International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 1, Number 2, pages 260–276, July 2011

Implementations of a Parallel Algorithm for Computing Euclidean Distance Map in Multicore Processors and GPUs

Duhu Man, Kenji Uda, Hironobu Ueyama, Yasuaki Ito, and Koji Nakano
Department of Information Engineering,
Hiroshima University
1-4-1 Kagamiyama, Higashi-Hiroshima, Hiroshima, 739–8527, Japan

Received: January 31, 2011
Revised: May 20, 2011
Accepted: June 20, 2011
Communicated by Sayaka Kamei

Abstract

Given a 2-D binary image of size $n \times n$, Euclidean Distance Map (EDM) is a 2-D array of the same size such that each element is storing the Euclidean distance to the nearest black pixel. It is known that a sequential algorithm can compute the EDM in $O(n^2)$ and thus this algorithm is optimal. Also, work-time optimal parallel algorithms for shared memory model have been presented. However, the presented parallel algorithms are too complicated to implement in existing shared memory parallel machines. The main contribution of this paper is to develop a simple parallel algorithm for the EDM and implement it in two different parallel platforms: multicore processors and Graphics Processing Units (GPUs). We have implemented our parallel algorithm in a Linux server with four Intel hexad-core processors (Intel Xeon X7460 2.66GHz). We have also implemented it in the following two modern GPU systems, Tesla C1060 and GTX 480, respectively. The experimental results have shown that, for an input binary image with size of 9216×9216, our implementation in the multicore system achieves a speedup factor of 18 over the performance of a sequential algorithm using a single processor in the same system. Meanwhile, for the same input binary image, our implementation on the GPU achieves a speedup factor of 26 over the sequential algorithm implementation.

Keywords: Euclidean Distance Map, Proximate Points, Multicore Processors, GPUs

1 Introduction

In many applications of image processing such as blurring effects, skeletonizing and matching, it is essential to measure distances between featured pixels and nonfeatured pixels. For a 2-D binary image with size of $n \times n$, treating black pixels as featured pixels, Euclidean Distance Map (EDM) assigns each pixel with the distance to the nearest black pixel using Euclidean distance as underlying distance metric. We refer reader to Figure 1 for an illustration of Euclidean Distance Map. Assuming that points p and q of the plane are represented by their Cartesian coordinates $(x(p), y(p))$ and $(x(q), y(q))$, as usual, we denote the Euclidean distance between the points p and q by $d(p, q) = \sqrt{(x(p) - x(q))^2 + (y(p) - y(q))^2}$.

Many algorithms for computing EDM have been proposed in the past. Breu *et al.* [1] and Chen *et al.* [2, 3] have presented $O(n^2)$-time sequential algorithm for computing Euclidean Distance Map.

$\sqrt{10}$	3	$\sqrt{10}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	3	$\sqrt{10}$
$\sqrt{5}$	2	$\sqrt{5}$	2	1	0	1	2	2	$\sqrt{5}$
$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{2}$	1	$\sqrt{2}$
1	0	1	2	2	2	2	1	0	1
$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$
$\sqrt{5}$	2	2	1	0	1	2	$\sqrt{5}$	2	$\sqrt{5}$
$\sqrt{10}$	$\sqrt{8}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	3	3	$\sqrt{10}$
$\sqrt{10}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	$\sqrt{5}$	2	$\sqrt{5}$	$\sqrt{8}$
3	2	1	0	1	2	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$
$\sqrt{10}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	2	1	0	1	2

Figure 1: Illustrating Euclidean Distance Map

Since all pixels must be read at least once, these sequential algorithms with time complexity of $O(n^2)$ is optimal. Since in any EDM algorithm, each of the n^2 pixels has to be scanned at least once. Roughly at the same time, Hirata [7] presented a simpler $O(n^2)$-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 EDM algorithms have been developed for various parallel models. Lee *et al.* [9] presented an $O(\log^2 n)$-time algorithm using n^2 processors on the EREW PRAM. Pavel and Akl [17] 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.* [5] have presented a work-optimal algorithm running in $O(\log n)$ time using $\frac{n^2}{\log n}$ EREW processors and in $O(\frac{\log n}{\log\log n})$ time using $\frac{n^2 \log\log n}{\log n}$ CRCW processors. Later, Hayashi *et al.* [6] have exhibited a more efficient algorithm running in $O(\log n)$ time using $\frac{n^2}{\log n}$ processors on the EREW PRAM and in $O(\log\log n)$ time using $\frac{n^2}{\log\log n}$ processors on the PRAM. Since the product of the computing time and the number of processors is $O(n^2)$, 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. Thus, these algorithms are work-time optimal. Recently, Chen *et al.* [4] have proposed two parallel algorithms for EDM on Linear Array with Reconfigurable Pipeline Bus System [10]. Their first algorithm can compute EDM in $O(\frac{\log n \log\log n}{\log\log\log n})$ time using n^2 processors and second algorithm can compute EDM in $O(\log n \log\log n)$ time using $\frac{n^2}{\log\log n}$ processors.

In practice, now many applications have employed both general multicore processors and emerging GPUs (Graphics Processing Unit) as real platforms to achieve an efficient acceleration. We have also implemented and evaluated our parallel EDM algorithm in the both platforms, a Linux server with four Intel hexad-core processors (Intel Xeon X7460 2.66GHz [8]) and two different GPU (Graphics Processing Unit) systems, Tesla C1060 [15] and GTX 480 [11], respectively. The experimental results show that, for an input binary image with size of 9216×9216, our parallel algorithm can achieve 18 times speedup in the multicore system over the performance of a sequential algorithm. Further, for the same input image, our parallel algorithm for the GPU system achieves a speedup factor of 26.

The remainder of this paper is organized as follows: Section 2 introduces the proximate points problem for Euclidean distance metric and discusses several technicalities that will be crucial ingredients to our subsequent parallel EDM algorithm. Section 3 shows the proposed parallel algorithm for computing Euclidean Distance Map of a 2-D binary image. In Section 4, we show access modes for Steps 1 and 2 in our algorithm. Section 5 exhibits the performance of our proposed algorithm on various multicore platforms. Finally, Section 6 offers concluding remarks.

2 Proximate Points Problem

In this section, we review the proximate problem [6] along with a number of geometric results that will lay the foundation of our subsequent algorithms. Throughout, we assume that a point p is represented by its Cartesian coordinates $(x(p), y(p))$.

Consider a collection $P = \{p_1, p_2, ..., p_n\}$ of n points sorted by x-coordinate, that is, such that $x(p_1) < x(p_2) < ... < x(p_n)$. We assume, without loss of generality, that all the points in P have distinct x-coordinates and that all of them lie above the x-axis. The reader should have no difficulty to confirm that these assumptions are made for convenience only and do not impact the complexity of our algorithms.

