A work efficient parallel algorithm for exact Euclidean Distance Transform

Manduhu and Mark W. Jones

***Abstract*—A fully-parallelized work-time optimal algorithm is presented for computing the exact Euclidean Distance Transform (EDT) of a 2D binary image with the size of $n \times n$. Unlike existing PRAM and other algorithms, this algorithm is suitable for implementation on modern SIMD architectures such as GPUs. As a fundamental operation of 2D EDT, 1D EDT is efficiently parallelized first. Specifically, the GPU algorithm for the 1D EDT, which uses CUDA binary functions such as ballot(), ffs(), clz() and shfl(), runs in $O(log_{32}n)$ time and performs $O(n)$ work. Using the 1D EDT as a fundamental operation, the fully-parallelized work-time optimal 2D EDT algorithm is designed. This algorithm consists of three steps. Step 1 of the algorithm runs in $O(log_{32}n)$ time and performs $O(N)$ (N=n^2) of total work on GPU. Step 2 performs $O(N)$ of total work and has an expected time complexity of $O(logn)$ on GPU. Step 3 runs in $O(log_{32}n)$ time and performs $O(N)$ of total work on GPU. As far as we know, this algorithm is the first fully-parallelized and realized work-time optimal algorithm for GPUs. Experimental results show that this algorithm outperforms prior state-of-the-art GPU algorithms.**

***Index Terms*—work-time optimal parallel algorithm, SIMD architecture, 1D EDT, 2D EDT, binary operations on GPU.**

I. Introduction

GIVEN a binary image with $n \times n$ pixels each of them either white or black, Euclidean Distance Transform (EDT) computes the distance of each pixel to the nearest black pixel which is termed *site* or *feature point*. The Euclidean distance transform is useful for a variety of applications including image processing, computer vision, pattern recognition, shape analysis and computational geometry [1][2][3]. Obviously, the 2D EDT can be computed in $O(n^4)$ time by an exhaustive brute-force searching algorithm. Many efficient algorithms have been proposed in the past. Breu *et al.* [4] proposed the first $O(n^2)$ time (linear time) exact EDT algorithm based on Voronoi Diagram Intersections. The following two survey papers [5][6] compare and contrast many state-of-the-art sequential approaches for solving the problem in 2D and 3D, mainly focusing on the computation of exact EDT.

Consider a parallel algorithm which solves a problem with input size n in $T_p(n)$ time using p processors. The amount of work $W(n)$ performed by the algorithm is defined as the product $p \times T_p(n)$. The algorithm is termed work-optimal if $W(n) \in Q(T^*(n))$, where $T^*(n)$ is the running time of the fastest sequential algorithm for the problem. The parallel algorithm is termed work-time optimal [7] if it is work-optimal and, in addition, its running time $T_p(n)$ is shortest among all work-optimal algorithms under the same parallel model. A challenge of parallel algorithm design is to produce not only work-optimal but, indeed, whenever possible, work-time optimal algorithms.

The modern massively parallel architecture [8][9], which is composed of hierarchical memory and SIMD (Single Instruction Multiple Data) capability, creates the potential for algorithms to be transformed into efficient implementations. Especially, Graphics Processing Units (GPUs), enable solutions to problems impossible with previous computing approaches. When we design parallel algorithms for GPUs, we need to consider not only the complexity of the parallel algorithm itself but also the efficient utilization of the hierarchical memory and SIMD capability of the GPUs [10][11]. In this paper, we present a fully-parallelized work-time optimal algorithm for the 2D EDT, which can be implemented efficiently on modern SIMD architectures such as GPUs.

Early attempts for computing the EDT using graphics hardware include the work of Hoff *et al.* [12]. By rendering a right-angled cone for each feature pixel in the image, the approximation of the distance function is obtained. Then depth-testing graphics hardware is used to compute the distance map. Their method suffers from overdrawing and tessellation error. Sud *et al.* [13] proposed a method to use the bilinear interpolation equation to compute the distance vector on a polygon. Their method can compute highly accurate distance maps for complex models, but its complexity is dependent on the number of sites in the image. Therefore, it is not suitable for problems with many sites. Later, several efficient GPU-based algorithms which are either work optimal or time optimal have been proposed including JFA [14], SKW [15], FastGPU [16], PBA [17] and Honda's algorithm [18]. However, none of these algorithms are work-time optimal.

In this paper, we present a fully-parallelized work-time optimal exact EDT algorithm which is suitable for implementation on modern SIMD architectures such as GPUs. Following the idea of PBA, the computation of the exact EDT is performed in a dimension reduction manner. The computation is done in one dimension (row wise) first, then in the second dimension (column wise). The algorithm consists of three steps. The first step of the algorithm runs in $O(log_{32}n)$ time and performs $O(N)$ of total work on GPU. The second step performs $O(N)$ of total work and has an expected time complexity of $O(logn)$ on GPU. The third step runs in $O(log_{32}n)$ time and performs $O(N)$ of total work on GPU. Clearly, this algorithm is a work-time optimal algorithm. As far as we know, this algorithm is the first fully-parallelized work-time optimal algorithm for the GPUs. Experiments demonstrate that this algorithm outperforms prior state-of-the-art GPU algorithms.

Manduhu and Mark W. Jones are with the Department of Computer Science, Swansea University, UK, SA1 8EN.
E-mail: {M.Duhu, M.W.Jones}@Swansea.ac.uk
Manduhu is supported by a Sêr Cymru COFUND fellowship.
Manuscript received April 19, 2005; revised August 26, 2015.

II. LITERATURE

A. Exact EDT

Many algorithms for the exact EDT have been proposed. Saito and Toriwaki [19] proposed a sequential algorithm based on dimensionality reduction, since the squared Euclidean distance value is separable, the distance transform can be computed along each principal direction. Recently, Torelli *et al.* [20] implemented Saito's algorithm on a cluster using MPI. However, Saito's algorithm has a time complexity of $O(n^3)$ and it will result in a poor performance in practice. Meijster *et al.* [21] proposed a two-scan linear-time sequential algorithm which follows the same concepts as described in the Saito's algorithm. Breu *et al.* [4] also proposed a linear-time sequential algorithm with time complexity of $O(n^2)$. Since each pixel of input image needs to be visited at least once, these linear-time algorithms are time optimal. Later, Maurer *et al.* [22] proposed a linear time sequential algorithm for a binary image in arbitrary dimension using dimensionality reduction strategy. Recently, Wang and Tan [23] proposed a method in which perpendicular bisector line is used to improve the locality of computation. They also extended it to an arbitrary dimension [24]. Lotufo and Zampirolli [25] showed a new way to compute the exact EDT based on gray-scale mathematical morphology. In their algorithm, an erosion procedure is repeatedly applied to each image column until the column does not change.

To accelerate the computation of the EDT, various parallel algorithms have been proposed on different parallel models. The fastest parallel algorithm which runs in $O(1)$ time using $O(n^3)$ processors was proposed by Datta and Soundaralakshmi in [32]. Their algorithm was originally designed for Reconfigurable Meshes which require each processor capable of communicating with neighboring processors via interconnection buses. Because of the number of processors needed and the special requirement in the architecture, their method is not suited to modern SIMD processors. There are also many algorithms proposed for the PRAM (Parallel Random Access Machine) model. Lee *et al.* [33] use the dimensionality reduction approach to compute the exact EDT in $O(log^2 n)$ time using $O(N)$ processors on an EREW (Exclusive Read Exclusive Write) PRAM machine. Pavel and Akl [34] proposed an algorithm running in $O(logn)$ time using $O(N)$ processors on a EREW PRAM machine. However, these two algorithms are not work optimal. Considering the computation on each image row, Fujiwara *et al.* [35] proposed a work-optimal algorithm running in $O(logn)$ time using $O(\frac{n}{logn})$ processors on the EREW PRAM and in $O(\frac{logn}{loglogn})$ time using $O(\frac{nloglogn}{logn})$ processors on the CRCW PRAM. Later, Hayashi *et al.* [7] proposed a more efficient algorithm running in $O(logn)$ time using $O(\frac{n}{logn})$ processors on the EREW PRAM and in $O(loglogn)$ time using $O(\frac{n}{loglogn})$ processors on the CRCW PRAM. Since the product of the computing time and the number of processors is $O(n)$, these two algorithms are work optimal.

To efficiently utilize hierarchical memory and SIMD capability of modern GPUs, a special SIMD-like programming paradigm has to be employed. However, the PRAM algorithms mentioned above are too complex to fit into such a paradigm. Therefore, there are few algorithms to compute the exact EDT on GPUs using either the sequential algorithms or the PRAM algorithms for exact EDT. Zampirolli and Filipe [29] proposed a GPU implementation of Lotufo's [25] mathematical morphological algorithm. However, in Lotufo's algorithm, an erosion procedure is repeatedly applied to each image column until the column does not change, and it is computationally inefficient. Later, Zampirolli and Filipe [16] proposed a raster-scan based GPU algorithm termed FastGPU algorithm which can efficiently utilize the hierarchical memory of GPUs. They show that the time complexity of the proposed GPU algorithm is $O(n^3/p)$ where p is the number of available processors. Man *et al.* [30] proposed a SIMD-like algorithm and implemented it on a Multi-core processor and a GPU, respectively. Since the computation on each image row is performed by single thread, their algorithm is with a higher time complexity of $O(n)$.

