# A GPU Implementation of Computing Euclidean Distance Map with Efficient Memory Access

Duhu Man, Kenji Uda, Yasuaki Ito, and Koji Nakano
*Department of Information Engineering,*
*Hiroshima University*
*1-4-1 Kagamiyama, Higashi Hiroshima, Hiroshima, 739–8527, Japan*
{*manduhu, uda, yasuaki, nakano*}*@cs.hiroshima-u.ac.jp*

***Abstract*—Recent Graphics Processing Units (GPUs), which have many processing units, can be used for general purpose parallel computation. To utilize the powerful computing ability, GPUs are widely used for general purpose processing. Since GPUs have very high memory bandwidth, the performance of GPUs greatly depends on memory access. The main contribution of this paper is to present a GPU implementation of computing Euclidean Distance Map (EDM) with efficient memory access. Given a 2-D binary image, EDM is a 2-D array of the same size such that each element is storing the Euclidean distance to the nearest black pixel. In the proposed GPU implementation, we have considered many programming issues of the GPU system such as coalescing access of global memory, shared memory bank conflicts and partition camping. In practice, we have implemented our parallel algorithm 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 $9216 \times 9216$, our implementation can achieve a speedup factor of 52 over the sequential algorithm implementation.**

***Keywords*-Euclidean Distance Map; Proximate Points; GPU; Coalescing Access;**

## I. 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, such as sequential algorithm [1], [2], [3], [4] and parallel algorithm [5], [6], [7]. Recently, Chen *et al.* [8] have proposed two parallel algorithms for EDM on Linear Array with Reconfigurable Pipeline Bus System [9]. Their first algorithm can computes EDM in $O(\frac{\log n \log\log n}{\log\log\log n})$ time using

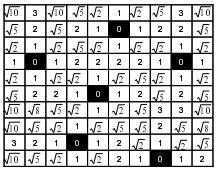


Figure 1. Euclidean Distance Map

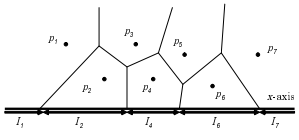



Figure 2. Proximate intervals

$n^2$ 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 emerging GPUs (Graphics Processing Unit) as real platforms to achieve an efficient acceleration. In GPU implementation, there are some programming issues of the GPU system such as coalescing access of global memory, shared memory bank conflicts and partition camping [10]. Coalescing is to hide the access latency of the global memory. When sequential threads access sequential and aligned values in the off-chip global memory, the GPU will automatically combine them into a single transaction. An on-chip shared memory is divided into 16 equally-sized modules of 32-bit width, called banks. In the on-chip shared memory, the successive 32-bit words are assigned to successive banks. To avoid bank conflicts and achieve maximum throughput, concurrent threads should access different banks. Another important programming issue for global memory is that named as partition camping. Actually the global memory is divided into several partitions of 256-byte width.

In our previous paper [11], we have shown 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 and a modern GPU system, respectively. The experimental results have shown that, for an input binary image with size of $10000 \times 10000$, the proposed parallel algorithm can achieve 18 times speedup in the multicore system, comparing with

978-0-7695-4569-1/11 $26.00 © 2011 IEEE
DOI 10.1109/ICNC.2011.19

the performance of general sequential algorithm. Meanwhile, for the same input image, the proposed parallel algorithm can achieve 5 times speedup in that of GPU system. However, it is not enough to cope the above programing issues. Therefore, the main contribution of this paper is to show a GPU implementation of the proposed algorithm with more efficient memory access. In GPU implementation, we have considered many programming issues of the GPU system such as coalescing access of global memory, shared memory bank conflicts and partition camping. We have implemented and evaluated our new parallel EDM algorithm in the following two GPU systems, Tesla C1060 [12] and GTX 480 [13], respectively. The experimental results have shown that, for an input binary image with size of $9216 \times 9216$, our implementation can achieve a speedup factor of 52 over the sequential algorithm implementation.

## II. 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 [14], 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 points 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.

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.