Recall that for every point p_i of P the locus of all the points in the plane that are closer to p_i than to any other points in P is referred to as the *Voronoi polygon* associated with p_i and is denoted by $V(i)$. The collection of all the Voronoi polygons of points in P partitions the plane into the Voronoi diagram of P (see [18], p. 204). Let I_i, $(1 \leq i \leq n)$, be the locus of all the points q on the x-axis for which $d(q, p_i) \leq d(q, p_j)$ for all p_j, $(1 \leq j \leq n)$. In other words, $q \in I_i$ if and only if q belongs to the intersection of the x-axis with $V(i)$, as illustrated in Figure 2. In turn, this implies that I_i must be an interval on the x-axis and that some of the intervals I_i, $(2 \leq i \leq n-1)$, may be empty. A point p_i of P is termed a *proximate point* whenever the interval I_i is nonempty. Thus, the Voronoi diagram of P partitions the x-axis into *proximate intervals*. Since the point of P are sorted by x-coordinate, the corresponding proximate intervals are ordered, left to right, as $I : I_1, I_2, ..., I_n$. A point q on the x-axis is said to be a *boundary point* between p_i and p_j if q is equidistance to p_i and p_j, that is, $d(p_i, q) = d(p_j, q)$. It should be clear that p is boundary point between proximate points p_i and p_j if and only if the q is the intersection of the (closed) intervals I_i and I_j. To summarize the previous discussion, we state the following result;

Proposition 2.1. *The following statements are satisfied:*

1) *Each I_i is an interval on the x-axis;*

2) *The intervals $I_1, I_2, ..., I_n$ lie on x-axis in this order, that is, for any nonempty I_i and I_j with $i < j$, I_i lies to the left of I_j.*

3) *If the nonempty proximate intervals I_i and I_j are adjacent, then the boundary point between p_i and p_j separates $I_i \cup I_j$ into I_i and I_j.*

Referring again to Figure 2, among the seven points, five points p_1, p_2, p_4, p_6 and p_7 are proximate points, while the others are not. Note that the leftmost point p_1 and the rightmost point p_n are always proximate points.

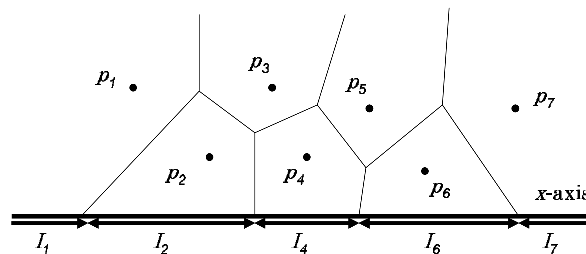


Figure 2: Illustrating proximate intervals

Given three points p_i, p_j, p_k with $i < j < k$, we say that p_j is *dominated* by p_i and p_k whenever p_j fails to be a proximate point of the set consisting of these three points. Clearly, p_j is dominated by p_i and p_k if the boundary of p_i and p_j is to the right of that of p_j and p_k. Since the boundary of any two points can be computed in $O(1)$ time, the task of deciding for every triple (p_i, p_j, p_k), whether p_j is dominated by p_i and p_k takes $O(1)$ time using single processor.

Consider a collection $P = \{p_1, p_2, ..., p_n\}$ of points in the plane sorted by x-coordinate, and a point p to the right of P, that is, such that $x(p_1) < x(p_2) < ... < x(p_n) < x(p)$. We are interested in updating the proximate intervals of P to reflect the addition of p to P, as illustrated in Figure 3.

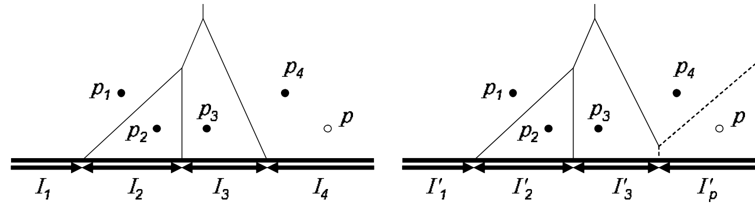


Figure 3: Illustrating the addition of p to $P = \{p_1, p_2, p_3, p_4\}$

We assume, without loss of generality, that all points in P are proximate points and let $I_1, I_2, ..., I_n$ be the corresponding proximate intervals. Further, let $I'_1, I'_2, ..., I'_n, I'_p$ be the updated proximate intervals of $P \cup \{p\}$. Let p_i be a point such that I'_i and I'_p are adjacent. By point 3 in Proposition 2.1, the boundary point between p_i and p separates I'_i and I'_p. As a consequence, point 2 implies that all the proximate intervals $I'_{i+1}, ..., I'_n$ must be empty. Furthermore, the addition of p to P does not affect any of the proximate intervals I_j, $1 \leq j \leq i$. In other words, for all $1 \leq j \leq i$, $I'_j = I_j$. Since $I'_{i+1}, ..., I'_n$ are empty, the points $p_{i+1}, ..., p_n$ are dominated by p_i and p. Thus, every point p_j, $(i < j \leq n)$, is dominated by p_{j-1} and p; otherwise, the boundary between p_{j-1} and p would be to the left of that of that between p_j and p. This would imply that the nonempty interval between these two boundaries corresponds to I'_j, a contradiction. To summarize, we have the following result:

Lemma 2.2. *There exists a unique points of p_i of P such that:*

- *The only proximate points of $P \cup \{p\}$ are $p_1, p_2, ..., p_i, p$.*
- *For $2 \leq j \leq i$, the point p_j is not dominated by p_{j-1} and p. Moreover, for $1 \leq j \leq i-1$, $I'_j = I_j$.*
- *For $i < j \leq n$, the point p_j is dominated by p_{j-1} and p and the interval I'_j is empty.*
- *I'_i and I'_p are consecutive on the x-axis and are separated by the boundary point between p_i and p.*

Let $P = \{p_1, p_2, ..., p_n\}$ be a collection of proximate points sorted by x-coordinate and let p be a point to the left of P, that is, such that $x(p) < x(p_1) < x(p_2) < ... < x(p_n)$. For further reference, we now take note of the following companion result to Lemma 2.2. The proof is identical and, thus, omitted.

Lemma 2.3. *There exists a unique points of p_i of P such that:*

- *The only proximate points of $P \cup \{p\}$ are $p, p_i, p_{i+1}, ..., p_n$.*
- *For $i \leq j \leq n$, the point p_j is not dominated by p and p_{j+1}. Moreover, for $i+1 \leq j \leq n$, $I'_j = I_j$.*
- *For $1 \leq j < i$, the point p_j is dominated by p and p_{j+1} and the interval I'_j is empty.*
- *I'_p and I'_i are consecutive on the x-axis and are separated by the boundary point between p and p_i.*

The unique point p_i whose existence is guaranteed by Lemma 2.2 is termed the *contact point* between P and p. The second statement of Lemma 2.2 suggests that the task of determining the unique contact point between P and a point p to the right or the left of P reduces, essentially, to binary search.