Cao *et al.* [17] proposed an exact EDT algorithm for GPU termed Parallel Banding Algorithm (PBA). By solving several programming issues of GPU including synchronization cost, occupancy and the efficient utilization of texture cache, the PBA achieved a very good performance. As reported in their paper, the PBA outperforms most of the existing GPU algorithms. The main idea of PBA is to partition an image row into several bands and use a single thread to perform the computation on each band independently. Then merge the results of all bands to produce the final results. Their algorithm consists of three steps, in the first step, 1D EDT is computed along the band. In the second step, the band-wise computation is performed to obtain all closest sites for each image row. In the third step, the distance of each pixel to the closest site is computed using the results of step 2. However, as shown in their algorithm, they fail to fully parallelize the first and the third steps of the algorithm. The time complexity of the first step and the third step is $O(m)$ and $O(n)$, respectively, where m is the band size. Especially, the total work the third step performs is $O(mK)$ where K is the number of closest sites obtained in the whole image. Clearly, in the worst case, it will increase to $O(mN)$ ($N = n^2$). Later, Leung *et al.* [31] extended PBA to compute the EDT on a large 3D surface by storing each binary pixel as a binary bit. However, they have not improved the complexity of PBA. We have compared Zampirolli's GPU algorithm [16] with PBA, and experiments show that, in most cases, Zampirolli's algorithm is about 3 times slower than the PBA. In [36], Macedo *et al.* used the exact EDT to compute the shadow map of 3D virtual scenes, and they also pointed out that the PBA is faster than most of the existing EDT algorithms on GPU.

B. Approximate EDT

Different approximate EDT algorithms which allow small errors in result have been proposed in the past and most of them use scan schemes to achieve linear time complexity. Two widely used methods are Borgefors's Chamfer distance transform [37] and Danielsson's Vector Propagation approach [38]. Chamfer metrics are defined by local masks. The weights

TABLE I: Different GPU algorithms for computing EDT

Reference	Algorithm	Exactness	Time	Work
Schneider et al. [15]	SKW	Approximate	$O(n)$	$O(N)$
Rong and Tan [14]	JFA	Approximate	$O(logn)$	$O(Nlogn)$
Yuan *et al.* [26]	based on JFA	Approximate	$O(logn)$	$O(Nlogn)$
Rong *et al.* [27]	based on JFA	Approximate	$O(logn)$	$O(Nlogn)$
Schneider *et al.* [28]	based on SKW	Approximate	$O(n)$	$O(N)$
Zampirolli and Filipe [29]	morphological erosion	Exact	–	–
Zampirolli and Filipe [16]	FastGPU	Exact	$O(n^3/p)$	–
Man *et al.* [30]	scan-based	Exact	$O(n)$	$O(N)$
Cao *et al.* [17]	PBA	Exact	$O(n)$	$O(mN)$
Leung *et al.* [31]	based on PBA	Exact	$O(n)$	$O(mN)$
Honda *et al.* [18]	based on SKW	Exact	$O(n)$	$O(N)$
In this paper	work-time optimal	Exact	$O(logn)$	$O(N)$

of these masks are chosen to minimize the deviation from the EDT. Chamfer DTs require two raster scans in the image. Tuan Q. Pham [39] parallelized the Chamfer distance transform by processing diagonal pixels simultaneously and implemented it on a quad core processors using OpenMP. Danielsson's algorithm generates the EDT in a similar way as the raster scanning of the Chamfer EDT. However, the propagated information is the absolute value of the relative coordinates of the nearest feature pixel, instead of only the relative distances. Therefore the method propagates two values in the masks, instead of one. Many improvements have also been proposed for Danielsson's algorithm [40][41][42][43]. Yamada [44] proposed a parallel version of Danielsson's algorithm by repeatedly applying a mask on every pixel until no pixel changes its value. This algorithm may be executed on all the pixels in parallel, but is computationally inefficient.

Several GPU algorithms which use Danielsson's vector propagation strategy to compute the approximate distance transform have been presented. Rong and Tan [14] proposed an algorithm termed Jump Flooding Algorithm (JFA) to compute Voronoi map and the EDT on GPUs. The JFA needs $O(logn)$ parallel steps and in each step, the information of a closest site is diffused to 8 pixels in relative position. However, the JFA performs a sub-optimal total work of $O(Nlogn)$ (N=n^2), it runs fast only if the image size is small. Later, Yuan *et al.* [26] improved the performance of JFA by storing a site's ID and coordinate separately. An extension of JFA algorithm [27] was proposed to compute Centroidal Voronoi Tessellation on 3D surfaces. However, these two papers have not improved the complexity of JFA algorithm itself. Schneider *et al.* [15] proposed a raster scan algorithm named SKW based on Danielsson's approach. The SKW allows concurrent propagation for pixels in the same row or column. Later, the SKW have been applied on different applications including 2D image processing and 3D volume rendering [28]. The SKW can be implemented on the GPU with $O(N)$ of total work and the generated distance transform is close to exact. However, the SKW has a sub-optimal time complexity of $O(n)$, and usually does not run faster than JFA. This is because it can only perform parallel propagation operation in one row (or column) at a time. Clearly, the serial nature of the propagation operation lays a restriction on further optimization of these algorithms.

While these approximate algorithms are adequate for many applications, there are cases for which the exact EDT is needed. For instance, the mathematical morphology dilation operator can be implemented as the threshold of a EDT, as presented in [45]. The occasional errors in the approximate EDT could lead to pixels missing from the dilated object. Thus, morphological closing, a dilation followed by an erosion with the same structuring element, could actually remove pixels from the original object, which is in contradiction with the basic properties of mathematical morphology. Satherley and Jones [46] proposed a vector propagation followed by a correction stage to compute an almost exact distance field.

Recently, Honda *et al.* [18] proposed a GPU algorithm which performs the vector propagation operation the same as SKW. A correction algorithm is then applied to correct errors caused by the vector propagation. Therefore, Honda's algorithm can compute the exact EDT of an image. Their algorithm also suffers from the sub-optimal time complexity of $O(n)$ as SKW does. Their algorithm is about 2 times faster than PBA when 100 images are processed on GPU simultaneously. Their algorithm achieved this performance by increasing the occupancy of GPU. For single image, if the size of image is larger than or equal to $8k \times 8k$, their algorithm can achieve a speed up factor of 1.54 compared with PBA. However, if the size of image is smaller than $8k \times 8k$, their algorithm is slower than PBA.

Table I shows the time complexity and total work of related GPU algorithms.

III. 1D Euclidean Disntance Transform

The 1D EDT is a fundamental operation in the 2D EDT. In this section we describe different parallel algorithms for the 1D EDT on EREW PRAM and GPU.

A. Preliminaries

The 1D EDT can be described as a 1D closest point problem. In 1D space, given a set $P = \{p_1, p_2, ..., p_n\}$ of n points in which points are sorted by their coordinates such that $coord(p_1) < coord(p_2) < ... < coord(p_n)$, we assume there are m $(0 \leq m \leq n)$ feature points within P and the distribution of these feature points are unpredictable. The closest point problem of P is to find the closest feature point for every point p_i $(1 \leq i \leq n)$ in P. If we use '1' to represent the feature point and '0' to represent the non-feature point, we can obtain a binary sequence. Formally,

let $I : \Omega \subset \mathbb{Z}^1 \rightarrow \{0,1\}$ be a binary sequence where $\Omega = \{1,...,n\}$, unless otherwise stated. Hence we have an object $\mathcal{O}$ including all the feature points:

$$\mathcal{O} = \{q \in \Omega \mid I(q) = 1\}. \quad (1)$$

Formal definition of the 1D closest point problem is given as:

Definition 1. The closest point problem of a sorted point set P in 1D space is the computation of a sequence C whose value at each point p is its closest feature point q from $\mathcal{O}$, where q satisfies:

$$\| coord(p) - coord(q) \| \leq \| coord(p) - coord(\hat{q}) \| \quad for\ all\ \ \hat{q} \in \mathcal{O}. \quad (2)$$

It is clear, the 1D closest point problem can be solved by a sequential algorithm in $O(n)$ time. As shown in Fig.1, all '0's of $input$ sequence represent the non-feature points and all '1's represent the feature points. First we set $V(i)$ to be '∞' if the corresponding point is a non-feature point, otherwise set $V(i)$ to $index(i)$. All '1's divide $input$ sequence into several segments and each segment leading by a '1' and end before next '1'. Each leading '1' corresponds to an index number in $V(i)$. If we process $V(i)$ from left to right and replace all '∞'s of a segment with the leading index number, then sequence $L(i)$ can be obtained. In the same way, sequence $R(i)$ can be obtained by processing $V(i)$ from right to left. By selecting the minimum between $|index(i)-L(i)|$ and $|index(i)-R(i)|$, the final result can be computed, see $output$ sequence.

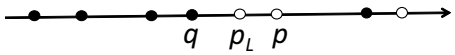
index(i)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
input	0	1	0	0	1	0	0	0	1	0	0	0	0	1	1	0
V(i)	∞	2	∞	∞	5	∞	∞	∞	9	∞	∞	∞	∞	14	15	∞
L(i)	∞	2	2	2	5	5	5	5	9	9	9	9	9	14	15	15
R(i)	2	2	5	5	5	9	9	9	9	14	14	14	14	14	15	∞
output	2	2	2	5	5	5	5	9	9	9	9	14	14	14	15	15

Fig. 1: Illustration of sequential algorithm for computing 1D closest point problem

Observation 1. If a point q in point set P is the feature point, its closest feature point is itself.

Observation 2. For a non-feature point p in point set P, if a feature point q on the left side has been found and there exists no other feature points between p and q, then q is the closest feature point of p on the left side. The closest point on the right side can be found in the same way.

The first observation means that if a point q is the feature point, we have no need to check other points. The second observation implies that, if a point p is the non-feature point, the search of its closest feature point can start from its nearest point, once the closest feature point found, we have no need to check further points. As shown in Fig.2, to find the closest feature point of p on the left side, we first check the nearest point p_L, it is the non-feature point, then check q, and q is the feature point, then stop checking further points.



Black dot : feature point
White dot : non-feature point

Fig. 2: Closest point in 1D space

B. Parallel algorithms on EREW PRAM

In this section we describe two parallel algorithms on EREW PRAM. Before describing the algorithm, we define an associative operator $\multimap$ which is the basic operation in our algorithm.

