The International Journal of Parallel, Emergent and Distributed Systems
Vol. 00, No. 00, Month 2011, 1–23

RESEARCH ARTICLE

Accelerating Computation of Euclidean Distance Map using the GPU with Efficient Memory Access

Duhu Man, Kenji Uda, Yasuaki Ito, and Koji Nakano*

Department of Information Engineering, Hiroshima University,
Higashi-Hiroshima, Japan

(*Received 00 Month 200x; in final form 00 Month 200x*)

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-dimensional binary image, EDM is a 2-dimensional 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 coalesced access of global memory and shared memory bank conflicts. To be concrete, transposing 2-dimensional arrays, which are temporal data stored in the global memory, with the shared memory, the main access from/to the global memory enables to be performed by coalesced access. In practice, we have implemented our parallel algorithm in the following three modern GPU systems: Tesla C1060, GTX 480 and GTX 580, respectively. The experimental results have shown that, for an input binary image with size of 9216×9216, our implementation can achieve a speedup factor of 54 over the sequential algorithm implementation.

Keywords: Euclidean distance map; proximate points; GPU; coalesced memory access; bank conflict; CUDA

1. Introduction

Recent Graphics Processing Units (GPUs), which have a lot of processing units, can be used for general purpose parallel computation. Since GPUs have very high memory bandwidth, the performance of GPUs greatly depends on memory access. CUDA (Compute Unified Device Architecture) [1] is the architecture for general purpose parallel computation on GPUs. Using CUDA, we can develop parallel algorithms to be implemented in GPUs. Therefore, many studies have been devoted to implement parallel algorithms using CUDA [2–10].

In many applications of image processing such as blurring effects, skeletonizing and matching, it is essential to measure distances between featured pixels and non-featured pixels. For a 2-dimensional 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 readers to Figure 1 for an illustration of Euclidean Distance

*Corresponding author. Email: nakano@cs.hiroshima-u.ac.jp

ISSN: 1744-5760 print/ISSN 1744-5779 online

DOI: 10.1080/17445760.YYYY.CATSid
http://www.informaworld.com

Map. Assuming that the 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 $d(p, q) = \sqrt{(x(p) - x(q))^2 + (y(p) - y(q))^2}$.

$\sqrt{10}$	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. Euclidean Distance Map

As is known to us, the computing time is an important issue in the real-time image processing, especially for images with large size. For example, the real-time image processing is the main part of many industrial applications such as the vision-guided robot bin-picking system etc. Actually the vision-guided robot bin-picking is one of the systems with highest interest of the industry. In order to positioning bins precisely, bins with markers can be used. Especially, circle markers are used for robot vision [11] since a circle must be seen as an ellipse from any angle. Thus, a fast and reliable ellipse detection algorithm is needed. The Euclidean distance transform can be used for the evaluation of the estimated ellipses in real time [12]. Therefore we also need a faster algorithm to implement the Euclidean distance transform.

Many algorithms for computing EDM have been proposed in the past, such as sequential algorithm [13–16] and parallel algorithm [17–19]. Breu *et al.* [13] and Chen *et al.* [14, 15] have presented $O(n^2)$-time sequential algorithm for computing 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 [16] 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 model. Lee *et al.* [20] presented an $O(\log^2 n)$-time algorithm using n^2 processors on the EREW PRAM. Pavel and Akl [19] 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.* [17] 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.* [18] 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.* [21] have proposed two parallel algorithms for EDM on Linear Array with Reconfigurable Pipeline Bus System [22]. Their first algorithm can computes EDM

in $O(\frac{\log n \log\log n}{\log\log\log n})$ time using n^2 processors and the 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 coalesced access of global memory and shared memory bank conflicts [23]. Coalesced access is necessary 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 or 32 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.

In our previous paper [5], we have shown an optimal parallel algorithm for computing Euclidean Distance Map (EDM) of a 2-dimensional 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 10000×10000, 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 5 times speedup in that of GPU system. However, it is not enough to cope with the above programming issues. Especially, in our implementation, 2-dimensional arrays are mainly accessed from/to the global memory four times. However, two times of them cannot reap the benefit of the coalesced access.