Now, suppose that the set $P = \{p_1, p_2, ..., p_{2n}\}$, with $x(p_1) < x(p_2) < ... < x(p_{2n})$ is partitioned into two subsets $P_L = \{p_1, p_2, ..., p_n\}$ and $P_R = \{p_{n+1}, p_{n+2}, ..., p_{2n}\}$. We are interested in updating the proximate intervals in the process or merging P_L and P_R. For this purpose, let $I_1, I_2, ..., I_n$ and $I_{n+1}, I_{n+2}, ..., I_{2n}$ be the proximate intervals of P_L and P_R, respectively. We assume, without loss of generality, that all these proximate intervals are nonempty. Let $I'_1, I'_2, ..., I'_{2n}$ be the proximate intervals of $P = P_L \cup P_R$. We are now in a position to state and prove the next result which turns out to be a key ingredient in our algorithms.

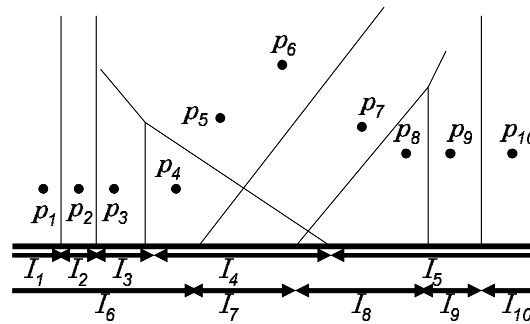
Lemma 2.4. *There exists a unique pair of proximate points $p_i \in P_L$ and $p_j \in P_R$ such that*

- *The only proximate points in $P_L \cup P_R$ are $p_1, p_2, ..., p_i, p_j, ..., p_{2n}$.*
- *$I'_{i+1}, ..., I'_{j-1}$ are empty, and $I'_k = I_k$ for $1 \leq k \leq i-1$ and $j+1 \leq k \leq 2n$.*
- *The proximate intervals I'_i and I'_j are consecutive and are separated by the boundary point between p_i and p_j.*

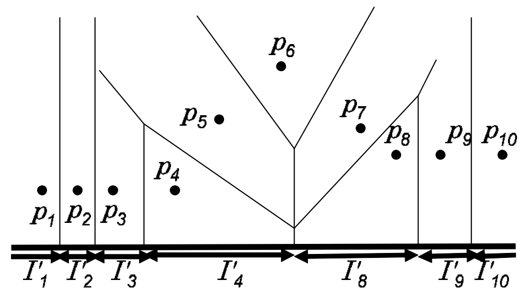
Proof. Let i be the smallest subscript for which $p_i \in P_L$ is the contact point between P_L and a point in P_R. Similarly, let j be the largest subscript for which the point $p_j \in P_R$ is the contact point between P_R and some point in P_L. Clearly, no point in P_L to the left of p_i can be proximate point of P. Likewise, no point in P_R to the left of p_j can be a proximate point of P.

Finally, by Lemma 2.2, every point in P_L to the left of p_i must be a proximate point of P. Similarly, by Lemma 2.3, every point in P_R to the right of p_i must be a proximate point of P, and proof of the lemma is complete. □

The points p_i and p_j whose existence is guaranteed by Theorem 2.4 are termed the *contact points* between P_L and P_R. We refer the reader to Figure 4 for an illustration. Here, the contact points between $P_L = \{p_1, p_2, p_3, p_4, p_5\}$ and $P_R = \{p_6, p_7, p_8, p_9, p_{10}\}$ are p_4 and p_8.



(a) Proximate interval of each point in two sets



(b) Merge of two point sets and their contact points

Figure 4: Illustrating the contact points between two sets of points

Next, we discuss a geometric property that enables the computation of the contact points p_i and p_j between P_L and P_R. For each point p_k of P_L, let q_k denote the contact point between p_k and P_R as specified by Lemma 2.3. We have the following result.

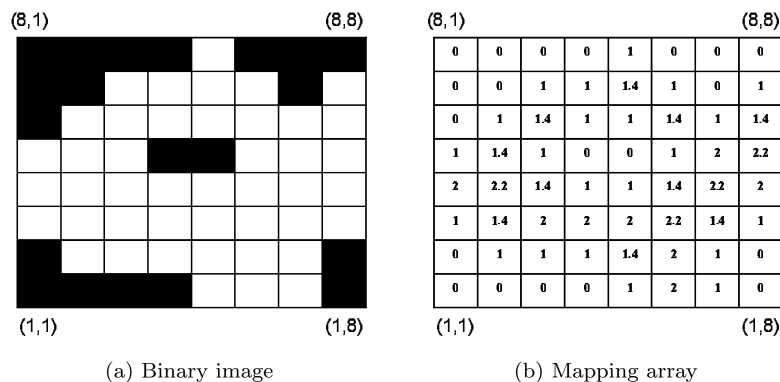
Lemma 2.5. *The point p_k is not dominated by p_{k-1} and q_k if $2 \leq k \leq i$, and dominated otherwise.*

Proof. If p_k, $(2 \leq k \leq i)$, is dominated by p_{k-1} and q_k, then I'_k must be empty. Thus, Lemma 2.4 guarantees that p_k, $(2 \leq k \leq i)$, is not dominated by p_{k-1} and q_k. Suppose that p_k, $(i+1 \leq k \leq n)$, is not dominated by p_{k-1} and q_k. Then, the boundary point between p_k and q_k is to the right of that between these two boundaries corresponds to I'_k, a contradiction. Therefore, p_k, $(i+1 \leq k \leq n)$, is dominated by p_{k-1} and q_k, completing the proof. □

Lemma 2.5 suggests a simple, binary search-like, approach to finding the contact points p_i and p_j between two sets P_L and P_R. In fact, using a similar idea, Breu et al. [1] proposed a sequential algorithm that computes the proximate points of an n-point planar set in $O(n)$ time. The algorithm in [1] uses a stack to store the proximate points found.

3 Parallel Euclidean Distance Map of 2-D Binary Image

A binary image I of size $n \times n$ is maintained in an array $b_{i,j}$, $(1 \leq i, j \leq n)$. It is customary to refer to pixel (i, j) as *black* if $b_{i,j} = 1$ and as *white* if $b_{i,j} = 0$. The rows of the image will be numbered bottom up starting from 1. Likewise, the columns will be numbered left to right, with column 1 being the leftmost. In this notation, pixel $b_{1,1}$ is in the south-west corner of the image, as illustrated in Figure 5(a). In Figure 5(a), each square represents a pixel. For this binary image, its final distance mapping array is shown in Figure 5(b).



(a) Binary image (b) Mapping array

Figure 5: A binary image and its mapping array

The *Voronoi map* associates with every pixel in I the closest black pixel to it (in the Euclidean metric). More formally, the Voronoi map of I is a function $v : I \rightarrow I$ such that, for every (i, j), $(1 \leq i, j \leq n)$, $v(i, j) = v(i', j')$ if and only if

$$d((i, j), (i', j')) = min\{d((i, j), (i'', j'')) \mid b_{i'',j''} = 1\},$$

where $d((i, j), (i', j')) = \sqrt{(i - i')^2 + (j - j')^2}$ is the Euclidean distance between pixels (i, j) and (i', j').