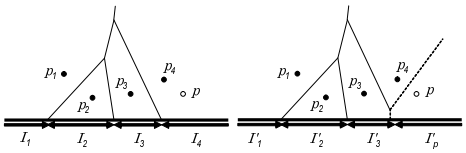



Figure 3. Addition of $p$ to $P = \{p_1, p_2, p_3, p_4\}$

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.

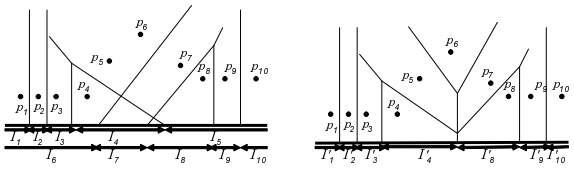
Now, suppose that the set $P = \{p_1, p_2, ..., p_{2n}\}$, with $x(p_1) < x(p_2) < ... < x(p_{2n})$ is partitioned into two subsets $P_L = \{p_1, p_2, ..., p_n\}$ and $P_R = \{p_{n+1}, p_{n+2}, ..., p_{2n}\}$. We are interested in updating the proximate intervals in the process or merging $P_L$ and $P_R$. 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 $p_j$.*

The proof has been shown in [15].

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. 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.

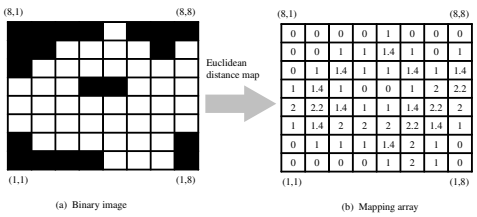
*Lemma 2.5: The point $p_k$ is not dominated by $p_{k-1}$ and $q_k$ if $2 \leq k \leq i$, and dominated otherwise.*

The proof has been shown in [15].

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.

## III. 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 the figure, each square represents a pixel. For this binary image, its final distance mapping array is shown in Figure 5(b).




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1.4 | 1 | 0 | 1 |
| 0 | 1 | 1.4 | 1 | 1 | 1.4 | 1 | 1.4 |
| 1 | 1.4 | 1 | 0 | 0 | 1 | 2 | 2.2 |
| 2 | 2.2 | 1.4 | 1 | 1 | 1.4 | 2.2 | 2 |
| 1 | 1.4 | 2 | 2 | 2 | 2.2 | 1.4 | 1 |
| 0 | 1 | 1 | 1 | 1.4 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 2 | 1 | 0 |

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 subimage 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 $d_{i,j} = min\{d((i, j), (i', j')) \mid b_{i',j'} = 1, 1 \leq j' \leq n\}$ to the nearest black pixel in the same column as $(i, j)$ in the subimage of $I$.

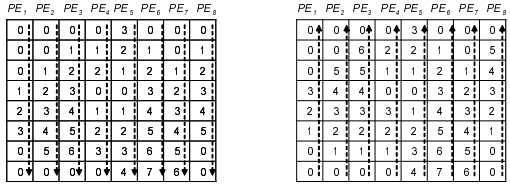
**Step 2.** For every $j$, $(1 \leq j \leq n)$, let $P_j = \{p_{i,j} = (i, d_{i,j}) \mid 1 \leq i \leq n\}$. Compute the proximate points $E(P_j)$ of $P_j$.

**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 $k$ processors PE(1), PE(2), ..., PE($k$) available. The parallel implementation of above algorithm is shown in follow:

**Step 1.** Partition the input image $I$ into $k$ subimages $I_1, I_2, ..., I_k$ along with column wise. For every pixel of each subimage $I_i$ $(1 \leq i \leq k)$, corresponding processor PE($i$) computes the distance to the nearest black pixel in the same column. In real implementation, first, each processor travels every column of corresponding subimage 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




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

Figure 6. Process each column with two directions

processor again travels every column of corresponding subimage 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 $O(n^2/k)$.