The main contribution of this paper is to show an improved GPU implementation of the algorithm with more efficient memory access. In our new implementation, we have considered programming issues of the GPU system such as coalesced access for global memory and shared memory bank conflicts. The new idea of our implementation is that we have improved the access for 2-dimensional arrays that are temporal data stored in the global memory which cannot be done with coalesced access in the previous implementation. To be concrete, transposing the 2-dimensional arrays with the shared memory, the access enables to be performed by coalesced access. We have implemented and evaluated our proposed parallel EDM algorithm in the following three GPU systems, Tesla C1060 [24], GTX 480 [25] and GTX 580 [26], respectively. The experimental results have shown that for an input binary image with size of 9216×9216, our implementation can achieve a speedup factor of 54 over the sequential algorithm implementation. Also, we have presented that the density of black pixels in an input image affects the performance of the proposed GPU implementation.

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-dimensional binary image. Section 4 introduces the features of the GPU system in CUDA. In Section 5, we review our previous GPU implementation. Section 6 exhibits a new GPU implementation considering programming issues for the GPU system. Section 7 shows the performance of the new GPU implementations on different GPU systems. Finally, Section 8 offers concluding remarks.

2. Proximate Points Problem

In this section, we review the proximate problem [18] 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 $(x(p), y(p))$.

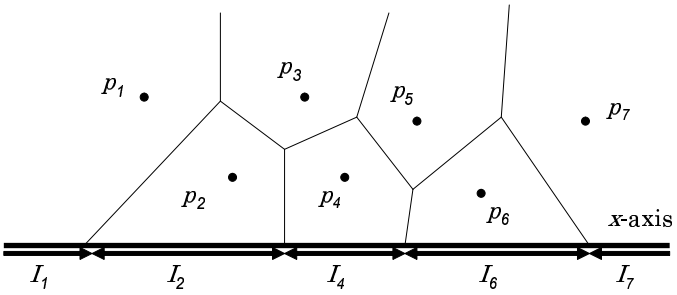


Figure 2. Proximate intervals

Consider a collection $P = \{p_1, p_2, ..., p_n\}$ of n points sorted by x-coordinate, that is, $x(p_1) < x(p_2) < ... < x(p_n)$. 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 [27], 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.

It is also clear that, the boundary of any two points can be computed by $O(1)$ time. For example, as shown in Figure 3, the coordinates of p_i and p_j are given. The coordinates of the midpoint of p_i and p_j can be computed in the formulas: $x_{mid} = \frac{(x_i+x_j)}{2}$ and $y_{mid} = \frac{(y_i+y_j)}{2}$. The slope of the line which crosses the points p_i and p_j can be computed by the formula: $\alpha = \frac{(y_j-y_i)}{(x_j-x_i)}$, here the α represents the slope of the line. Further, the slope of the perpendicular bisector line of p_i and p_j can be computed by the formula: $\beta = -\frac{1}{\alpha} = -\frac{(x_j-x_i)}{(y_j-y_i)}$, here the β represents the slope of the perpendicular bisector line. Finally the perpendicular bisector line of p_i and p_j can be computed by the formula: $y = \beta(x-x_{mid})+y_{mid} = -\frac{(x_j-x_i)}{(y_j-y_i)}(x-\frac{(x_i+x_j)}{2})+\frac{(y_i+y_j)}{2}$. The x-coordinate of the intersection point of the perpendicular bisector line and the x-axis can be obtained as follow: $x_{inter} = \frac{(y_j^2-y_i^2)+(x_j^2-x_i^2)}{2(x_j-x_i)}$. This intersection point is also the boundary point of p_i and p_j. Therefore the coordinate of the boundary point is $(\frac{(y_j^2-y_i^2)+(x_j^2-x_i^2)}{2(x_j-x_i)}, 0)$. The coordinate of the boundary point can

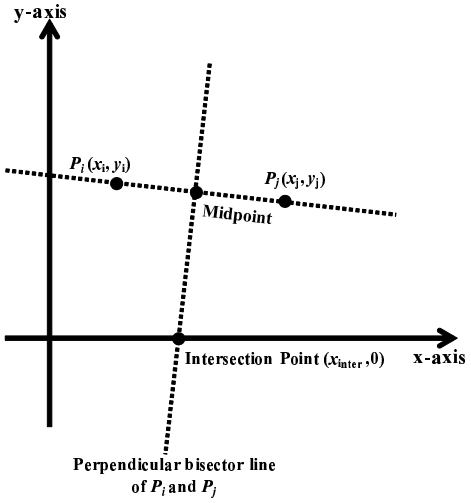


Figure 3. Perpendicular bisector line of two points

be computed in $O(1)$ time using a single processor.