The *Euclidean Distance Map* of image I associates with every pixel in I in the Euclidean distance to the closest black pixel. Formally, the Euclidean Distance Map is a function m: $I \rightarrow R$ such that for every (i, j), $(1 \leq i, j \leq n)$, $m(i, j) = d((i, j), v(i, j))$.

We now outline the basic idea of our algorithm for computing the Euclidean Distance Map of image I. We begin by determining, for every pixel in row j, $(1 \leq j \leq n)$, the nearest black pixel, if any, in the same column of I. More precisely, with every pixel (i, j) we associate the value

$$d_{i,j} = min\{d((i, j), (i', j')) \mid b_{i',j'} = 1, 1 \leq j' \leq n\}.$$

If $b_{i',j'} = 0$ for every $1 \leq j' \leq n$, then let $d_{i,j} = +\infty$. Next, we construct an instance of the proximate points problem for every row j, $(1 \leq j \leq n)$, in the image I involving the set P_j of points in the plane defined as $P_j = \{p_{i,j} = (i, d_{i,j}) \mid 1 \leq i \leq n\}$.

Having solved, in parallel, all these instances of the proximate points problem, we determine, for every proximate point $p_{i,j}$ in P_j, its corresponding proximity interval I_i. With j fixed, we determine, for every pixel (i, j) (that we perceive as a point on the x-axis), the identity of the proximity interval to which it belongs. This allows each pixel (i, j) to determine the identity of the nearest pixel to it. The same task is executed for all rows $1, 2, ..., n$ in parallel, to determine, for every pixel (i, j) in row j, the nearest black pixel. The details are spelled out in the following algorithm:

Algorithm : ***Euclidean Distance Map(I)***

Step 1 For each pixel (i, j), compute the distances

$$d_{i,j} = min\{|k - i| \mid b_{k,j} = 1, 1 \leq k \leq n\}$$

to the nearest black pixel in the same column.

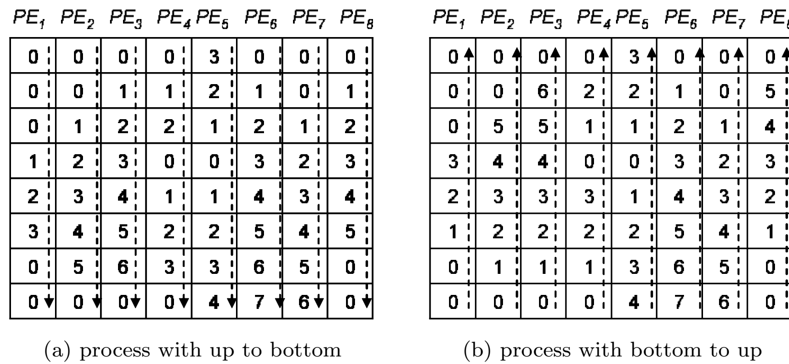
Step 2 For every j, $(1 \leq j \leq n)$, let $P_j = \{p_{i,j} = (i, d_{i,j}) \mid 1 \leq i \leq n\}$. Compute the proximate points $E(P_j)$ of P_j.

Step 3 For every point p in $E(P_j)$ determine its proximity interval of P_j.

Step 4 For every i, $(1 \leq i \leq n)$, determine the proximate interval of P_j to which the point $(i, 0)$ (corresponding to pixel (i, j)) belongs.

We assume that there are n processors PE(1), PE(2), ..., PE(n) available. The parallel implementation of above algorithm is shown as follows:

Step 1 We assign the i-th column $(1 \leq i \leq n)$ to processor PE(i) to computes the distance to the nearest black pixel in the same column. First, each PE(i) $(1 \leq i \leq n)$ reads pixel values in the i-th column from up to bottom to compute that distance, as illustrated in Figure 6(a) (its original input image is shown in Fig 5). Second, each processor PE(i) $(1 \leq i \leq n)$ read



(a) process with up to bottom (b) process with bottom to up

Figure 6: Process each column with two directions

pixel values in the i-th column from bottom to up to compute that distance, as illustrated in Figure 6(b). Finally, each processor selects a minimum value of calculated two distances as final value of the distance. It is clear that the time complexity of this step is $O(n)$.

Step 2 Again, we compute Euclidean Distance Map of input image I along with row wise.

Step 2.1 For every i-th row $(1 \leq i \leq n)$, each processor PE(i) computes the proximate points using the theorem of proximate points problem as foundation, as illustrated in Figure 7 and

(8,1) (8,8)

PE_8	0	0	0	0	3	0	0	0
PE_7	0	0	1	1	2	1	0	1
PE_6	0	1	2	1	1	2	1	2
PE_5	1	2	3	0	0	3	2	3
PE_4	2	3	3	1	1	4	3	2
PE_3	1	2	2	2	2	5	4	1
PE_2	0	1	1	1	3	6	5	0
PE_1	0	0	0	0	4	7	6	0

(1,1) (1,8)

Figure 7: Processing with row wise

Figure 8. In Figure 8, the Voronoi polygons correspond to 5th row (shaded row) of the image illustrated in Figure 7. The obtained proximate points are saved in a stack. It should be clear that each column has its own corresponding stack. Therefore, in order to add a new proximate point to the stack, we need to calculate boundary points of this new point and existed proximate points which are kept in the stack. Then according to locus of boundary points, we decide which points need to be deleted from the stack.

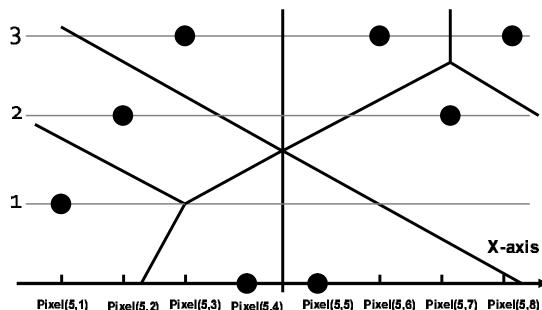


Figure 8: Voronoi polygons

Step 2.2 For every i-th row ($1 \leq i \leq n$), each processor PE(i) determines proximate intervals of obtained proximate points by computing boundary point of each pair of adjacent proximate points. The boundary point of each pair of adjacent proximate points can be obtained by calculating the intersection point of two lines, one line is x-axis and another is the normal line of the line which connects two adjacent proximate points. We refer reader to Figure 9 for the illustration. Each pair of adjacent proximate points can be obtained from the stack.

Step 2.3 According to the locus of boundary points obtained from Step 2.2, each processor determines the closest black pixel to each pixel of input image. The distance between a given pixel and its closest black pixel is also calculated in the obvious way.

It should be clear that, the whole Step 2 can be implemented in $O(n)$ time using n processors.