Definition 2. For two variables a_i and a_j, the associative operator $\multimap$ is defined as follows:

$$a_i \multimap a_j \equiv if \ \ (a_i < \infty \wedge a_j = \infty) \ \ then \ \ a_j \leftarrow a_i \quad (3)$$

Definition 3. Group operation is defined as follows:

$$a_i \multimap (a_k, a_{k+1}, ..., a_{k+j}) \Longrightarrow (a_i \multimap a_k,\ a_i \multimap a_{k+1},\ ...,\ a_i \multimap a_{k+j}) \quad (4)$$

where we call the variables on the left side of operator $\multimap$ as transmitter and the variables on the right side as receiver.

An example of the group operation is shown in Fig.3. The parallelization of the group operation is straightforward.

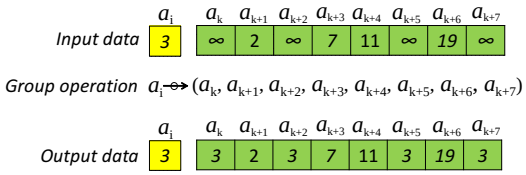


Fig. 3: Illustration of group operation

We describe only the parallel algorithms computing the closest feature point on the left side. The closest feature point on the right side can be computed in the same way. We describe two algorithms on the EREW PRAM, one is time efficient and another is work efficient.

The time efficient parallel algorithm can be described by a tree structure shown in Fig.4. Each node of the tree is a group operation. In the lower level of the tree the receivers receive the information from neighboring transmitters and in the higher level, receive the information from transmitters further away.

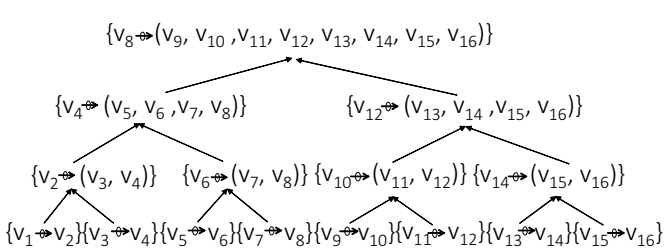


Fig. 4: Computing tree of the time optimal algorithm

Using $n/2$ processors, the proposed parallel algorithm can perform the computation in $O(logn)$ time. However, from the tree, the total work of the algorithm is $O(nlogn)$, and therefore it is not work-optimal. The pseudocode of the algorithm is given in Algorithm 1.

Algorithm 1 Time efficient parallel algorithm

```
1: input[0 : n − 1]: input binary sequence
2: for i ← 0, n − 1 do in parallel
3:     V[i] ← ∞
4:     if input[i] = 1 then
5:         V[i] ← i
6:     end if
7: end for
8: for i ← 2, n by 2^{i+1} do
9:     for j ← 0, n − 1 do in parallel
10:        from ← ⌊j/(i/2)⌋ × i + i/2 − 1
11:        dstn ← ⌊j/(i/2)⌋ × i + i/2 + (j mod (i/2))
12:        if (V[dstn] = ∞) ∧ (V[from] ≠ ∞) then
13:            V[dstn] ← V[from]
14:        end if
15:    end for
16: end for
```

Now we describe the work efficient parallel algorithm which performs $O(n)$ total work on the EREW PRAM. This algorithm is similar to the parallel prefix sum algorithm proposed in [47]. But here we extend it to calculate 1D EDT. It consists of two phases, the reduction phase and the down-sweep phase, each phase can be described by a tree structure, see Fig.5 and 6. The reduction phase traverses the tree from leaves to root computing intermediate results at internal nodes in the tree. The down-sweep phase traverses the tree computing final results from the intermediate results obtained in the reduction phase. To build a balanced binary tree for easy description, we add some empty nodes in the down-sweep tree where no operation is executed. Readers can refer to [47] for more details.

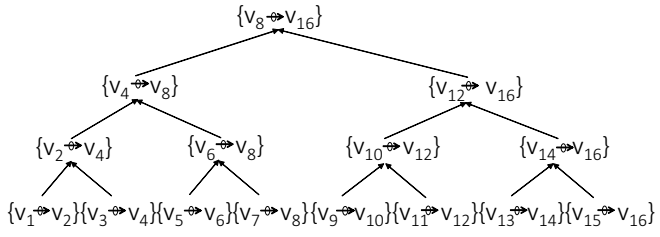


Fig. 5: Computing tree of reduction phase

This algorithm can complete the computation in $O(logn)$ time. From the tree structures, the total work of the algorithm is $O(n)$. Thus it is not only time efficient but also work efficient parallel algorithm. The pseudocode of the algorithm is given in Algorithm 2.

C. Super efficient parallel algorithm for GPU

1) Multi-levels of parallelism on GPU: CUDA (Compute Unified Device Architecture) enabled GPUs expose three

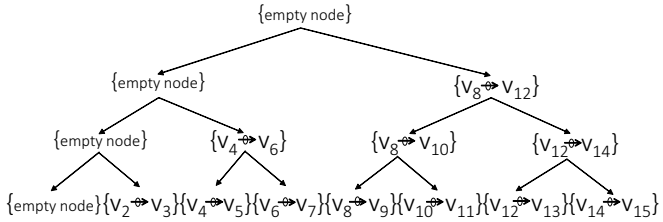


Fig. 6: Computing tree of down-sweep phase

Algorithm 2 Work efficient parallel algorithm

```
1: input[0 : n − 1]: input binary sequence
2: for i ← 0, n − 1 do in parallel
3:     V[i] ← ∞
4:     if input[i] = 1 then
5:         V[i] ← i
6:     end if
7: end for
8:
9: /* The reduction phase of the algorithm */
10: for i ← 0, log2 n − 1 do
11:     for j ← 0, n − 1 by 2^{i+1} do in parallel
12:         if (V[j + 2^{i+1} − 1] = ∞) ∧ (V[j + 2^i − 1] ≠ ∞)
    then
13:             V[j + 2^{i+1} − 1] ← V[j + 2^i − 1]
14:         end if
15:     end for
16: end for
17:
18: /* The down-sweep phase of the algorithm */
19: for i ← log2 n − 2, 0 do
20:     for j ← 0, n − 1 by 2^{i+1} do in parallel
21:         if (V[j + 3 · 2^i − 1] = ∞) ∧ (V[j + 2^{i+1} − 1] ≠ ∞)
    then
22:             V[j + 3 · 2^i − 1] ← V[j + 2^{i+1} − 1]
23:         end if
24:     end for
25: end for
```

levels of parallelism to users [48]. The first level is represented by Warp. The warp is a group of 32 contiguous threads which execute in SIMD fashion. Threads within a warp are referred as Lanes. Each thread of a warp can have its own local memory called Registers. Other threads have no access to these registers. Threads can exchange data within a warp using the Shuffle intrinsic.

The second level is represented by Thread Blocks, each of which can hold up to 1024 threads in modern GPUs. All threads in a block can access on-chip memory called Shared Memory, which allows them to exchange data at L1-cache speed. However, such data exchanges typically require synchronization. CUDA provides a synchronization intrinsic for this purpose. The shared memory consists of several banks and each bank cannot be accessed by multiple threads simultaneously. This Bank Conflict is an important programming issue in practice.

The third level is represented by Grid, which consists of a number of thread blocks. Threads from different blocks can only communicate via an off-chip memory called Global Memory. If threads from the same warp simultaneously access words in the global memory that lie in the same aligned 128-byte segment, the hardware merges these 32 reads or writes into one coalesced memory transaction that is as fast as accessing a single word. There is no explicit grid-wide synchronization provided. Communication through the global memory is supported by a shared L2-cache. However the size of this cache is very limited.

The following built-in variables are used in CUDA programming relating the thread hierarchy of CUDA:

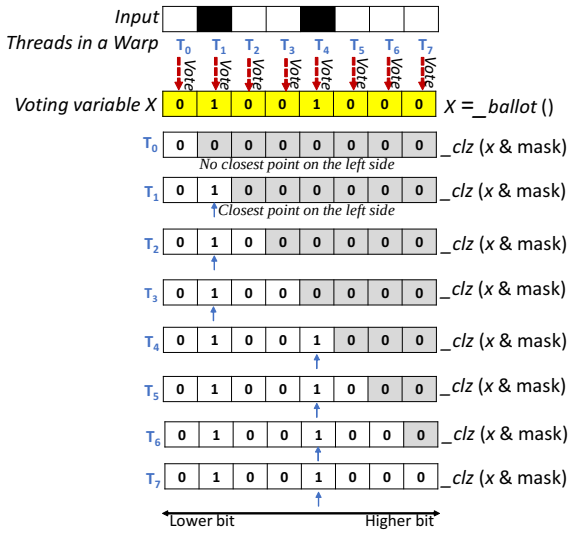
- **warpSize** returns the number of threads in a warp.
- **threadIdx.x, threadIdx.y, threadIdx.z** returns the thread ID in the x-axis, y-axis, and z-axis of the thread which is belonging to a particular CUDA block.
- **blockDim.x, blockDim.y, blockDim.z** returns the number of threads in a CUDA block in the x-axis, y-axis, and z-axis.
- **blockIdx.x, blockIdx.y, blockIdx.z** returns the block ID in the x-axis, y-axis, and z-axis of the block which is belonging to a particular CUDA grid.

2) Super efficient parallel algorithm for GPU: Algorithm 2 demonstrates that the 1D closest point problem can be solved similar to the parallel prefix sum algorithm. As shown in Algorithm 2, each element of the sequence receives information from all elements on the left side (or right side). However, Observation 2 shows that, for the 1D closest point problem, each element has no need to receive information directly from all elements on the left side (or right side). The search of its closest feature point can start from its nearest point, once the closest feature point is found, we have no need to check further points. For the 1D closest point problem, we can design a parallel algorithm simpler than the parallel prefix sum algorithm on the GPUs.

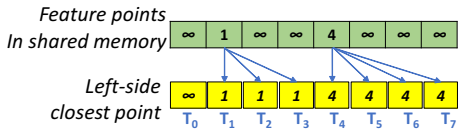
Because of the importance of binary operations [49], CUDA provides some warp-level binary operations for improving GPU's efficiency. These operations allow the threads of a warp to compute cooperatively. The binary operations used in this work are shown as follows. Using these operations we can design a super efficient GPU algorithm for the 1D closest point problem.