**Step 2.** Again, we compute Euclidean distance map of input image $I$ along with row wise.

**Step 2.1** Partition the input image into $k$ subimages $I'_1, I'_2, ..., I'_k$ along with row wise. For every row of each subimage $I'_i$ $(1 \leq i \leq k)$, each processor PE($i$) $(1 \leq i \leq k)$ computes the proximate points using the theorem of proximate points problem as foundation, as illustrated in Figure 7 and Figure 8. In Figure 8, the Voronoi polygons

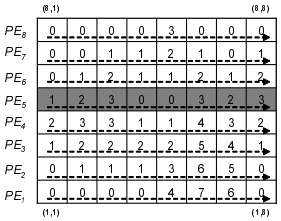



Figure 7. Processing with row wise

correspond to $5^{th}$ row (shaded row) of the image shown in Fig 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.

**Step 2.2** For every row of each subimage $I'_i$ $(1 \leq i \leq k)$, each processor PE($i$) determines proximate intervals of

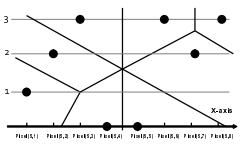



Figure 8. Voronoi polygons

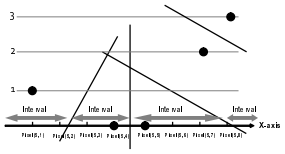



Figure 9. Proximate intervals

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, 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.

## IV. Compute Unified Device Architecture (CUDA)

Graphics Processing Units (GPUs) can achieve a high computational throughput due to their large number of processing cores and different memory spaces. All the processing cores are organized into several streaming multi-core processors. For fully utilizing all the processing cores of a GPU, numerous threads are required. Compute Unified Device Architecture (CUDA) [16] organizes these threads into a large *grid* of *thread blocks*. Each thread block contains a number of threads which can be executed on an assigned streaming multi-core processor. Threads of a thread block are organized into several *warps* and each warp contains 32 threads. At a time, only a half warp (or full warp) of a thread block can be executed by the assigned streaming multi-core processor concurrently . The grid will launch a segment of codes, named a *kernel*, to occupy a GPU device at a time. 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.

On the other hand, GPUs can provide different memory spaces for different applications and each memory space has its own advantages and drawbacks. In CUDA architecture, each memory space has a corresponding specification. In this paper we only introduce few of them shown as follows.

*Global memory* is a main device memory of GPUs and which is off-chip memory. Therefore it has heavy access latency to each processing core. Fortunately, CUDA provides a technique known as *coalescing* [10] 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. Another important programming issue for global memory is that named as *partition camping*. Actually the global memory is divided into several partitions of 256-byte width. If several concurrent thread blocks access through a same partition, it will cause the partition camping. In order to avoid the partition camping, concurrent accesses to global memory by all active thread blocks need to be separated evenly amongst partitions. If several concurrent blocks access through a same partition, it will cause the partition camping.

*Shared memory* is a sort of on-chip memory and which is located within each streaming multi-core processor. It has almost no access latency and only visible to the thread block which is executed by the corresponding streaming multi-core processor. Just as global memory is divided into several partitions, shared memory is also divided into 16 equally-sized modules of 32-bit width, called banks. It means that, in shared memory, the successive 32-bit words are assigned to successive banks. To achieve maximum throughput, concurrent threads of a thread block should access different banks, otherwise, bank conflicts will occur. In practice, the shared memory can be used as a cache to hide the access latency of the global memory.

## V. Our Previous Implementation of EDM Algorithm on GPUs

In this section, we show our previous implementation of EDM algorithm on GPUs [15].

### A. 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.

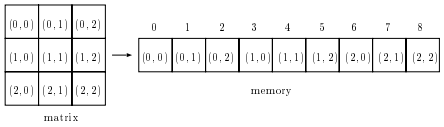



Figure 10. Arrangement of a $3 \times 3$ matrix into a memory

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.