Theorem 3.1. *For a given binary image I with the size of $n \times n$, Euclidean Distance Map of image I can be computed in $O(n)$ time using n processors.*

Suppose that we have k processors ($k < n$). If this is the case, a straightforward simulation of n processors by k processors can achieve optimal slowdown. In other words, each of the k processors performs the task of $\frac{n}{k}$ processors in our Euclidean Distance Map algorithm. For example, in Step 1,

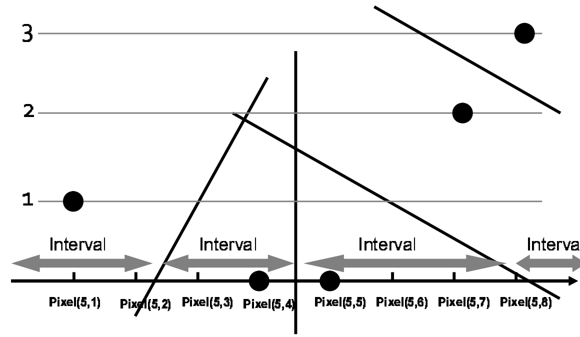


Figure 9: Proximate intervals

the i-th processor ($1 \leq i \leq k$) computes the nearest black pixel within the same column for rows from $(i-1) \cdot \frac{n}{k} + 1$-th to $i \cdot \frac{n}{k}$. This can be done in $O(n \cdot \frac{n}{k}) = O(\frac{n^2}{k})$ time. Thus, we have,

Corollary 3.2. *For a given binary image I with the size of $n \times n$, Euclidean Distance Map of image I can be computed in $O(\frac{n^2}{k})$ time using k processors.*

4 Access Modes

As known, in general, a matrix is stored in a row-major fashion in memory. In other words, the (i, j)-th element of a matrix is arranged to the $i \cdot w + j$-th element in an array in the memory, where w is the width of the matrix as illustrated in Figure 10.

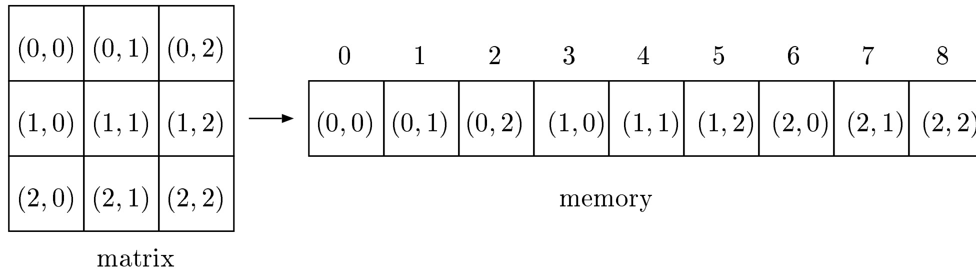


Figure 10: Arrangement of a 3×3 matrix into a memory

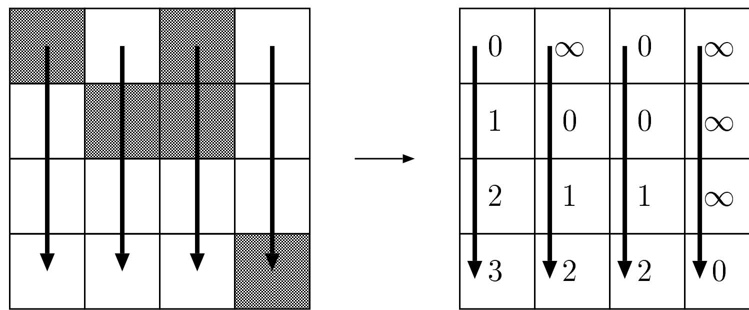
The key part of our Euclidean Distance Map algorithm is Step 1 and Step 2. We will define several access modes which affect the performance of our algorithm. Recall that in Step 1, pixel values are read in column wise, and the distances to the nearest black pixel are written in column wise. Instead, we can write the distances to the nearest black pixel in row wise. In other words, we can read the pixel values in column wise (i.e. *Vertical*), or in row wise (i.e. *Horizontal*) and write the distances in column wise (i.e *Vertical*) or in row wise (i.e. *Horizontal*). The readers should refer to Figure 11 for illustrating the possible four access modes of Step 1.

Let $d_{i,j}$ denote the resulting distances of Step 1. For each access mode we can write $d_{i,j}$ as follows:

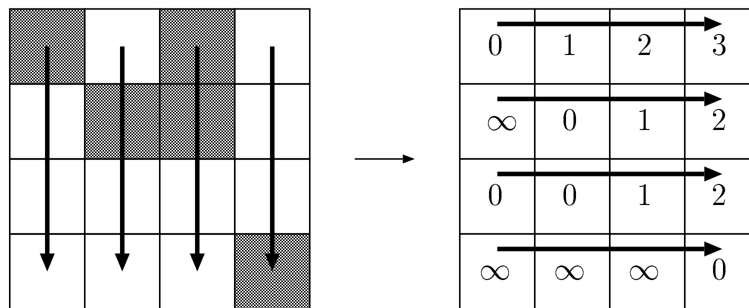
VV (Vertical-Vertical) $d_{i,j} = min\{|k-i| \mid b_{k,j} = 1, 1 \leq k \leq n\}$

VH (Vertical-Horizontal) $d_{j,i} = min\{|k-i| \mid b_{k,j} = 1, 1 \leq k \leq n\}$

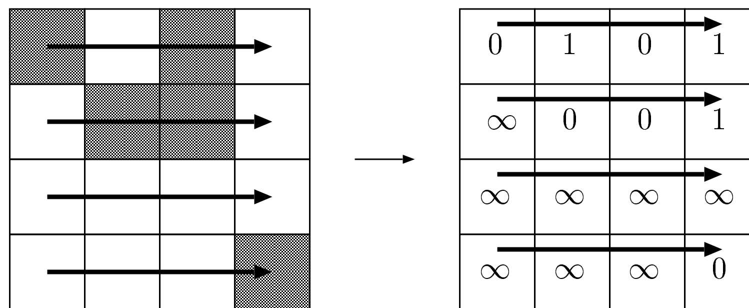
HH (Horizontal-Horizontal) $d_{i,j} = min\{|k-j| \mid b_{i,k} = 1, 1 \leq k \leq n\}$



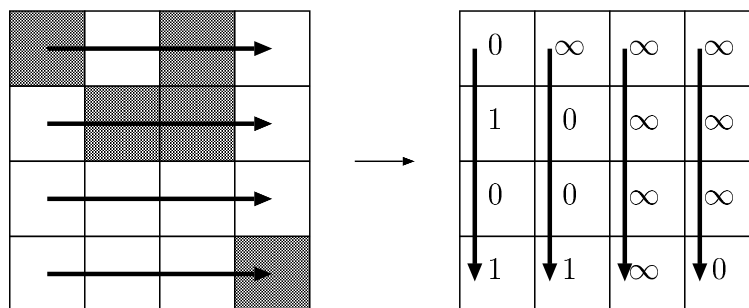
VV (Veritcal-Veritcal) access mode



VH (Vertical-Horizontal) access mode



HH(Horizontal-Horizontal) access mode



HV(Horizontal-Vertical) access mode

Figure 11: Access modes for Step 1

HV (Horizontal-Vertical) $d_{j,i} = min\{|k - j| \mid b_{i,k} = 1, 1 \leq k \leq n\}$

Note that, for VH and HV access modes, the resulting values stored in the two dimensional array is transposed.