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, therefore 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 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.

LEMMA 2.2. *There exists a unique point of p_i of P such that:*

- *The only proximate points of $P \cup \{p\}$ are $p_1, p_2, ..., p_i, p$.*

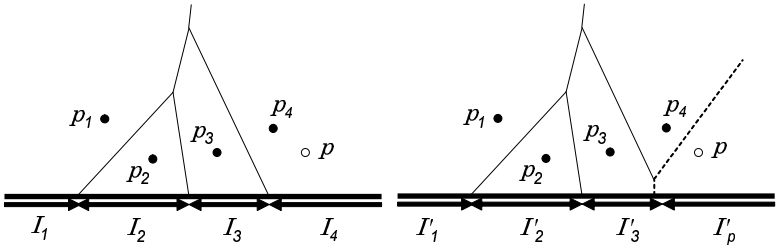
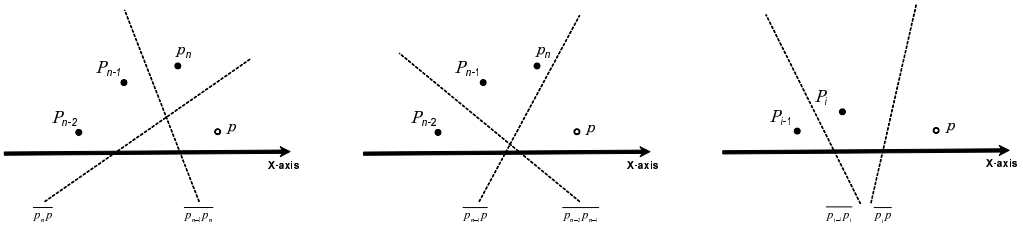


Figure 4. Illustrating the addition of p to $P = \{p_1, p_2, p_3, p_4\}$

- *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.*

We show an intuitive proof of the lemma by geometry. As shown in Figure 5(a), the line $\overline{p_n p}$ and line $\overline{p_{n-1}p_n}$ denote the perpendicular bisector lines of the point pair $\{p_n, p\}$ and the point pair $\{p_{n-1}, p_n\}$. The intersection of $\overline{p_n p}$ and the x-axis is located left to the intersection of $\overline{p_{n-1}p_n}$ and the x-axis. It implies the proximate interval of p_n is empty. Now we draw the perpendicular bisector lines of the point



(a) (b) (c)

Figure 5. Perpendicular bisector lines

pair of $\{p_{n-2}, p_{n-1}\}$ and the point pair of $\{p_{n-1}, p\}$, they are denoted by line $\overline{p_{n-2}p_{n-1}}$ and line $\overline{p_{n-1}p}$, see Figure 5(b). The intersection of $\overline{p_{n-1}p}$ and the x-axis is also located left to the intersection of $\overline{p_{n-2}p_{n-1}}$ and the x-axis. It means the proximate interval of p_{n-1} is also empty. We repeat the procedure until find a point, p_i, $1 < i < n-1$, its proximate interval is nonempty, see Figure 5(c). As shown in the figure, the line $\overline{p_{i-1}p_i}$ denotes the perpendicular bisector line of the point pair of $\{p_{i-1}, p_i\}$ and the line $\overline{p_i p}$ denotes the perpendicular bisector line of the point pair of $\{p_i, p\}$. It is clear that the intersection of $\overline{p_i p}$ and x-axis is located right to the intersection of $\overline{p_{i-1}p_i}$ and x-axis. It means the proximate interval of p is decided. The proximate interval of p_i is adjacent to the proximate interval of p. The intersection of $\overline{p_i p}$ and x-axis is the boundary point of p_i and p. It also imply that the point p can not affect the proximate interval of p_j, where $1 \leq j \leq i-1$.

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 $x(p) < x(p_1) < x(p_2) < ... < x(p_n)$. 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.