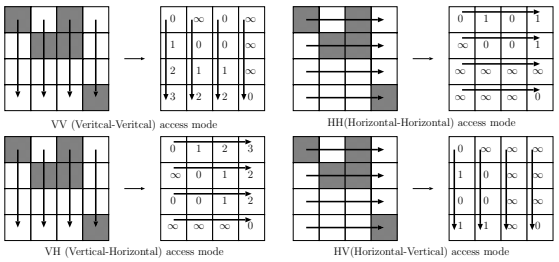



Figure 11. Access modes for Step 1

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)
$$d_{i,j} = min\{|k-j| \mid b_{i,k} = 1, 1 \leq k \leq n\}$$
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 **VV-HH**, **HH-VV**, **VH-VH**, and **HV-HV**, because the access mode satisfies 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 VV-HH, HH-VV, VH-VH, and HV-HV.

### B. Implementations with Different Access Modes

In our previous work [15], we have implemented our previous proposed parallel EDM algorithm with the above four access modes. Also, we have evaluated our proposed parallel EDM algorithm with Tesla C1060 [12] which consists of 240 Streaming Processor Cores and 4GB global memory. The experimental result shown in [15], the performance of VH-VH access mode was better than the other access modes. This is because in VH-VH access mode, the GPU implementation can benefit from coalescing access to the global memory significantly.

## VI. New Implementation of EDM Algorithm on GPUs

In this section, we show a new implementation of EDM Algorithm on GPUs.

*1) New Access Mode with Efficient Memory Access:* As we see in above section, VH-VH access mode can obtain the best performance of four access modes. Therefore it is clear that coalescing access to global memory plays an important role in our GPU implementations. However, VH-VH access mode cannot fully benefit from coalescing access because its memory writing does not support coalescing access. Therefore, in this subsection, we show a new implementation of the proposed algorithm which can fully utilize the coalescing in each implementing step in memory read and write. We call the access mode of the new implementation as *VTV-VTV access mode* (VTV stands for *Vertical-Transpose-Vertical*). To keep two conditions as shown in the previous section, following operations are performed in each step; (i) An input data is read from global memory with coalesced read. (ii) The results are transposed with shared memory. (iii) The transposed results are written into the global memory with coalesced write. More specifically, in the new access mode of Step 1, the 2-D array of the input image will be read in column wise by each thread. After processing, the results will be transposed using shared memory. Then the transposed data is written into another array in column wise by each thread as the results of Step 1 and the input data of Step 2. In the new implementation of Step 2, the 2-D extra array which contains the results of Step 1 will be read in column wise by each thread. After reading data from the 2-D extra array, the results of Step 2 will be transposed using shared memory. Then the transposed results are written into the extra array column by column by each thread. It

is clear that, in VTV-VTV access mode, each step can be implemented with full coalescing.

### A. GPU implementation for VTV-VTV access mode

Here we use Tesla C1060 as the experimental system to describe the new implementation of the proposed algorithm. In Tesla C1060 supported CUDA architecture, the maximum size of shared memory per thread block is 16 Kbytes. It is clear that shared memory can contain up to 4096 32-bit unsigned integers. Binary values of input image are stored in an array named $A$ in global memory. The size of input image is $n \times n$ and $n$ is integral multiple of 512.

We now show the new implementation of Step 1 of the proposed parallel EDM algorithm. First we create a kernel named *Kernel_UpBottom* to implement the up-to-bottom process of Step 1. The *Kernel_UpBottom* will be launched by a 1-D grid. Thread blocks of the grid is also set as one dimensional. The number of available threads in a thread block is configured as 512 (maximum number of available threads). It means that the number of thread blocks is $n/512$. Each thread block allocate a 2-D array with size of $512 \times (4096/512)$ in the shared memory. Now we partition the input image into $n/512$ subimages along with column wise and assign each thread block to each subimage. Each thread of a thread block processes each column of corresponding subimage. We further divide each subimage into $n/(4096/512)$ tiles along with row wise and copy each tile into the shared memory array orderly using threads of corresponding thread block. The copy must be performed in coalescing. The width of each tile is 512 and the height is $4096/512$.