In the same way, we can define four possible access modes VV, VH, HH, HV for Step 2. For example, in VV mode, the distances are read in column wise and the resulting values of Euclidean Distance Map are written in column wise.

The readers should have no difficulty to confirm that possible combinations of access modes for Steps 1 and 2 are **VVHH**, **HHVV**, **VHVH**, and **HVHV**, because the access mode satisfy the following two conditions:

Condition 1 If the resulting values in Step 1 are stored in a transposed array, those in Step 2 also must be transposed. Otherwise, the resulting Euclidean Distance Map is transposed.

Condition 2 The writing directions of Step 1 and Step 2 must be orthogonal.

Therefore, in the notation $r_1w_1r_2w_2$ of access modes, w_1 and r_2 must be distinct from Condition 1 and the number of H in r_1, w_1, r_2, and w_2 must be even from Condition 2. Therefore, the possible access modes are VVHH, HHVV, VHVH, and HVHV.

5 Experimental Results

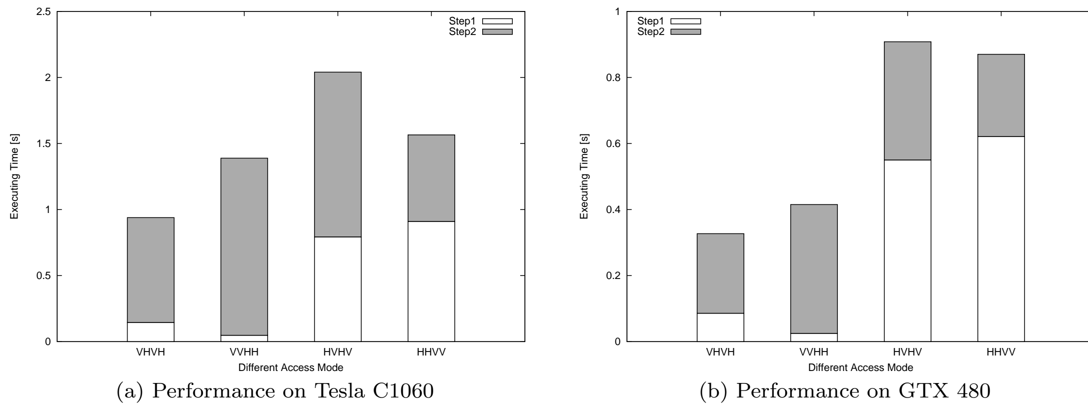
We have implemented and evaluated our proposed parallel EDM algorithm in the following two different platforms, a general multicore processor system and modern GPU systems, respectively. The multicore processor system is a Linux server with four Intel hexad-core processors (Intel Xeon X7460 2.66GHz [8]), that is, there are twenty four cores available. Each multicore processor has its own local three-level caches that are 64KB L1 cache, 3MB L2 cache and 16MB L3 cache. The capacity of the main memory is 128GB. One of the experimental GPU systems is a Tesla C1060 [15] which consists of 240 Streaming Processor Cores and 4GB Global Memory. Another one is GTX 480 [11] which consists of 480 Streaming Processor Cores and 1.5GB Global Memory. GTX 480 can provide L1 and L2 caches to Global Memory [13].

Our proposed algorithm has implemented in C language with OpenMP 2.0 (Open Multi-Processing) in that of multicore processor system. The OpenMP is an application programming interface that supports shared memory environments [16]. It consists of a set of compiler directives and library routines. Using OpenMP, it is relatively easy to create parallel applications in FORTRAN, C, and C++. Table 1 shows the performance of our proposed algorithm with different access modes in the multicore processor system. The size of the input image is 9216 × 9216. In the table, each measurement is an average value of 20 experiments and, Step 1 and Step 2 are corresponding steps of our proposed parallel algorithm. It is clear that, in HVHV access mode, our implementation can achieve the best performance and it can obtain approximate 18 times speedup. The table also exhibits the scalability of the proposed algorithm. As shown, our proposed algorithm can scale well with the number of using cores smaller than or equal to 4. Actually we have implemented the proposed algorithm in a multiprocessor system with 4 multicore processors. Therefore when the number of using cores is smaller than or equal to 4, all the using cores will be distributed into different multicore processors. Consequently each level cache of a multicore processor is occupied by only one core. It means only one core utilizing all the available cache. However, when the number of using threads is more than 4, the scalability of our implementation is decreasing significantly. One main reason of the phenomenon is that, when the number of using cores is larger than 4, L2 and L3 cache of each multicore processor will be shared by multiple cores. It will decrease the efficiency of our implementation significantly. Meanwhile, many other factors such as Memory-CPU bus bandwidth, communication overhead and synchronization overhead also play the important roles in the scalability. Hence we can understand why the real speedup is decreasing along with increasing the number of using processors.

In another way, our proposed algorithm has been implemented in that of GPU systems using CUDA (Compute Unified Device Architecture) [12], a general purpose parallel computing architecture. Actually, CUDA is a new parallel programming model and instruction set architecture. CUDA

comes with a software environment that allows developers to use C-like high-level programming language.

As known, an important programming issue on GPUs is the reduction of heavy access latency of Global Memory [13]. Fortunately, the CUDA can provide a technique known as coalescing [13] to hide the access latency of the Global Memory. When 16 (or 32) sequential threads access 16 (or 32) sequential and aligned values in the Global Memory, the GPU will automatically combine them into a single transaction.



(a) Performance on Tesla C1060 (b) Performance on GTX 480

Figure 12: Performance of the proposed algorithm on different GPU systems with different access mode (n=9216)

Figure 12 shows the performance of our proposed algorithm with different access modes in that of GPU systems. The input image is the same image used in the multicore implementation. Recall that the size of input image is 9216×9216. As shown in the Figure, different with the multicore implementation, our proposed algorithm can achieve the best performance in VHVH access mode on the GPUs.

For clear explanation, first we describe the details of the GPU implementation of the proposed parallel Euclidean Distance Map algorithm. Here we just describe the GPU implementation of VHVH access mode. For other access modes, their implementations can be understood in the same way.

For implementing Step 1 of the proposed algorithm, we partition the original input image into $(\frac{n}{k})$ subimages along with column wise. It means that there are $(\frac{n}{k})$ CUDA blocks [13] and each CUDA block processes each corresponding subimage independently. The number of threads in each CUDA block is k and we configure the value of k according to the occupancy [13] of CUDA. For example, for processing image with size of 4608×4608, if we set the number of using threads as 512, then there is only 9 CUDA blocks created and as the result, only 9 streaming processors are in active. However, in Tesla C1060 system, there are 30 streaming processors are available. It means that there are 21 streaming processors in idle status. Therefore considering the occupancy, we can set the number of using threads in a CUDA block as 128 or 64. Each thread of a CUDA block processes each corresponding column of the sumbimage. We refer reader(s) to Figure 13 as an simple illustration. In Figure 13, each T_i represents a thread of a CUDA block and each arrow represents an access of a pixel value by one thread. It is clear that, for a sumbimage, the access of each row can be performed in coalescing.