- **int __ballot (int p)** returns a 32-bit integer in which bit k is set if and only if the predicate p provided by the thread in lane k of the warp is non-zero.
- **int __clz (int x)** counts Leading Zeros: Returns the number of consecutive zero bits beginning at the most significant bit of the 32-bit integer x.
- **int __ffs (int x)** finds the position of the first (least significant) bit set to 1 in x, where the least significant bit position is 1.
- **int __shfl (int x, int srclane)** copy variable x from the thread indexed by $srclane$.

We use a simple example to explain the warp operations for computing the closest point on the left side, see Fig.7a. For easy description, we assume warpSize is 8 and Integers just



(a) Warp operations for finding the thread which holds the closest site; Gray cells represent masked bits; The blue arrow indicates the result of warpSize-1-clz() on $X \,\&\, mask$, which is the closest point on the left side for that thread.



(b) Communication between threads using the shared memory

Fig. 7: Finding the closest site with $ballot()$ and $clz()$ intrinsic

consist of 8 bits. First, each thread reads the corresponding input point, and votes the corresponding bit of a integer variable accordingly, see Fig.7a (readers can imagine that all threads are now operating on the same variable). Each thread sets the corresponding bit of the integer variable as 1 if the corresponding point is a feature point, otherwise set the bit as 0. The voting procedure can be implemented using the ballot() intrinsic of CUDA. The ballot() intrinsic can pack all voting bits from all threads in a warp into a single 8-bit integer variable and returns this variable to every thread. Now every thread holds a copy of the voting variable. Since the left-side closest point must locate on the left of the current point, the higher (warpSize-lane-1) bits of the variable should be masked with 0. For example, for thread T_3, its lane is 3, thus the higher 4 bits of the variable are masked by 0. Then clz() intrinsic is used to count the leading zeros in the masked value. It is clear the lane of the thread which holds the closest point can be computed by (warpSize-clz()-1). For thread T_3, intrinsic clz() returns 6 leading zeros, therefore its closest point is held by thread T_1. Each thread holding a feature point writes the index of the feature point into the corresponding shared memory place, see Fig.7b, the first row. All threads then read the corresponding shared memory place to obtain the left-side closest point according to the thread lane obtained from previous step. CUDA provides some special intrinsics such as shfl() to support the register-level communication between threads of a warp. Using this intrinsic, we can avoid the use

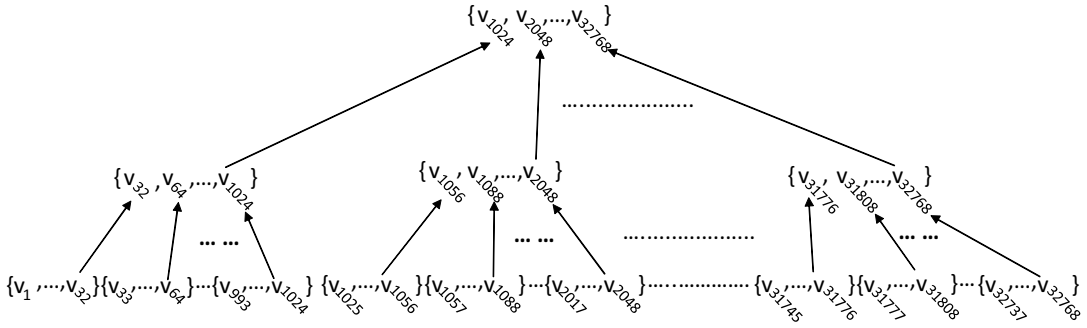


Fig. 8: Computing tree for the reduction phase of the super efficient GPU algorithm

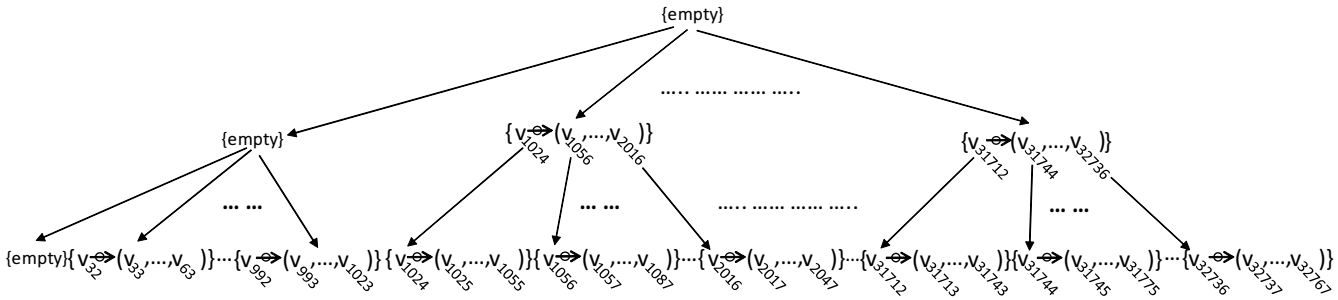


Fig. 9: Computing tree for the down sweep phase of the super efficient GPU algorithm

of shared memory. Access to the shared memory is slower than access to the register memory. Each thread obtains the closest feature point using the shfl() intrinsic which performs communication between the current thread and the thread which holds the closest feature point. The closest feature point on the right side can be computed by masking the lower (lane+1) bits with 0 and using ffs() instead of clz().

According to Observation 2, each position has no need to receive information from all positions individually on the left side (or right side). Therefore the warp operation can be used in each level of the computing tree of the reduction phase, see Fig.8. The computing tree is a balanced multi-branch tree. Each parent node has multiple children, where the number of children is equal to the warp size, i.e. it is 32. If we assume all warps of all CUDA blocks can be executed simultaneously, the depth of the tree should be $log_{32}n$. As described in the previous section, we also add some empty nodes in the down-sweep tree where no operation is executed in the empty nodes, see Fig.9. Initially, the empty node is inserted at the root of the tree. On each level, each child node is generated by applying the group operation $\multimap$ to the values from the reduction phase and the values from the parent node. The size of each group is 32, i.e. the warp size.

Algorithm 3 shows the operations inside a CUDA block, readers can extend it for a CUDA grid. The number of threads each CUDA blocks holds is 1024. The code is for finding the closest feature point on the left side. By reading the input data in the opposite direction or using the ffs() intrinsic instead of clz(), the code can be changed for finding the closest feature point on the right side.

If we assume all warps of all CUDA blocks can be executed at the same time, the time complexity of the algorithm is $O(log_{32}n)$ and the total work is $O(n)$.

The final closest point is computed by selecting the minimum between the left-side closest point and the right-side closest point. The algorithm needs to pass through twice, once for the computation of left-side closest point and another for the right-side closest point.

One-Pass Closest Point We now introduce a modification that allows us to make just one pass. The right-side closest point can be computed from the left-side closest point directly using this approach. Instead of the mask (Fig. 7a) being the higher (warpSize-lane-1) bits we mask the higher (warpSize-lane) bits of the voting variable.

Step One: Figure 10, row *LN* Each thread writes the thread index of its closest left neighbor that is not itself using _clz(X & mask) where mask is the higher (warpSize-lane) bits.

Step Two: Figure 10, row *RN* Each thread checks if it is a black pixel. If it is, get i such that $i = LN[thread]$ or $i = 0$ if $LN[thread] = \infty$. Now write its thread number to $RN[i]$. For example, for thread T_7, its left-side closest point is stored in $LN[7] = 4$. Now T_7 writes its own index 7 to $RN[LN[7]]$ i.e., $RN[4] = 7$. There is at most one thread which is holding a black pixel and its nearest point according to LN is at '∞'. This thread will write its own index to $RN[0]$ (see thread T_2).

Step Three: Figure 10, row *CS* Each thread can now evaluate its closest point to both the left and right. If the pixel is black, its closest point is itself. For white pixels, the closest left point is given by $LN[thread]$. The closest right point is given by $RN[LN[thread]]$ or $RN[0]$ if $LN[thread] = \infty$.

For example, for thread T_8, its left-side closest point is stored in $LN[8]$, and its right-side nearest point is stored in $RN[LN[8]]$. For thread T_1, its left-side closest point is at '∞', and its right-side closest point is stored in $RN[0]$. CS

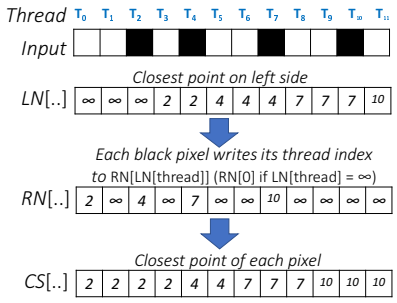


Fig. 10: One pass computation of closest point

is calculated from the decision on which of the left and right points is closest per thread. The algorithm requires only simple instructions per thread and can be independently processed for each thread using the warp instructions.

IV. 2D Euclidean Distance Transform

A. Definition

Given a 2D binary image, its Euclidean distance transform computes the distance of each pixel to the closest black pixel termed *site*. Formally, let $I : \Omega \subset \mathbb{Z}^2 \rightarrow \{0, 1\}$ be a 2D binary image where the domain Ω is convex and, in particular $\Omega = \{1, ..., n\} \times \{1, ..., n\}$, unless otherwise stated. By convention, 0 is associated to white pixel, and 1 to black pixel. Hence we have an object $\mathcal{O}$ including all the black pixels:

$$\mathcal{O} = \{q \in \Omega \mid I(q) = 1\} \tag{5}$$

Formal definition of the 2D EDT is given as:

Definition 4. Euclidean Distance Transform of a 2D binary image is the transformation that generates a map D whose value in each pixel p is the smallest distance from this pixel to $\mathcal{O}$:

$$D(p) := min\{d(p, q) \mid q \in \mathcal{O}\} = min\{d(p, q) \mid I(q) = 1\} \tag{6}$$

where the distance metric is Euclidean distance.