To simplify the description, we use two indices $i$ and $i+1$ to designate two successive pixels in a column, and another two indices $j$ and $j+1$ to designate two successive tiles in a subimage. First, elements of tile$_j$ are copied to the shared memory array by corresponding thread block with the coalescing. Each thread of the thread block travels each corresponding column of tile$_j$, which is stored in shared memory, from up to bottom to compute the distance value of each pixel. The computed distance values also are kept in the shared memory array. Namely, the computed distance value of a pixel will cover the value of the pixel in the shared memory array. After all distance values are computed, the corresponding thread block copy them from shared memory to an extra array (located in global memory ) named $B$. In a column of the input image, the distance value of pixel$_{i+1}$ must depend on the distance value of pixel$_i$. Therefore, the distance values of first row of tile$_{j+1}$ must depend on the distance values of last row of tile$_j$. For each column of tile$_j$, its corresponding thread will use a register to keep the distance value of last pixel. Now let the thread block copy tile$_{j+1}$ to the shared memory array with the coalescing. For each column of tile$_{j+1}$ , each thread uses the value of register to compute the distance value of first pixel. Same to the computation of the distance values of tile$_j$, we can compute the distance values of other pixels of tile$_{j+1}$ from up to bottom. For each column of tile$_{j+1}$, each thread also uses the register to keep the distance value of last pixel. In the same way, the distance values of other tiles can be computed. The implementation of the up-to-bottom process is finished.

The implementation of the bottom-to-up process is similar with the implementation of the up-to-bottom process. The computed distance values of the bottom-to-up process are stored in another extra array named $C$. In real implementation, the bottom-to-up process is also implemented in *Kernel_UpBottom*.

Now we need to select minimum values from corresponding elements of two extra arrays $B$ and $C$ as final result of Step 1 and store the selected minimum values into array $A$. Here, in order to write the selected minimum values into array $A$ with coalescing, we use shared memory to implement a simple transpose and, after the transpose, the selected minimum values can be easily written into array $A$ with coalescing. The details of the implementation are given below.

We create a new kernel named *Kernel_MinSelec_Transpose* to implement the selection of minimum values. The *Kernel_MinSelec_Transpose* will be launched by a 2-D grid with $(n/32) \times (n/32)$ thread blocks. Each thread block is set as two dimensional and the number of available threads in a thread block is configured as $32 \times 4$. A 2-D array with size of $32 \times 32$ need to be allocated in shared memory. It is clear that, in each extra array, the size of each tile is $32 \times 32$. Each thread block will be assigned to each corresponding tile. All threads of a thread block read corresponding tile of extra arrays $B$ and $C$ row by row and select the minimum values, then write the selected minimum values into the shared memory array column by column. All threads of each thread block again read the values of the shared memory array row by row and write them into a tile of array $A$ in coalescing. We refer the reader to Figure 12 for an illustration. In the figure, $T_i$ represents a thread and each arrow represents data access of each thread. In this case, the selected minimum

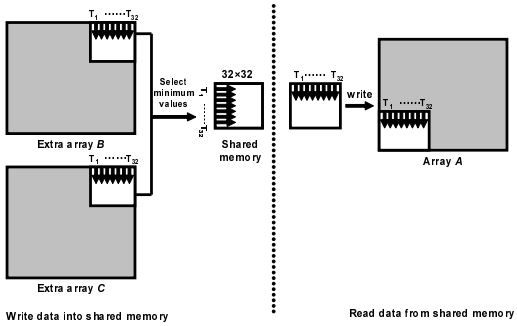



Figure 12. VTV access mode in Step 1 with coalescing access

values are written into array $A$ with coalescing. Since there is no performance penalty for noncontiguous access patterns in shared memory.

However, in above implementation, the use of shared memory will result in another problem, shared memory bank conflicts. As given above, the size of shared memory array is $32 \times 32$. It means one column of this array is mapped into the same bank of shared memory. Since 32 is integral multiple of 16 and there are 16 banks in shared memory of Tesla C1060 system. When threads of a thread block access to a column of the shared memory array, the bank conflicts will occur. A simple method to free this shared memory bank conflicts is add a new column to the shared memory array. As shown in Figure 13, after adding a new column to the shared memory array, elements of each column are mapped into different banks.