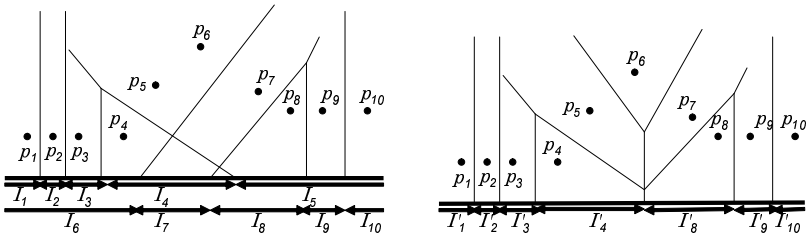
LEMMA 2.4. *There exists a unique pair of proximate points $p_i \in P_L$ and $p_j \in P_R$ such that*

- *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 6 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 6. 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.

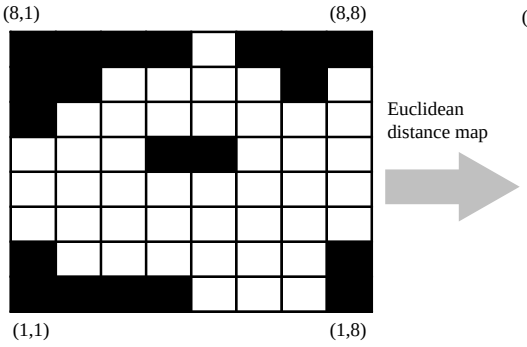
LEMMA 2.5. *The point p_k is not dominated by p_{k-1} and q_k if $2 \leq k \leq i$, and dominated otherwise.*

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. [13] proposed a sequential algorithm that computes the proximate points of an n-point planar set in $O(n)$ time. The algorithm in [13] uses a stack to store the proximate points found.

3. Parallel Euclidean Distance Map of 2-dimensional 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 7(a). In Figure 7(a), each square represents a pixel. For this binary image, its final distance mapping array is shown in Figure 7(b).



(a) Binary image

(8,1) (8,8)

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

(1,1) (1,8)

(b) Mapping array

Figure 7. 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:

vspace2mm **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.

vspace2mm **Step 2** 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. vspace2mm

e 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 compute 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 8(a) (its original input image is shown in Fig 7). Second, each processor PE(i) $(1 \leq i \leq n)$ reads pixel values in the same column from bottom to up to compute that distance, as illustrated in Figure 8(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 $O(n)$.

noindent **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,

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

(a) process with up to bottom

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

(b) process with bottom to up

Figure 8. Process each column with two directions

as illustrated in Figure 9 and Figure 10. In Figure 10, the Voronoi polygons

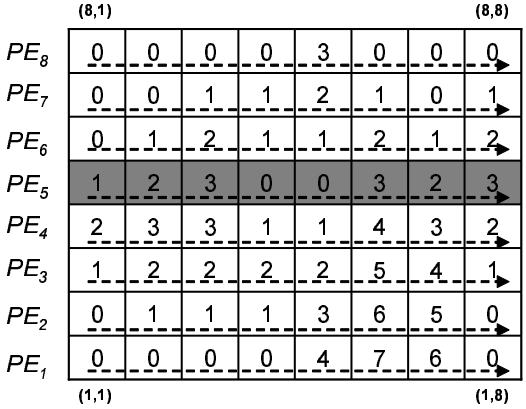


Figure 9. Processing with row wise

correspond to 5th row (shaded row) of the image illustrated in Figure 9. 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.

noindent **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 11 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 the input image. The distance between a given pixel and its closest black pixel is also calculated in the obvious way.

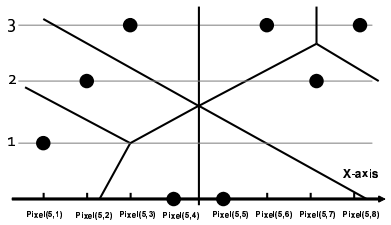


Figure 10. Voronoi polygons

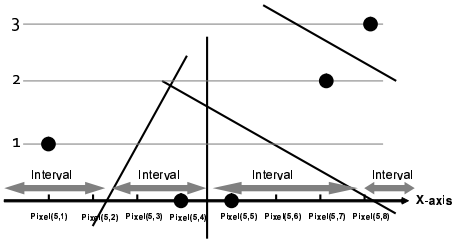


Figure 11. Proximate intervals

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.*

4. Compute Unified Device Architecture (CUDA)

CUDA uses two types of memories in the NVIDIA GPUs: *the global memory* and *the shared memory* [23]. The global memory is implemented as an off-chip DRAM of the GPU, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The efficient usage of the global memory and the shared memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the coalescing* of the global memory access and *the bank conflict* of the shared memory access [5, 28]. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed in the same time. Thus, threads should perform coalesced access when they access to the global memory. Figure 12 illustrates the CUDA hardware architecture.