B. Computation of closest site

The closest site of each pixel in the same column can be obtained by 1D EDT algorithm presented above. The results of 1D EDT provide a set of candidate sites to each row of the image. However not all of them are the closest site of the row. As shown in Fig.11, each circle dot represents the closest site (black pixel) of each pixel in the same column. By drawing voronoi diagram (blue line) of these black pixels, the dominant area of each black pixel can be exhibited. It is clear, only c_1, c_4, c_5 and c_7 have a dominant interval on image row j. This provides us a way to compute the closest sites for each row of the image. The same as PBA, three rules are applied for the computation of closest sites.

Algorithm 3 Super efficient GPU algorithm

```
index ← blockIdx.x × blockDim.x + threadIdx.x
x ← ∞                          ▷ All variables are in Register memory
if input[index]=1 then
    x ← index
end if

/********* The reduction phase *********/
voted_x ← __ballot(x< ∞)
masked_x ← mask higher (warpSize - lane - 1) bits of voted_x with 0
count_zeros ← __clz(masked_x)
closest_index ← ∞
if count_zeros < warpSize then
    closest_p ← warpSize - count_zeros - 1
end if
closest_index ← __shfl(x,closest_p)
if lane = warpSize - 1 then
    shared[⌊threadIdx.x/warpSize⌋] ← closest_index
                               ▷ shared[] is in Shared memory
end if
synchronize  threads

if threadIdx.x < warpSize then
    x ← shared[threadIdx.x]
    voted_x ← __ballot(x< ∞)
    masked_x ← mask higher (warpSize - lane - 1) bits of voted_x with 0
    count_zeros ← __clz(masked_x)
    if count_zeros < warpSize then
        closest_p ← warpSize - count_zeros - 1
    end if
    shared[threadIdx.x] ← __shfl(x,closest_p)
end if
synchronize  threads

/********* The down-sweep phase *********/
if ⌊threadIdx.x/warpSize⌋ >0 ∧ closest_index=∞ then
    closest_index ← shared[⌊threadIdx.x/warpSize⌋-1]
end if
output[index] ← closest_index
```

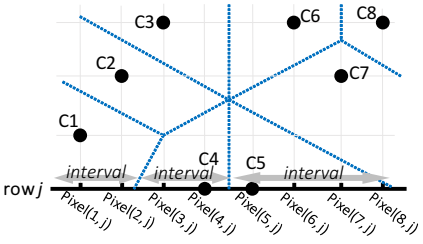
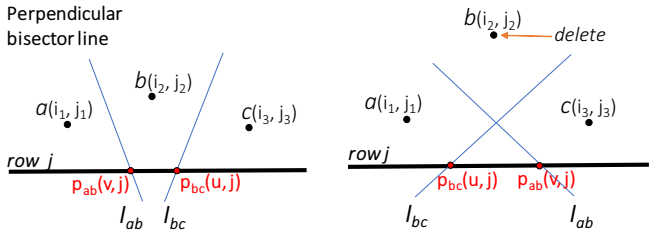
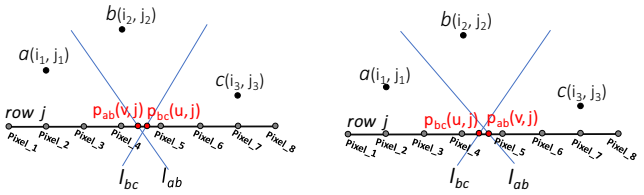


Fig. 11: Voronoi diagram of all black pixels



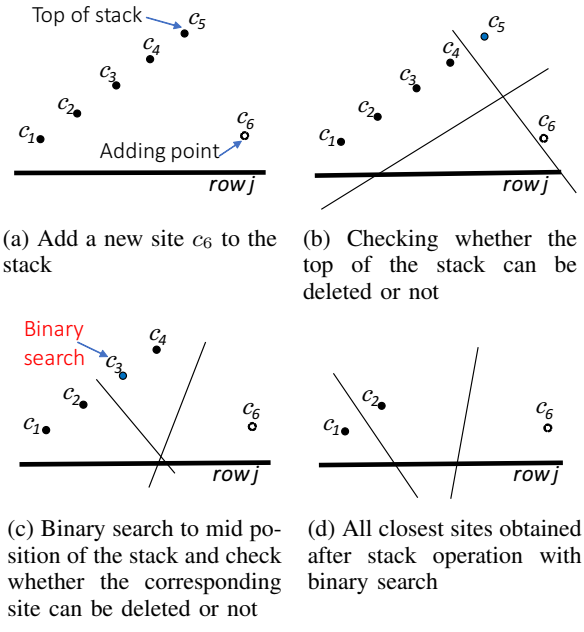
(a) Without deletion of site where $v < u$ (b) With deletion of site where $v > u$

Fig. 12: Determination of closest sites of a row by perpendicular bisector line



(a) Case with $v < u$ (b) Case with $v > u$

Fig. 13: Special cases where both intersection points lay between two successive pixels



(a) Add a new site c_6 to the stack (b) Checking whether the top of the stack can be deleted or not

(c) Binary search to mid position of the stack and check whether the corresponding site can be deleted or not (d) All closest sites obtained after stack operation with binary search

Fig. 14: Deletion of unnecessary sites with binary search on Stack

We first consider a general case. As shown in Fig.12a, there are three sites $a(i_1, j_1), b(i_2, j_2), c(i_3, j_3)$ where $i_1 < i_2 < i_3$. We draw perpendicular bisector line l_{ab} of $a(i_1, j_1)$ and $b(i_2, j_2)$, l_{bc} of $b(i_2, j_2)$ and $c(i_3, j_3)$. The line l_{ab} intersects with the image row j at point $p_{ab}(v, j)$ and the line l_{bc} at $p_{bc}(u, j)$. It is clear if $v < u$ then site $b(i_2, j_2)$ has a dominant interval on the image row. However if $v > u$ then site $b(i_2, j_2)$ has no dominant interval on the image row. We delete the site $b(i_2, j_2)$ from the candidate set.

We remember that now we are computing the closest site of each pixel of a binary image where all pixels are with integer coordinates. If both intersection points $p_{ab}(v, j)$ and $p_{bc}(u, j)$ lay between successive two pixels, see Fig.13, no matter what position relation both intersection points hold, the site $b(i_2, j_2)$ should be deleted from the candidate set, even $v < u$, since there is no pixel where its closest site is $b(i_2, j_2)$.

Rule 1. Given an image row j and three candidate sites $a(i_1, j_1), b(i_2, j_2), c(i_3, j_3)$ where $i_1 < i_2 < i_3$. The perpendicular bisector line l_{ab} of $a(i_1, j_1)$ and $b(i_2, j_2)$ intersects with the image row j at point $p_{ab}(v, j)$ and the perpendicular bisector line l_{bc} of $b(i_2, j_2)$ and $c(i_3, j_3)$ at point $p_{bc}(u, j)$. If $\lceil v \rceil \geq \lceil u \rceil$ then site $b(i_2, j_2)$ has no dominant interval on the image row and it can be deleted from the candidate set.

All closest sites of a row of the image can be obtained by stack operations. The push and pop operations of the stack is based on Rule 1.

Let $p(v, j), q(u, j)$ be two pixels of image row j where $v < u$. If site $c(i_1, j_1)$ and $c'(i_3, j_3)$ are the closest site of p and q respectively, then we have $i_1 \leq i_3$. The order of all dominant intervals is exactly same with the order in which the corresponding closest sites appear. Based on this, a binary search operation can be used to reduce the amount of stack operations. As shown in Fig.14a, there 5 sites $c_1, ..., c_5$ already in the stack and now we want to add c_6 to the stack. We first check the top of the stack, that is c_5, whether it can be deleted from the stack or not, see Fig.14b. If c_5 can be popped from the stack, then the forward binary search is used to find the next site c_3, and check it, see Fig.14c. If c_3 can be popped from the stack, then the forward binary search is applied again to find the next site to check (it means all candidate sites between c_3 and c_5 can be popped from the stack without checking). Otherwise the backward binary search is applied to find the next site to check. The combination of the binary search and stack operations is applied until there is no site can be popped further, see Fig.14d.

Rule 2. Assume there k sites $c_1, c_2, ..., c_k$ already exist in the stack and now we are adding a new site c_h to the stack. If the top of the stack, c_k, can be popped from the stack, the next site to check is $c_{\lfloor k/2 \rfloor + 1}$ whose index can be obtained by the binary search.

However, the performance of stack operations with the binary search depends on the number of closest sites among all candidate sites. If the number of closest sites among all candidate sites is small, then the stack operations with the binary search can achieve a high performance. Otherwise, the performance of stack operations will decrease significantly because a large number of binary searches will be performed.

Now we consider the merge of two independent stacks one named S_1 and another named S_2. The merge of two stacks is an important operation in the second step of our parallel algorithm. We add sites from the bottom of S_2 (we call it the sending stack) to the top of S_1 (we call it the receiving stack). It is clear, during merging procedure, once there are two sites on the top of S_1 that are from S_2, and neither of these two sites can be popped from S_1 further, then we can add the rest of S_2

to S_1 without checking. However, in practice, data movement is with heavy cost. Therefore, to avoid moving data, a linked list can be used to connect two stacks to create a new larger stack, as proposed in [17]. Then the rest of S_2 has no need to be moved to S_1.

Rule 3. We merge two independent stacks S_1 and S_2 by adding sites from the bottom of S_2 to the top of S_1. If there are two sites from S_2 in the top of S_1, and neither of these two sites can be popped from S_1 further, then we can add the rest of S_2 to S_1 without checking.

C. Parallel algorithm for 2D Euclidean Distance Transform

We describe a fully-parallelized work-optimal 2D EDT algorithm in this subsection.

Step 1: Compute the closest site of each pixel in the same column using the 1D EDT algorithm. The results of 1D EDT will be stored in a 2D array named R_{1D}.