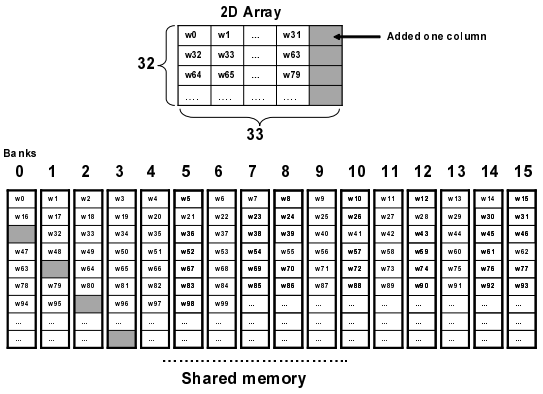



Figure 13. Bank conflict free map

Now we consider the partition camping of global memory. In Tesla C1060 system, global memory is divided into 8 partitions of 256-byte width and all data in strides of 2048 bytes will map into the same partition. Therefore in each extra array, all elements which belong to same column will be mapped into a single partition. In our implementation, each thread block will be assigned to each tile of an extra array according to general Cartesian coordinate interpretation, as shown in Figure 14. In the figure, $\mathrm{B}_{i,j}$ represents the assigned thread block of each tile. The figure also shows that, when the selected minimum values are written into array $A$, the several concurrent thread blocks need to access columns of the array $A$ which are located in the same partition of global memory. It will result in the partition camping.

In order to avoid the partition camping, we can use Diagonal coordinate to interpret the coordinate of each tile. More details about the partition camping, the readers can find in the reference paper [17].

Now we show the new implementation of Step 2 of the proposed parallel EDM algorithm. From the description of the parallel EDM algorithm we know that, each column of

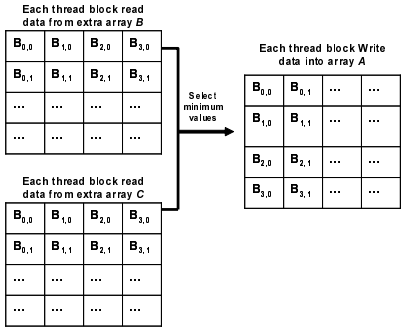



Figure 14. Write array $A$ with partion camping

the array $A$ needs a stack to keep the corresponding proximate points. We allocate a 2-D array named $Stacks$ in global memory. Each column of the 2-D array is used as the stack for each corresponding column of the array $A$. We create a new kernel named *Kernel_BoundaryPoint* to implement Step 2 of the proposed algorithm. The *Kernel_BoundaryPoint* will be launched by a 1-D grid and thread blocks of the grid is also set as one dimensional. The number of available threads in a thread block is configured as 512. It means the number of thread blocks is $n/512$. Each thread block allocate a 2-D array with size of $512 \times (4096/512)$ in shared memory. The array $A$ is also partitioned into $n/512$ partitions and each partition is divided into $n/(4096/512)$ tiles. Each thread block will be assigned to each corresponding partition. Each tile of a partition will be copied into the shared memory array in order by threads of the assigned thread block. It is clear that the copy is implemented in coalescing. Each thread of a thread block uses the data of shared memory to operate on the corresponding stack to obtain the proximate points. Since each stack is kept in the corresponding column of array $Stacks$, therefore, if the sizes of several stacks are same, the access to stacks by the corresponding threads also can benefit from the part coalescing.