CUDA parallel programming model has a hierarchy of thread groups called *grid*, *block* and *thread*. A single grid is organized by multiple blocks, each of which has equal number of threads. The blocks are allocated to streaming processors such that all threads in a block are executed by the same streaming processor in parallel. All threads can access to the global memory. However, as we can see in Figure 12, threads in a block can access to the shared memory of the streaming processor to which the block is allocated. Since blocks are arranged to multiple streaming processors, threads in different blocks cannot share data in shared memories.

CUDA C extends C language by allowing the programmer to define C functions, called *kernels*. By invoking a kernel, all blocks in the grid are allocated in streaming processors, and threads in each block are executed by processor cores in a single streaming processor. In the execution, threads in a block are split into groups of thread called *warps*. Each of these warps contains the same number of threads and is execute independently. When a warp is selected for execution, all threads

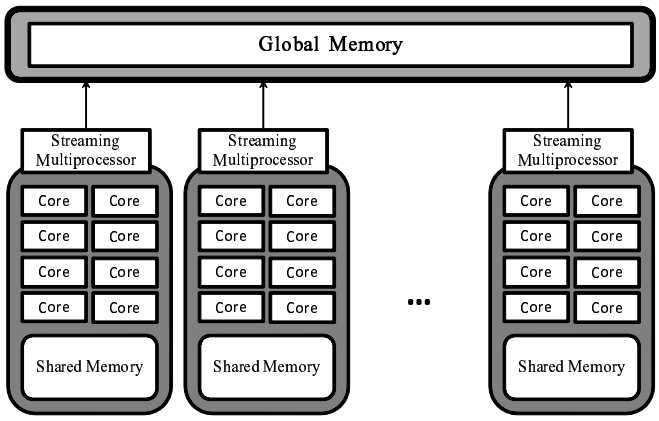


Figure 12. CUDA hardware architecture

includegraphics[scale=0.8]coalescing.eps

Figure 13. Coalesced and stride access

execute the same instruction. Any flow control instruction (e.g. if-statements in C language) can significantly impact the effective instruction throughput by causing threads of the same warp to diverge, that is, to follow different execution paths. If this happens, the different execution paths have to be serialized. When all the different execution paths have completed, the threads back to the same execution path. For example, for an if-else statement, if some threads in a warp take the if-clause and others take the else-clause, both clauses are executed in serial. On the other hand, when all threads in a warp branch in the same direction, all threads in a warp take the if-clause, or all take the else-clause. Therefore, to improve the performance, it is important to make branch behavior of all threads in a warp uniform.

As we have mentioned, the coalesced access to the global memory is a key issue to accelerate the computation. As illustrated in Figure 13, when threads access to continuous locations in a row of a 2-dimensional array (*horizontal access*), the continuous locations in address space of the global memory are accessed in the same time (*coalesced access*). However, if threads access to continuous locations in a column (*vertical access*), the distant locations are accessed in the same time (*stride access*). From the structure of the global memory, the coalesced access maximizes the bandwidth of memory access. On the other hand, the stride access needs a lot of clock cycles. Thus, we should avoid the stride access (or the vertical access) and perform the coalesced access (or the horizontal access) whenever possible.

5. Our Previous Implementation of EDM Algorithm on GPUs

In this section, we show our previous implementation of EDM algorithm on GPUs [5]. We have defined several memory access modes which affect the performance of our algorithm. Using the access modes, we have implemented a parallel EDM algorithm.

5.1 Access Modes

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 14 for illustrating the possible four access modes of Step 1.