Step 2: Compute closest sites for each row of the image using the stack operation.

- **Step 2.1:** The creation of the stacks. Partition each row of R_{1D} into $n/3$ groups each with 3 elements. Each processor performs the stack operations to the corresponding group following Rule 1.
- **Step 2.2:** The merge of the stacks. Merge every two stacks using a processor following Rule 2 and 3. In the merging procedure, we add sites from the bottom of one stack to the top of the another. Repeat the merging procedure until one stack left.

Step 3: For each pixel, find the closest site and compute the distance between them.

- **Step 3.1:** Find the dominant interval for each closest site. If we draw the perpendicular bisector line between each pair of closest sites, this line may intersect with the image row at a point. It is clear, each pair of such intersection points decides a dominant interval for a closest site, see Fig.15. We allocate a 1D array of size n where n is the row size of input image, and initialize each element of the 1D array with tuple (∞, ∞). Assume the perpendicular bisector line of two sites $c_l(i_l, j_l)$ and $c_r(i_r, j_r)$ intersects with the image row j at a point $p(v, j)$. We then fill $\lfloor v \rfloor^{th}$ element of the 1D array with (i_l, j_l) which is the coordinate of site c_l, see Fig.15. (Rule 1 guarantees that at most one intersection point appears between two successive pixels.)
- **Step 3.2:** Compute the distance of each pixel to the closest site. The closest site can be found by solving the right-side closest point problem on the 1D array.

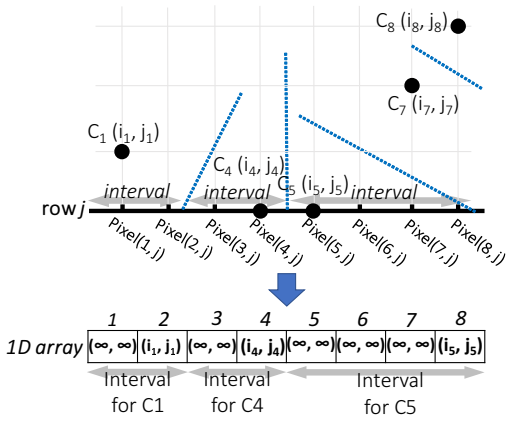


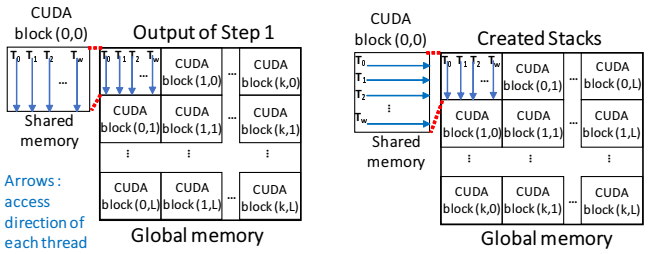
Fig. 15: Presentation of each interval on 1D array

D. GPU implementation

Considering the coalesced access of the global memory of the GPU, Step 1 is implemented along each row of the input image. The *One-pass algorithm* described in Section III-C2 is employed. If the shared memory is capable of storing the whole image row, then the global memory just needs to be accessed twice, one for read and another for write.

In Step 2, the output of Step 1 is partitioned into many tiles and a CUDA block is assigned to a tile, see Fig.16a. Each CUDA block reads the global memory along column and creates the stacks in the shared memory. It is clear the access to the global memory is coalesced. The creation of stacks in the shared memory is also without bank conflicts. Each CUDA block writes the created stacks back to the global memory in a transposed location, see Fig.16b. For example, CUDA block$\{0, L\}$ reads the left-bottom corner of the input array, see Fig.16a, and writes the created stacks back to the right-top corner of the output array, see Fig.16b. For this specific step, the read of the shared memory has a bank conflict. However, it still can achieve a better performance, since an explicit matrix transpose [30] is avoided. The binary search on each stack is not applied. Since experiments show that, in most cases, the use of binary search will lead to a poor performance. To make a trade-off between the creation cost and the merging cost of stacks, the size of initial stack should be larger than 3. The merge of stacks are implemented in global memory directly. For each stack, we use two pointers, one indicates the bottom of the stack and another the top of the stack. During the stack merging procedure, we just move these two pointers. The merge of two stacks terminates once two sites are added to the receiving stack, and are not popped from the receiving stack further, see Rule 3. This avoids the addition of the rest of the sending stack to the receiving stack, as the GPU is sensitive to memory operations.

In Step 3, one thread warp is assigned to one stack to compute the perpendicular bisector lines, where the stack is indicated by a pair of pointers (remember that the final stack consists of many small stacks each of them is indicated by a pair of pointers, here we assign a thread warp to a small stack). The 1D array mentioned in Step 3.1 is allocated in



(a) Threads create stacks using the shared memory

(b) Threads write stacks back to the global memory in a transposed location

Fig. 16: Creation of stacks using the shared memory

the shared memory. Finally we transpose the results of Step 3 using the shared memory following the algorithm of [50].

Now we give a brief discussion on how to implement the algorithm outside CUDA, for example in OpenCL (Open Computing Language) [51] or Vulkan [52]. The main programming issues would be in the GPU implementation of Step 1. OpenCL provides equivalent functions of clz() and shfl() of CUDA, but the current version does not provide an equivalent function to ballot(). User can emulate ballot() using the atomic_or($operand1$, $operand2$) function of OpenCL, which performs bit-wise or-operation on two operands $operand1$ and $operand2$ and stores the result in $operand1$. These two operands are unsigned integers in our work. The $operand1$ should be in Local Memory of OpenCL (which is equivalent to the shared memory of CUDA). For each thread, the corresponding bit of $operand2$ is set to 1 if the current thread holds a black pixel, otherwise 0. Other bits of $operand2$ are set to 0. Since each thread needs to wait for all other threads to finish the atomic operation, there should be a synchronization between atomic_or() and clz(). Vulkan is a graphics and compute API that provides low-overhead, cross-platform access to modern GPUs. Latest version of Vulkan is integrated with SPIR-V (Standard Portable Intermediate Representation) [53] which is an intermediate language for parallel compute and graphics by Khronos Group. SPIR-V (version 1.3) provides a corresponding function to ballot(). SPIR-V Extended Instructions for GLSL (OpenGL Shading Language) [54] provides a function FindUMsb() which finds the most significant bit in a unsigned integer. SPIR-V also provides an equivalent of the shfl() function.

E. Complexity analysis

We now analyze the complexity of the GPU implementation of the algorithm. We assume all CUDA warps of all CUDA blocks can be executed simultaneously.

Fact 1. Step 1 takes $O(log_{32}n)$ time and $O(N)$ of total work on GPU.

Proof. The height of the computing tree shown in Fig.8 and 9 is $log_{32}n$, it yields the time complexity claimed. In each level of the computing tree, the number of threads computing is $n/32^m$ where m $(0 \leq m \leq log_{32}n - 1)$ is the height of current level of the tree. Therefore the total work of Step 1 is $O(N)(N = n^2)$. □

Fact 2. The expected time complexity of Step 2 is $O(logn)$. The total work Step 2 performs is $O(N)$.

Proof. The height of the merging tree is $O(logn)$. If the distribution of closest sites among candidate sites allows all processors to perform the same amount of work, then Step 2 will run in $O(logn)$ time. It yields the time complexity claimed.

For an image row, suppose there are $k(k \leq n)$ candidate sites obtained and $k_{non}(k_{non} < k)$ sites among them are not the closest sites of the image row. These k_{non} sites will be popped from the stacks on each level of the merging tree. Suppose the number of sites which will be popped from the stacks on ith $(0 \leq i \leq logn - 1)$ level of the merging tree is k^i_{non}, we have

$$k_{non} = k^0_{non} + k^1_{non} + ... + k^i_{non} + ... + k^{logn-1}_{non}.$$

When we add (push operation) one site, which is from the sending stack, to the receiving stack, it may cause one or more pop operations on the receiving stack. On the other hand, we remember that the merge of two stacks will terminate once there are two sites in the receiving stack that are from the sending stack and they cannot be popped from the receiving stack further. It means that, if k^i_{non} pop operations are executed in ith level of the merging tree, the number of push operations executed is at most k^i_{non}+$2 \times n/2^i$, where $n/2^i$ is the number of threads computing in ith level of the merging tree. These threads will add (push operation) at least two sites into the receiving stack. Therefore the total number of push operations in all levels of the merging tree is:

$$\begin{aligned}
&(k^0_{non} + 2 \times n/2^0) + (k^1_{non} + 2 \times n/2^1) + ...+ \\
&(k^i_{non} + 2 \times n/2^i) + ... + (k^{logn-1}_{non} + 2 \times n/2^{logn-1}) = \\
&(k^0_{non} + k^1_{non} + ... + k^i_{non} + ... + k^{logn-1}_{non})+ \\
&2 \times n \times (1/2^0 + 1/2^1 + ... + 1/2^i + ... + 1/2^{logn-1}) \leq \\
&k_{non} + 4 \times n \leq \\
&5 \times n = \\
&O(n)
\end{aligned}$$

□

Fact 3. Step 3 takes $O(log_{32}n)$ time and $O(N)$ of total work on GPU.

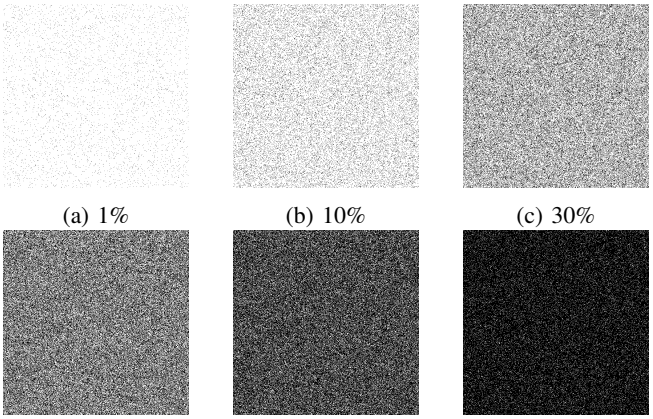
Proof. The intersection points between the image row and the perpendicular bisector lines of each pair of closest sites can be computed in $O(1)$ time and $O(n)$ of total work on GPU. The right-side closest point problem can be solved in $O(log_{32}n)$ time and $O(n)$ work on GPU (as proved in Fact 1). It yields the complexity claimed. □

F. Evaluation

The evaluation is performed on a machine with an Intel Core i7-6700 CPU (3.40GHz) and a NVIDIA GeForce GTX1080 graphics card with 8GB of video RAM. The programming

environment is Visual Studio 2015 with CUDA 8.0 running on Windows 10.