By following Step 1 of the proposed parallel EDM algorithm, we can easy to know that, each thread need to access each pixel value of the corresponding column two times. One is for computing results of up-to-bottom process and another one is for computing results of bottom-to-up process. After selecting a minimum value for each pixel, each thread writes the selecting result into an extra array, which stores the results of Step 1, along with row wise. It is clear that, the both up-to-bottom process and bottom-to-up process can benefit from full coalescing. But the writing of the extra array

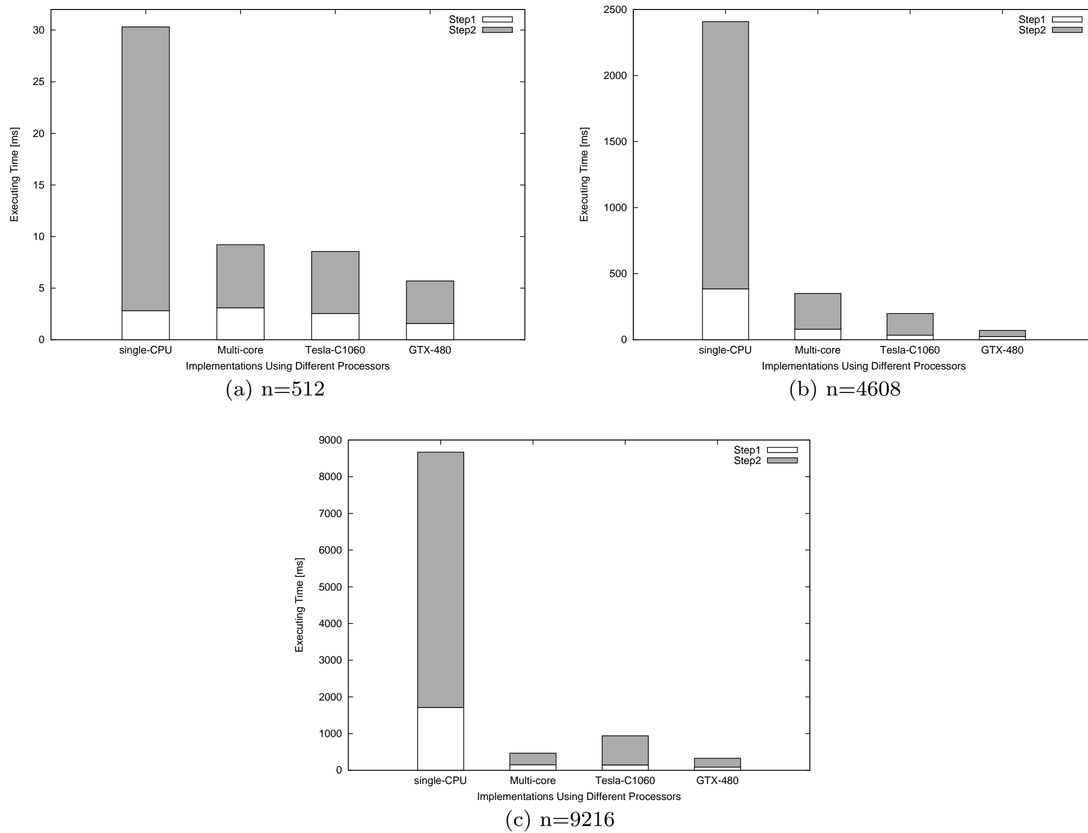
Figure 13: Mapping CUDA Blocks into subimages

cannot benefit from the coalescing at all (recall that now we are describing the implementation of VHVH access mode). However, in the implementation of VVHH access mode, the writing of the extra array is also can benefit from the full coalescing. Therefore in VVHH access mode, the implementation of Step 1 can achieve the most significant performance (as shown in Figure 12). Differently, in HHVV access mode, the whole implementation of Step 1 can not benefit from the coalescing at all. Therefore Step 1 of the HHVV access mode achieved the worst performance (as shown in Figure 12). Other than the Step 1 of HHVV access mode, in Step 1 of HVHV access mode, the extra array can be written by using the coalescing. Therefore it can achieve a little better performance than HHVV access mode (as shown in Figure 12).

In Step 2 of the proposed algorithm, a stack is needed for computing boundary points of each column. Therefore we need to allocate a 2-D array in Global memory for keeping all the stacks. We use each column of the 2-D array as the stack for each corresponding column of input image. Each thread accesses elements of corresponding column of the extra array, which stores the results of Step 1, to obtain elements of corresponding stack (recall that now we are describing the implementation of VHVH access mode). However the push-pop operations of all stacks are not uniform. Therefore the access of the extra array can not be performed in full coalescing. In the same way, the access of the stacks also can not be performed in full coalescing. This is reason to why the implementation of Step 2 can not achieve a significant performance even in HHVV access mode. After computing boundary points, we compare the y-coordinate of each boundary point with the y-coordinate of each pixel to obtain the distance to closest black pixel. If we assume that the mapping results will be stored in a 2-D array named output array, it needs all threads accesses the output array along with row wise. In other words, each thread will access the corresponding row of the output array, and it can not utilize the coalescing (recall that now we are describing the implementation of HVHV access mode). However, in Step 2 of VVHH access mode, its whole implementation can not benefit from the coalescing at all. This is reason to why Step 2 of HVHV access mode can be little faster than Step 2 of VVHH access mode.

On the other hand, we have also evaluated the proposed parallel algorithm with the different sized input images. Figure 14 shows the performance of the proposed parallel algorithm for processing images with different sizes on different parallel systems. The maximum number of available threads on a CUDA block is always proportion to 512. Therefore the size of input images is also configured to be proportion to 512. For each system, we have shown the performance of the corresponding implementation with the most efficient access mode.

Actually the proposed parallel EDM algorithm is simple. However it also can give us the fol-



(a) n=512

(b) n=4608

(c) n=9216

Figure 14: Performance of the proposed algorithm on different parallel systems for processing images with different sizes

lowing information. Each processor will be mapped into each subimage statically and sizes of all subimages are almost same. Therefore the proposed algorithm can achieve a perfect loading balance for all processors. On the other hand, each processor processes each subimage independently. There is no any data sharing between any two processors. It means there is no any communication between processors. Considering all of this, it is clear that why the multi-core implementation of the proposed algorithm can achieve such good performance (as shown in Figure 14). Figure 14 also show that, comparing with the performance of multi-core implementation, the performance of the GPU implementation on Tesla C1060 is not so good. Actually from explanation of Figure 12, we know that, the coalescing access of Global memory plays an important role in GPU programming. However, no one implementation of proposed four access modes can benefit from the coalescing in all implementing steps. Therefore, for processing image with size of 9216×9216, the GPU implementation on Tesla C1060 have just achieved 9 times speedup. However, other than in Tesla C1060, the GPU implementation have achieved about 26 times speed up on that of GTX 480. Since, GTX 480 can be programmed in new generation of CUDA architecture named *Fermi* [14]. In Fermi architecture, the number of active threads in a CUDA block is increased to 32 (full wrap [13]). Differently, in the old CUDA architecture, the number of active threads is just 16 (half wrap). Additionally, other than Tesla C1060, GTX 480 can provide two level L1 and L2 caches to global memory and, in the Step 2 of our proposed algorithm, we have used stack which is very sensitive to caches.