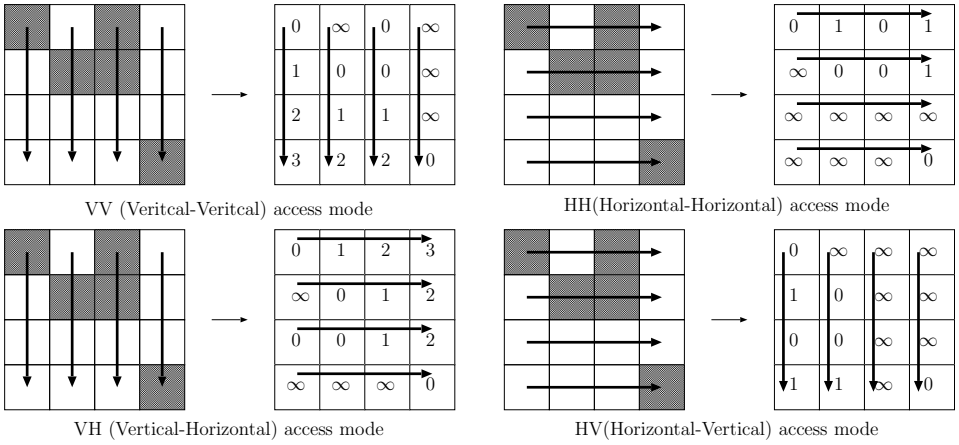


Figure 14. Access modes for Step 1

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 and 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 $r_1 w_1 r_2 w_2$ 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.

5.2 Implementations with Different Access Modes

In our previous work [5], we have implemented our proposed parallel EDM algorithm with the above four access modes. Also, we have evaluated our proposed parallel EDM algorithm with Tesla C1060 [24] which consists of 240 Streaming Processor Cores and 4GB global memory. The experimental result shown in [5], 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 coalesced access to the global memory significantly.

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

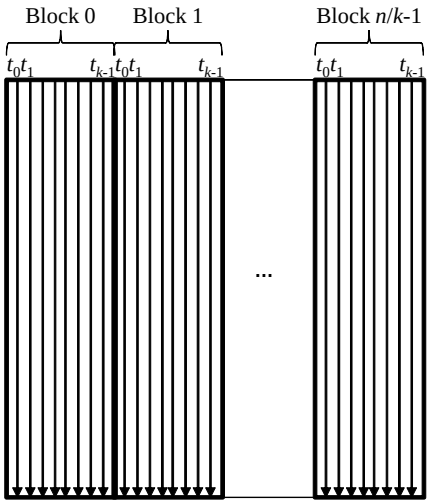


Figure 15. Mapping blocks into subimages

For implementing Step 1 of the algorithm, we partition the original input image of $n \times n$ into $\frac{n}{k}$ subimages along with column wise, where k is the number of threads in one block. We assign $\frac{n}{k}$ blocks are assigned to subimages and each block processes each corresponding subimage independently. Each thread of a block processes each corresponding column of the subimage. We refer readers to Figure 15 as an simple illustration. In Figure 15, each $t_i (0 \leq i \leq k-1)$ represents a thread of a block and each arrow represents an access of a pixel value by one thread. It is clear that, for a subimage, the access to each row can be performed in coalescing.

By following Step 1 of the parallel EDM algorithm, each thread needs to access each pixel value of the corresponding column two times. One is access for computing results of up-to-bottom process and the other is access for computing results of bottom-to-up process. After selecting the minimum value for each pixel, each thread writes the minimum one 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. However, the writing of the extra array cannot benefit from the coalescing at all. On the other hand, in the implementation of VV-HH access mode, the writing of the extra array is also can benefit from the full coalescing. Therefore in VV-HH access mode, the implementation of Step 1 can achieve the most significant performance. Differently, in HH-VV access mode, the whole implementation of Step 1 cannot benefit from the coalescing at all since the read and write operation for the global memory is stride access. Therefore Step 1

of the HH-VV access mode achieved the worst performance.

In Step 2 of the algorithm, stacks are necessary for computing boundary points. Since one stack is used for the computation of each column, n stacks are necessary in total. We allocate a 2-dimensional array in the global memory to the stacks. Each stack is assigned to one column of the 2-dimensional array. Also, each thread reads elements of corresponding column of the extra array, which stores the results of Step 1, to obtain elements of corresponding stack. However the push-pop operations for the stacks are not uniform. Therefore the access of the extra array cannot be performed in full coalescing. In the same way, the access of the stacks also cannot be performed in full coalescing. This is reason that the implementation of Step 2 cannot achieve a significant performance even in HH-VV 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 the closest black pixel. If we assume that the mapping results will be stored in a 2-dimensional array named output array, it needs all threads accesses the output array along with row wise. In other words, each thread accesses the corresponding row of the output array, and it cannot utilize the coalescing. However, in Step 2 of VV-HH access mode, its whole implementation cannot benefit from the coalescing at all. This is the reason that Step 2 of HV-HV access mode can be little faster than Step 2 of VV-HH access mode.

