binary images of size $16K \times 16K$, our algorithm is up to 1.54 times faster than the PBA, up to 3.83 times faster than the JFA and up to 132 times faster than the CPU implementation.

To see the throughput of the computation, we have evaluated the running time for 2, 4, 8, 16, 32, 62, and 100 binary images of size $2K \times 2K$. We have used single-block (synchronous) sweep to compute the mixed Voronoi map. For each sweep, we use one CUDA block with 1024 threads. Hence, each warp of 32 threads is assigned to 64 columns (or rows). From Table II, we can see that our implementation of the Euclidean distance map is the faster than the JFA and the PBA for 4 or more images. The PBA can be faster than ours for 2 images because only 8 CUDA blocks are used in our implementation. For 100 images, our implementation is 1.43-2.08 times faster than the PBA and 2.92-4.25 times faster than the JFA. Also, it is 121-172 times faster than the CPU implementation. Further, since the Euclidean distance map and the Voronoi maps of 100 images with $2K \times 2K$ $(4M)$ pixels can be computed in approximately 100 milliseconds, our GPU implementations have potentiality to process Full HD videos with 2.1M pixels in more than 1000 fps.

VI. Conclusion

We have shown a GPU implementation that computes the mixed/connected/complete Voronoi maps and the Euclidean distance map of a binary image. The idea of our GPU implementation is to compute the mixed Voronoi map by a simple operation. After that, it is converted into the connected/complete Voronoi maps and the Euclidean distance map. For binary images of size $16K \times 16K$, our algorithm is up to 1.59 times faster than the PBA and 139 times faster than the CPU implementation. For 100 images, our implementation is 1.43-2.08 times faster than the PBA, 2.92-4.25 times faster than the JFA, and 121-172 times faster than the CPU implementation.

References

[1] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[2] Y. Takeuchi, D. Takafuji, Y. Ito, and K. Nakano, "ASCII art generation using the local exhaustive search on the GPU," in *Proc. of International Symposium on Computing and Networking*, Dec. 2013, pp. 194–200.

[3] H. Kouge, Y. Ito, and K. Nakano, "A GPU implementation of clipping-free halftoning using the direct binary search," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (LNCS 8630)*, Aug. 2014, pp. 57–70.

[4] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 3.1," 2010.

[5] ——, "NVIDIA CUDA C programming guide version 7.0," Mar 2015.

[6] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 260–276, July 2011.

[7] H. Breu, J. Gil, D. Kirkpatrick, and M. Werman, "Linear time Euclidean distance transform algorithms," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 17, no. 5, pp. 529–533, May 1995.

[8] L. Chen, "Optimal algorithm for complete Euclidean distance transform," *Chinese J. Computers*, vol. 18, no. 8, pp. 611–616, 1995.

[9] L. Chen and H. Chuang, "A fast algorithm for Euclidean distance maps of a 2-d binary image," *Information Processing Letters*, vol. 51, pp. 25–29, 1994.

[10] T. Hirata, "A unified linear-time algorithm for computing distance maps," *Information Processing Letters*, vol. 58, pp. 129–133, 1996.

[11] A. Fujiwara, T. Masuzawa, and H. Fujiwara, "An optimal parallel algorithm for the Euclidean distance maps of 2-d binary images," *Information Processing Letters*, vol. 54, pp. 295–300, 1995.

[12] T. Hayashi, K. Nakano, and S. Olariu, "Optimal parallel algorithm for finding proximate points, with applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 12, pp. 1153–1166, December 1998.

[13] S. Pavel and S. Akl, "Efficient algorithms for the Euclidean distance transform," *Parallel Processing Letters*, vol. 5, no. 2, pp. 205–212, 1995.

[14] L. K and Z. S. Q, *Parallel Computing Using Optical Interconnections*. Boston, USA: Kluwer Academic Publishers, 1998.

[15] L. Chen, P. Yi, C. Yixin, and X. Xiaohua, "Efficient parallel algorithms for Euclidean distance transform," *The Computer Journal*, vol. 47, no. 6, pp. 694–700, 2004.

[16] Y.-H. Lee, S.-J. Horng, T.-W. Kao, F.-S. Jaung, Y.-J. Chen, and H.-R. Tsai, "Parallel computation of exact Euclidean distance transform," *Parallel Computing*, vol. 22, no. 2, pp. 311–325, 1996.

[17] G. Rong and T.-S. Tan, "Jump flooding in GPU with applications to Voronoi diagram and distance transform," in *Proc. of Symposium on Interactive 3D graphics and games*, March 2006, pp. 109 – 116.

[18] J. Schneider, M. Kraus, and R. Westermann, "GPU-based euclidean distance transforms and their application to volume rendering," *Computer Vision, Imaging and Computer Graphics. Theory and Applications*, pp. 215–228, 2010.

[19] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing Euclidean distance map with efficient memory access," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 68–76.

[20] ——, "Accelerating computation of Euclidean distance map using the GPU with efficient memory access," *Journal of Parallel, Emergent and Distributed Systems*, vol. 28, no. 5, pp. 383–406, 2013.

[21] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan, "Parallel banding algorithm to compute exact distance transform with the GPU," in *The Proc. of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, Feb. 2010, pp. 83–90.

[22] K. Nakano, "Simple memory machine models for GPUs," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 29, no. 1, pp. 17–37, 2014.