1) Comparison with PBA step by step: Both PBA and our algorithm consist of the same steps. We compare two algorithms step by step varying the density of sites. Examples of input images are shown in Fig.17. We use the original source code of PBA which is available online at PBA's project web site for comparison.



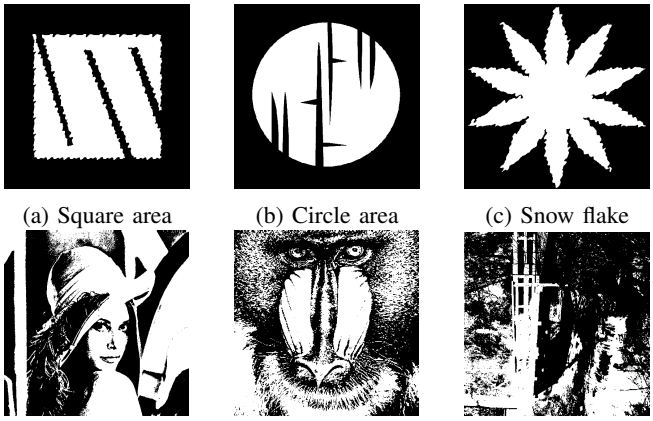
(a) 1% (b) 10% (c) 30%

(d) 50% (e) 70% (f) 90%

Fig. 17: Input images with different densities of black pixels

Table II shows the running time of each step of two algorithms. The size of input image is $8k \times 8k$. As shown in the table, the speedup factor of Step 1 is always more than 2, since the time complexity of Step 1 of our algorithm is only $O(log_{32}n)$. Also, Step 1 of our algorithm is implemented by the one-pass approach which just needs to go through the data once, see Section III-C2 and Section IV-D. To access the global memory with coalescing, a matrix transpose is needed in PBA following Step 1. However, our algorithm has no need to perform this transpose since the transpose is hidden in Step 2, see Section IV-D. There is no significant difference between two algorithms in Step 2, our algorithm is slightly faster than PBA. The performance of PBA is dominated by Step 3 and the main difference between two algorithms also appeared in this step. Since, the PBA needs $O(n)$ time in this step, versus $O(log_{32}n)$ time of our algorithm. Finally, both algorithms perform the matrix transpose again to get correct results.

2) Comparison with PBA varying geometric structure of input image: Both PBA and our algorithm are very sensitive to the content of the input images. Therefore we use images with different geometric structure to test them. Images used in experiments are shown in Fig.18. The first three images contain different geometric shapes. These three images are chosen since geometrical characters of the input image may impact the performance of algorithms. The last three images contain real objects. We choose these three images since which have been universally used as impartial benchmarks for image analysis algorithms. The size of the images is $1k \times 1k$. Our algorithm always outperforms the PBA, see Fig.19, no matter how complex the input images are.



(a) Square area (b) Circle area (c) Snow flake

(d) Lena (e) Mandrill (f) Walk bridge

Fig. 18: Input images with different geometric structure

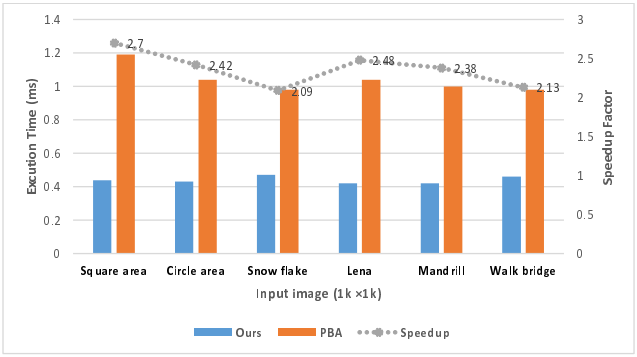


Fig. 19: Performance comparison between PBA and our algorithm with images containing different geometric structure

3) Comparison with other algorithms: In the absence of source code availability from Honda [18], we compare our work with JAF, PBA and Honda's algorithm using the data presented in [18] which was created by testing the various approaches on the same GPU (NVIDIA GeForce GTX 1080). The following four images are used in the experiments: an image with 0.01% of black pixels (the location of black pixels is chosen randomly), an image with 1% of black pixels, an image with 50% of black pixels and the image of Lena. The size of image varies from 512×512 to $16k \times 16k$. As shown in Table III, for smaller images, the JFA can achieve a better performance comparing with PBA and Honda's algorithm. The PBA is faster than JFA and Honda's algorithm for the images with the size smaller than $8k \times 8k$. Honda's algorithm can achieve a better performance for the images with size larger than or equal to $8k \times 8k$. Our algorithm outperforms all three algorithms in every case, see Table III. The maximum speedup factor is 4.67, 2.86 and 6.64 compared with JAF, PBA and Honda's algorithm, respectively.

V. Conclusion

In this paper, we present a fully-parallelized work-time optimal exact EDT algorithm. Compared to the existing PRAM algorithms and other algorithms, this algorithm is suitable for implementation on modern SIMD architectures such as GPUs.

TABLE II: Running time (milliseconds) of each step of two algorithms varying the density of sites

Density of sites	Algorithm	Step 1	Transpose	Step 2	Step 3	Transpose	Total	Speedup
1%	PBA	7.86	3.41	11.22	12.38	3.51	38.38	1.84
	Ours	3.20	–	10.67	3.40	3.54	20.81	
10%	PBA	7.78	3.56	14.92	14.93	3.55	44.74	2.10
	Ours	3.39	–	11.21	3.21	3.49	21.30	
30%	PBA	7.98	3.44	14.02	15.9	3.47	44.81	1.95
	Ours	3.39	–	12.54	3.49	3.53	22.95	
50%	PBA	8.03	3.4	13.24	16.4	3.40	44.47	1.91
	Ours	3.37	–	12.72	3.55	3.54	23.18	
70%	PBA	7.85	3.53	12.59	18.34	3.46	45.77	1.99
	Ours	3.35	–	12.39	3.75	3.48	22.97	
90%	PBA	7.81	3.45	12.37	16.51	3.47	43.61	1.94
	Ours	3.32	–	11.92	3.62	3.51	22.37	

TABLE III: Running time (milliseconds) of different algorithms with different input images

Input image	Algorithm	512×512	$1k \times 1k$	$2k \times 2k$	$4k \times 4k$	$8k \times 8k$	$12k \times 12k$	$16k \times 16k$
0.01% random	JFA	0.2114	0.9257	3.967	18.00	81.48	193.6	364.8
	PBA	0.1275	0.3737	1.396	5.904	25.7	58.61	106.5
	Honda's	0.66	1.233	2.9	7.567	23.83	55.16	95.25
	Ours	**0.0987**	**0.2914**	**1.071**	**4.432**	**19.26**	**45.97**	**78.02**
1% random	JFA	0.2117	0.9366	3.966	18.00	81.49	194.8	364.9
	PBA	0.2072	0.5816	2.006	7.644	33.1	74.33	133.6
	Honda's	0.6758	1.276	2.965	7.84	24.9	57.24	99.75
	Ours	**0.1002**	**0.399**	**1.304**	**5.327**	**20.81**	**47.36**	**81.42**
50% random	JFA	0.2095	0.9719	3.977	18.15	82.00	197.3	367.1
	PBA	0.2714	0.8552	2.70	10.62	43.37	94.79	170.1
	Honda's	0.6517	1.343	3.12	8.446	27.32	62.81	109.9
	Ours	**0.1035**	**0.4658**	**1.541**	**6.162**	**23.18**	**51.74**	**89.89**
Lena	JFA	0.1936	0.8169	3.168	13.53	58.23	137.2	252.9
	PBA	0.3059	0.8473	2.40	8.523	33.86	75.92	134.5
	Honda's	0.7081	1.463	3.171	8.028	24.73	57.26	94.5
	Ours	**0.1066**	**0.423**	**1.435**	**5.671**	**22.05**	**48.21**	**86.01**

As a fundamental operation, 1D EDM is efficiently parallelized first. Three different algorithms are proposed for the 1D EDT on the EREW PRAM and GPU. Specifically, the GPU algorithm for the 1D EDT, which uses CUDA binary functions such as ballot(), ffs(), clz() and shfl(), runs in $O(log_{32}n)$ time and performs $O(n)$ work. Using the 1D EDT as a fundamental operation, we propose a fully-parallelized work-time optimal 2D EDT algorithm. This algorithm consists of three steps. The first step of the algorithm runs in $O(log_{32}n)$ time and performs $O(N)$ of total work on GPU. The second step performs $O(N)$ of total work and has an expected time complexity of $O(logn)$ on GPU. The third step runs in $O(log_{32}n)$ time and performs $O(N)$ of total work on GPU. Clearly, this algorithm is a work-time optimal algorithm. As far as we know, this algorithm is the first fully-parallelized work-time optimal algorithm for the GPUs. Experiments show that this algorithm outperforms most of the state-of-the-art GPU algorithms. Currently, this algorithm is the fastest 2D EDT algorithm with the lowest complexity on GPUs.

REFERENCES

[1] C. Arcelli, G. S. di Baja, and L. Serino, "Distance-driven skeletonization in voxel images," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 4, pp. 709–720, April 2011.

[2] S. Loncaric, "A survey of shape analysis techniques," *Pattern Recognition*, vol. 31, no. 8, pp. 983 – 1001, 1998.