6. New Implementation of EDM Algorithm on GPUs

The main purpose of this section is to show our new implementation of EDM algorithm in the GPU. In the followings, we introduce a new access mode and a new implementation with it.

6.1 New Access Mode with Efficient Memory Access

As we see in the previous section, VH-VH access mode can obtain the best performance of four access modes. Therefore it is clear that coalesced access to global memory plays an important role in our GPU implementations. However, VH-VH access mode cannot fully benefit from coalesced access because its memory writing does not support coalesced 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;

(1) An input data is read from global memory with coalesced read.
(2) The results are transposed with shared memory.
(3) The transposed results are written into the global memory with coalesced write.

More specifically, in the new access mode of Step 1, the 2-dimensional array of the input image is read in column wise by each thread. After processing, the results are transposed using shared memory. 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-dimensional extra array which contains the results of Step 1 is read in column wise by each thread. After reading data from the 2-dimensional extra array, the resulting values of Step 2

are transposed using shared memory. 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.

6.2 GPU Implementation with New Access Mode

We now show the new implementation of Step 1 for VTV-VTV access mode. The results, which are stored to 2-dimensional arrays, of up-to-bottom and bottom-to-up process are obtained by the same manner of the implementation for VH-VH access mode shown in Section 5.2. After that, each resulting 2-dimensional array is divided into subimages whose size is 32×32. One block is assigned to each subimage and each block runs independently.

In each block, the minimum values from corresponding elements in the two 2-dimensional arrays are selected. To obtain the results of Step 1 in VTV-VTV access mode, the minimum ones are transposed. In our proposed implementation, to transpose them, we utilize the shared memory. As shown in Figure 16, the 32 resulting values of up-to-bottom and bottom-to-up process each are read in column wise using coalesced access with 32 threads. The minimum ones are selected and written to the shared memory in column wise. The above read and write operation is executed column by column. After that, the values are written to the corresponding transposed position in the global memory in column wise with coalesced access. Using the shared memory, all the access from/to the global memory can be coalesced.

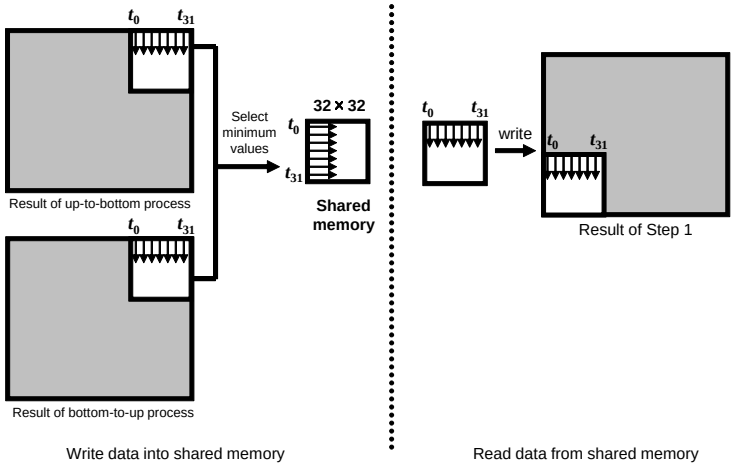


Figure 16. Coalesced Transpose with Shared Memory

However, in the above implementation, the use of shared memory causes another problem, shared memory bank conflicts. As given above, the size of the shared memory array is 32×32. It means that one column of this array is mapped into the same bank of shared memory, since there are 16 or 32 banks in shared memory of CUDA GPU [23]. If multiple threads in a block access to the distinct banks in the shared memory, the access can be serviced simultaneously. On the other hand, if threads access to the same bank, the access has to be serialized. In our implementation, when threads write the minimum values to the shared memory, they write the minimum ones to the same column of the 2-dimensional array in the shared memory (Figure 17(a)). Therefore, bank conflict occurs. To avoid the bank conflict, we add a dummy column to the shared memory array (Figure 17(b)).