As shown in Figure 14, for processing small sized images, our implementations can not obtain a significant speedup factor in all parallel systems, because in comparison with total execution time, there is considerable overhead due to parallel processing.

6 Concluding remarks

In this paper, we have presented an optimal parallel algorithm for computing Euclidean Distance Map (EDM) of a 2-D binary image. Using proximate points problem as preliminary foundation, we have proposed a simple but efficient parallel EDM algorithm which can achieve $O(\frac{n^2}{k})$ time using k processors. To evaluate the performance of the proposed algorithm, we have implemented it in a Linux server with four Intel hexad-core processors (Intel Xeon X7460 2.66GHz) [8] and two different GPU (Graphics Processing Unit) systems, Tesla C1060 [15] and GTX 480 [11], respectively. The experimental results have shown that, for an input binary image with size of 9216×9216, the proposed parallel algorithm can achieve 18 times speedup in the multicore system, comparing with the performance of general sequential algorithm. Meanwhile, for the same input image, the proposed parallel algorithm can achieve 26 times speedup in that of GPU systems.

References

[1] Heinz Breu, Joseph Gil, David Kirkpatrick, and Michael Werman. Linear time euclidean distance transform algorithms. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 17(5):529–533, May 1995.

[2] Ling Chen. Optimal algorithm for complete euclidean distance transform. *Chinese Journal Computers*, 18(8):611–616, 1995.

[3] Ling Chen and Henry Y. H. Chuang. A fast algorithm for euclidean distance maps of a 2-d binary image. *Information Processing Letters*, 51:25–29, 1994.

[4] Ling Chen, Yi Pan, Yixin Chen, and Xiao hua Xu. Efficient parallel algorithms for euclidean distance transform. *The Computer Journal*, 47(6):694–700, 2004.

[5] Akihiro Fujiwara, Toshimitsu Masuzawa, and Hideo Fujiwara. An optimal parallel algorithm for the euclidean distance maps of 2-d binary images. *Information Processing Letters*, 54:295–300, 1995.

[6] Tatsuya Hayashi, Koji Nakano, and Stephan Olariu. Optimal parallel algorithm for finding proximate points, with applications. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1153–1166, December 1998.

[7] Tomio Hirata. A unified linear-time algorithm for computing distance maps. *Information Processing Letters*, 58:129–133, 1996.

[8] Intel Corporation. *Intel Xeon Processor 5000 Sequence*. http://www.intel.com/products/processor/xeon7000/.

[9] Yu-Hua Lee, Shi-Jinn Horng, Tzong-Wann Kao, Ferng-Shi Jaung, Yuung-Jih Chen, and Horng-Ren Tsai. Parallel computation of exact euclidean distance transform. *Parallel Computing*, 22(2):311–325, 1996.

[10] Keqin Li, Yi Pan, and Si-Qing Zheng. *Parallel Computing Using Optical Interconnections*. Kluwer Academic Publishers, Boston, USA, 1998.

[11] NVIDIA Corporation. *GetForce GTX 480*. http://www.nvidia.com/object/product_getforce_gtx_480_us.html.

[12] NVIDIA Corporation. *NVIDIA, CUDA Architecture*. http://www.nvidia.com/object/cuda_home_new.html.

[13] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf.

[14] NVIDIA Corporation. *NVIDIA, Fermi Architecture*. http://www.nvidia.com/object/fermi_architecture.html.

[15] NVIDIA Corporation. *Tesla C1060 Computing Processor*. http://www.nvidia.com/object/product_tesla_c1060_us.html.

[16] OpenMP.org. *OpenMP Application Program Interface*. http://www.openmp.org.

[17] Sandy Pavel and Selim G. Akl. Efficient algorithms for the euclidean distance transform. *Parallel Processing Letters*, 5(2):205–212, 1995.

[18] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Berlin: Springer-Verlag, third corrected printing edition, 1990.

Table 1: Performance of proposed algorithm in multicore processor system with different access mode (n=9216)

(a) HVHV access mode

Num of using cores	Step 1 [s]	Speedup	Step 2 [s]	Speedup	Total [s]	Speedup
1	1.7120	1.00	6.9570	1.00	8.6690	1.00
2	0.8690	1.97	3.5556	1.95	4.4246	1.95
4	0.5001	3.42	1.7429	3.99	2.2430	3.86
8	0.2899	5.90	0.9009	7.72	1.1908	7.27
12	0.2199	7.78	0.6824	10.19	0.9023	9.60
16	0.1616	10.59	0.5579	12.46	0.7195	12.04
20	0.1515	11.30	0.4689	14.83	0.6204	13.97
24	0.1501	11.40	0.3164	21.98	0.4665	18.58

(b) HHVV access mode

Num of using cores	Step 1 [s]	Speedup	Step 2 [s]	Speedup	Total [s]	Speedup
1	0.5010	1.00	11.0020	1.00	11.5030	1.00
2	0.2510	1.99	5.5998	1.96	5.8508	1.96
4	0.1621	3.09	2.9361	3.74	3.0982	3.71
8	0.0986	5.08	1.4365	7.65	1.5351	7.49
12	0.0945	5.30	1.2510	8.79	1.3455	8.54
16	0.0901	5.56	0.9557	11.51	1.0458	10.99
20	0.0921	5.43	0.8223	13.37	0.9144	12.57
24	0.0954	5.25	0.9221	11.93	1.0175	11.30

(c) VVHH access mode

Num of using cores	Step 1 [s]	Speedup	Step 2 [s]	Speedup	Total [s]	Speedup
1	4.6215	1.00	6.0230	1.00	10.6445	1.00
2	2.3666	1.95	2.9645	2.03	5.3311	1.99
4	1.1999	3.85	1.5451	3.89	2.7450	3.87
8	0.6189	7.46	0.7774	7.74	1.3963	7.62
12	0.5892	7.84	0.5221	11.53	1.1113	9.57
16	0.5718	8.08	0.3912	15.39	0.9630	11.05
20	0.5426	8.51	0.3021	19.93	0.8447	12.60
24	0.6198	7.45	0.2697	22.33	0.8895	11.96

(d) VHVH access mode

Num of using cores	Step 1 [s]	Speedup	Step 2 [s]	Speedup	Total [s]	Speedup
1	2.4126	1.00	8.6009	1.00	11.0135	1.00
2	1.2169	1.98	4.3899	1.95	5.6068	1.96
4	0.6201	3.89	2.2158	3.88	2.8359	3.88
8	0.3621	6.66	1.3003	6.61	1.6624	6.62
12	0.2479	9.73	0.8125	10.58	1.0604	10.38
16	0.2319	10.40	0.6415	13.40	0.8734	12.60
20	0.1789	13.48	0.5211	16.50	0.7000	15.73
24	0.1625	14.84	0.4497	19.12	0.6122	17.99