[3] L. Lam, S. W. Lee, and C. Y. Suen, "Thinning methodologies-a comprehensive survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 9, pp. 869–885, Sep 1992.

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

[5] R. Fabbri, L. da Fontoura Costa, J. C. Torelli, and O. M. Bruno, "2d Euclidean distance transform algorithms: A comparative survey," *ACM Comput. Surv.*, vol. 40, pp. 2:1–2:44, 2008.

[6] M. W. Jones, J. A. Baerentzen, and M. Sramek, "3d distance fields: a survey of techniques and applications," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 4, pp. 581–599, July 2006.

[7] T. Hayashi, K. Nakano, and S. Olariu, "Work-time optimal k-merge algorithms on the PRAM," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 3, pp. 275–282, Mar. 1998.

[8] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sep. 2011.

[9] B. Bramas, "Fast sorting algorithms using AVX-512 on intel knights landing," *CoRR*, vol. abs/1704.08579, 2017.

[10] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 72–86, Jan. 2017.

[11] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 105–118, Jan. 2011.

[12] K. E. Hoff, III, J. Keyser, M. Lin, D. Manocha, and T. Culver, "Fast computation of generalized voronoi diagrams using graphics hardware," in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 277–286.

[13] A. Sud, N. Govindaraju, R. Gayle, and D. Manocha, "Interactive 3D distance field computation using linear factorization," in *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, ser. I3D '06. New York, NY, USA: ACM, 2006, pp. 117–124.

[14] G. Rong and T.-S. Tan, "Jump flooding in GPU with applications to voronoi diagram and distance transform," in *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, ser. I3D '06. New York, NY, USA: ACM, 2006, pp. 109–116.

[15] J. Schneider, M. Kraus, and R. Westermann, "GPU-based real-time discrete Euclidean distance transforms with precise error bounds," in *VISAPP (1)*, 2009, pp. 435–442.

[16] F. d. A. Zampirolli and L. Filipe, "A fast CUDA-based implementation for the Euclidean distance transform," in *2017 International Conference on High Performance Computing Simulation (HPCS)*, July 2017, pp. 815–818.

[17] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan, "Parallel banding algorithm to compute exact distance transform with the GPU," in *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '10. New York, NY, USA: ACM, 2010, pp. 83–90.

[18] T. Honda, S. Yamamoto, H. Honda, K. Nakano, and Y. Ito, "Simple and fast parallel algorithms for the voronoi map and the Euclidean distance map, with GPU implementations," in *2017 46th International Conference on Parallel Processing (ICPP)*, Aug 2017, pp. 362–371.

[19] T. Saito and J.-I. Toriwaki, "New algorithms for Euclidean distance transformation of an n-dimensional digitized picture with applications," *Pattern Recognition*, vol. 27, no. 11, pp. 1551 – 1565, 1994.

[20] J. C. TORELLI, R. FABBRI, G. TRAVIESO, and O. M. BRUNO, "A high performance 3d exact Euclidean distance transform algorithm for distributed computing," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 24, no. 06, pp. 897–915, 2010.

[21] A. Meijster, J. B. Roerdink, and W. H. Hesselink, "A general algorithm for computing distance transforms in linear time," in *Mathematical Morphology and its applications to image and signal processing*. Springer, 2002, pp. 331–340.

[22] C. R. Maurer, R. Qi, and V. Raghavan, "A linear time algorithm for computing exact Euclidean distance transforms of binary images in arbitrary dimensions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 2, pp. 265–270, Feb 2003.

[23] J. Wang and Y. Tan, "Efficient Euclidean distance transform using perpendicular bisector segmentation," *CVPR 2011*, pp. 1625–1632, 2011.

[24] ——, "Efficient Euclidean distance transform algorithm of binary images in arbitrary dimensions," *Pattern Recognition*, vol. 46, no. 1, pp. 230 – 242, 2013.

[25] R. A. Lotufo and F. A. Zampirolli, "Fast multidimensional parallel Euclidean distance transform based on mathematical morphology," in *Proceedings XIV Brazilian Symposium on Computer Graphics and Image Processing*, Oct 2001, pp. 100–105.

[26] Z. Yuan, G. Rong, X. Guo, and W. Wang, "Generalized voronoi diagram computation on GPU," in *2011 Eighth International Symposium on Voronoi Diagrams in Science and Engineering*, June 2011, pp. 75–82.

[27] G. Rong, Y. Liu, W. Wang, X. Yin, D. Gu, and X. Guo, "GPU-assisted computation of centroidal voronoi tessellation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 3, pp. 345–356, March 2011.

[28] J. Schneider, M. Kraus, and R. Westermann, "GPU-based Euclidean distance transforms and their application to volume rendering," in *selected papers of VISIGRAPP 2009, Communications in Computer and Information Science (CCIS) 68*. Springer-Verlag Berlin Heidelberg, 2010, pp. 215–228.

[29] F. de Assis Zampirolli and L. Filipe, "Distance transform separable by mathematical morphology in GPU," in *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, J. Ruiz-Shulcloper and G. Sanniti di Baja, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 41–48.

[30] D. Man, K. Uda, Y. Ito, and K. Nakano, "Accelerating computation of Euclidean distance map using the GPU with efficient memory access," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 28, no. 5, pp. 383–406, 2013.

[31] Y.-S. Leung, X. Wang, Y. He, Y.-J. Liu, and C. C. L. Wang, "A unified framework for isotropic meshing based on narrow-band Euclidean distance transformation," *Computational Visual Media*, vol. 1, pp. 239–251, 2015.

[32] A. Datta and S. Soundaralakshmi, "Constant-time algorithm for the Euclidean distance transform on reconfigurable meshes," *Journal of Parallel and Distributed Computing*, vol. 61, no. 10, pp. 1439 – 1455, 2001.

[33] 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.

[34] S. PAVEL and S. G. AKL, "Efficient algorithms for the Euclidean distance transform," *Parallel Processing Letters*, vol. 05, no. 02, pp. 205–212, 1995.

[35] A. Fujiwara, T. Masuzawa, and H. Fujiwara, "An optimal parallel algorithm for the Euclidean distance maps of 2-d binary images," *Inf. Process. Lett.*, vol. 54, no. 5, pp. 295–300, Jun. 1995.

[36] M. C. F. Macedo and A. L. Apolinário, Jr., "Euclidean

distance transform shadow mapping," in *Proceedings of the 43rd Graphics Interface Conference*, ser. GI '17. School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada: Canadian Human-Computer Communications Society, 2017, pp. 171–180.

[37] G. Borgefors, "Distance transformations in arbitrary dimensions," *Computer Vision, Graphics, and Image Processing*, vol. 27, no. 3, pp. 321 – 345, 1984.

[38] P.-E. Danielsson, "Euclidean distance mapping," *Computer Graphics and Image Processing*, vol. 14, no. 3, pp. 227 – 248, 1980.

[39] T. Q. Pham, "Parallel implementation of geodesic distance transform with application in superpixel segmentation," in *2013 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, Nov 2013, pp. 1–8.

[40] F. Leymarie and M. Levine, "Fast raster scan distance propagation on the discrete rectangular lattice," *CVGIP: Image Understanding*, vol. 55, no. 1, pp. 84 – 94, 1992.

[41] I. Ragnemalm, "Neighborhoods for distance transformations using ordered propagation," *CVGIP: Image Understanding*, vol. 56, no. 3, pp. 399 – 409, 1992.

[42] Y. Ge and J. M. Fitzpatrick, "On the generation of skeletons from discrete Euclidean distance maps," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 18, pp. 1055–1066, 1996.

[43] R. Satherley and M. W. Jones, "Vector-city vector distance transform," *Computer Vision and Image Understanding*, vol. 82, pp. 238–254, 2001.

[44] H. Yamada, "Complete Euclidean distance transformation by parallel operation," in *Proc. 7th International Conference on Pattern Recognition*, Montreal, Canada, 1984, pp. 33–338.

[45] O. Cuisenaire and B. Macq, "Fast Euclidean morphological operators using local distance transformation by propagation, and applications," in *Image Processing and Its Applications, 1999. Seventh International Conference on (Conf. Publ. No. 465)*, vol. 2, 1999, pp. 856–860 vol.2.

[46] R. Satherley and M. W. Jones, "Hybrid distance field computation," in *Volume Graphics 2001*, ser. Eurographics, K. Mueller and A. Kaufman, Eds. Springer Vienna, 2001, pp. 195–209.

[47] G. E. Blelloch, "Prefix sums and their applications," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-90-190, Nov. 1990.

[48] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[49] N. Cooperation, "CUDA C programming guide," 2017. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide

[50] G. Ruetsch and P. Micikevicius, "Optimizing matrix transpose in CUDA," NVIDIA, Tech. Rep., 2010.

[51] Khronos OpenCL Working Group, "The OpenCL™ specification version v2.2-10," 2019. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf

[52] Khronos Vulkan Working Group, "Vulkan 1.1.107 - A Specification (with all registered Vulkan extensions)," 2019. [Online]. Available: https://www.khronos.org/registry/vulkan/specs/1.1-extensions/pdf/vkspec.pdf

[53] Khronos Group, "SPIR-V Specification version 1.3," 2019. [Online]. Available: https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf

[54] ——, "SPIR-V Extended Instructions for GLSL," 2018. [Online]. Available: https://www.khronos.org/registry/spir-v/specs/unified1/GLSL.std.450.pdf

Manduhu received the MS and PhD degrees in information engineering from Hiroshima University, Japan, in 2010 and 2013, respectively. He is currently a Research Fellow in Swansea University, UK. His research interests include parallel algorithm and image processing.

Mark W. Jones received the B.Sc. and Ph.D. degrees from Swansea University. He is a Professor in the Department of Computer Science at Swansea University, where he leads the Visual Computing Research group. His research interests include global illumination, visualisation, data science, and associated algorithms and data structures.