Creating the stack for each column of the array $A$ to keep the proximate points, boundary points of each pair of adjacent proximate points need to be obtained. We store the obtained boundary points into a 2-D array named $BoundaryArray$. The space of the array $BoundaryArray$ is allocated from global memory. Each column of array $BoundaryArray$ is correspond to each column of the array $Stacks$. If we use push-pop operations to obtain each pair of proximate points from a stack, it just can benefit from the part coalescing. To fully utilize the coalescing, we take each stack as a 1-D array and access it from bottom position to obtain each pair of the proximate points. Obviously, boundary points kept in one column of the array $BoundaryArray$ are obtained for one row of the input image. By comparing $x$ coordinate of each pixel and boundary points, the closest black pixel to the pixel can be obtained easily. Now we

Table I
PERFORMANCE OF IMPLEMENTATION WITH VH-VH AND VTV-VTV ACCESS MODE ON DIFFERENT GPU SYSTEMS ($n$=9216)

(a) Tesla C1060

| | CPU | VH-VH access mode | | VTV-VTV access mode | |
|---|---|---|---|---|---|
| | Time[ms] | Time[ms] | Speed-up | Time[ms] | Speed-up |
| Step1 | 1712.00 | 145.37 | 11.77 | 38.97 | 43.92 |
| Step2 | 6957.13 | 794.54 | 8.75 | 413.49 | 16.82 |
| Total | 8669.13 | 939.91 | 9.22 | 452.46 | 19.16 |

(b) GTX480

| | CPU | VH-VH access mode | | VTV-VTV access mode | |
|---|---|---|---|---|---|
| | Time[ms] | Time[ms] | Speed-up | Time[ms] | Speed-up |
| Step1 | 1712.00 | 86.01 | 19.90 | 19.67 | 87.05 |
| Step2 | 6957.13 | 241.11 | 28.81 | 145.91 | 47.68 |
| Total | 8669.13 | 327.45 | 26.47 | 165.58 | 52.36 |

assume that the $\text{pixel}_{i+1}$ and $\text{pixel}_i$ are successive two pixels of a row in the input image. From the proximate point theorem we can easy to know that, $x$ coordinate of the closest black pixel of $\text{pixel}_i$ must be smaller than $x$ coordinate of the closest black pixel of $\text{pixel}_{i+1}$. It means that each element of array $BoundaryArray$ just need to be visited once. This can reduce the access times of global memory from $O(n^2)$ to $O(n)$. In final, the distance value of each pixel is calculated and the result is stored to array $A$.

Table I shows that the performance of the new implementation on the different GPU systems. For processing the image with size of $9216 \times 9216$, our new implementation using VTV-VTV access mode can achieve 19 and 52 times speedup on Tesla C1060 and GTX 480 system respectively over the performance of the sequential algorithm implementation. The experimental results also show that, even including data transfer time between CPU and GPU, our new implementation also can achieve about 10 and 30 times speedup on Tesla C1060 and GTX 480 system respectively.

## VII. CONCLUSIONS

In this paper, we have proposed a simple parallel algorithm for the EDM and shown an intuitive GPU implementation of the proposed algorithm. In that of GPU implementation, we have considered many programming issues of the GPU system such as coalescing access of global memory, shared memory bank conflicts and partition camping. We have implemented our parallel algorithm 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 $9216 \times 9216$, our implementation can achieve a speedup factor of 52 over the sequential algorithm implementation.

## REFERENCES


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

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

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

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

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

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

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

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

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

[10] 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.

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

[12] NVIDIA_Corporation., *Tesla C1060 Computing Processor*, http://www.nvidia.com/object/product_tesla_c1060_us.html.

[13] NVIDIA_Corporation, *GetForce GTX 480*, http://www.nvidia.com/object/product_getforce_gtx_480_us.html.

[14] F. Preparata and M. Shamos, *Computational Geometry: An Introduction*. Springer-Verlag, 1990.

[15] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of parallel computation of Euclidean distance map in multicore processors and GPUs," in *Proc. of International Conference on Networking and Computing*, 2010, pp. 120–127.

[16] NVIDIA_Corporation., *CUDA Architecture*, http://www.nvidia.com/object/cuda_home_new.html.

[17] G. Ruetsch and P. Micikevicius, *Optimizing Matrix Transpose in CUDA*, NVIDIA CUDA SDK, January 2009.