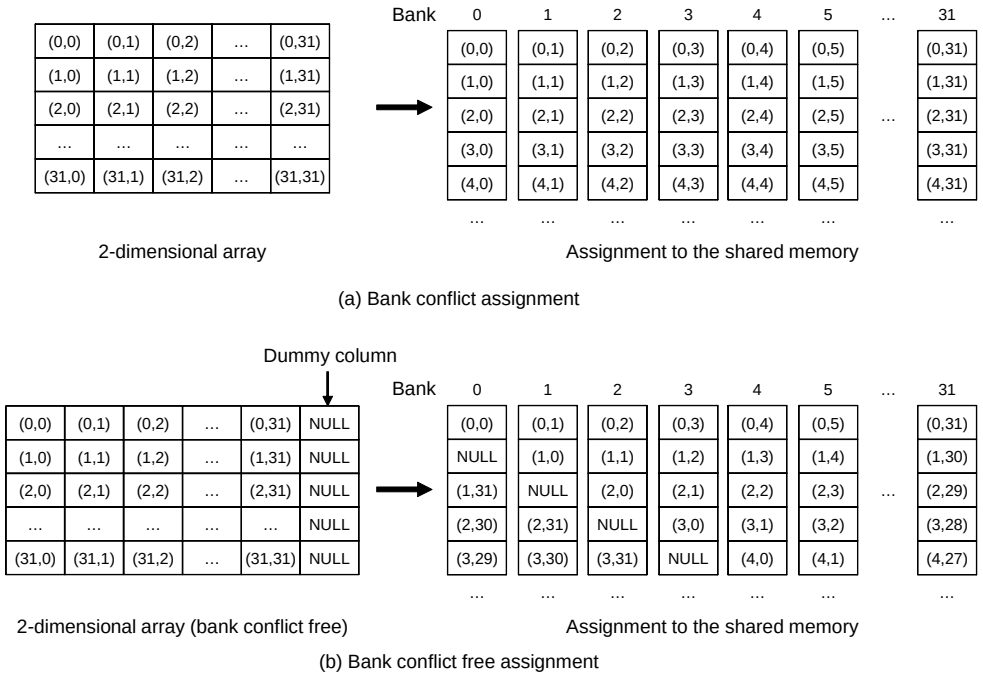


Figure 17. Bank conflict free map

Adding the dummy column, elements of each column are mapped into different banks and all the access in the transposing is free from the bank conflict.

In Step 2 of the implementation, the resulting values are transposed with the shared memory in the same manner as the above.

7. Performance Evaluation

In this section, we show the performance evaluation of the proposed GPU implementation through different experiments. In all the experiments, we have used a binary image of size 9216×9216. Every measurement is the average value of 20 experiments. For all measurements obtained from GPU systems, the variance corresponding to each measurement is always less than 1. For example, the experimental system is GTX 580 and the input image is the Lenna image (see Figure 18), then the variance of the 20 experiments is only 0.64.

Table 1 shows the performance of the new implementation on different GPU systems. For the binary image of Lenna (see Figure 18), our new implementation using VTV-VTV access mode can achieve 20, 46 and 54 times speedup on Tesla C1060, GTX 480 and GTX 580 system respectively, over the performance of the sequential algorithm implemented on a CPU system with Intel Core i7 processor [29]. The experimental results also show that, even if the total computing time includes data transfer time between host memory and global memory, our new implementation also can achieve about 10, 30 and 34 times speedup on Tesla C1060, GTX 480 and GTX 580 system, respectively. The table also show that, the implementation with the VTV-VTV access mode can achieve 1.6x speedup, compared with the implementation with VHVH access mode, in GTX 580 system. However it just achieve 1.4x speedup in Tesla C1060 system. Actually Tesla C1060 only support previous generation CUDA architecture. However GTX 580 can support new generation CUDA architecture, *Fermi* architecture [30] . Compared with the previous

Figure 18. Binary Image of Lenna

Table 1. 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	3956	147	26.9	39	101.4
Step2	7205	621	11.6	508	14.7
Total	11161	768	14.5	547	20.4

(b) GTX 480

	CPU	VH-VH access mode		VTV-VTV access mode	
	Time[ms]	Time[ms]	Speed-up	Time[ms]	Speed-up
Step1	3956	90	43.9	20	197.8
Step2	7205	273	26.3	221	35.4
Total	11161	363	30.7	241	46.0

(c) GTX 580

	CPU	VH-VH access mode		VTV-VTV access mode	
	Time[ms]	Time[ms]	Speed-up	Time[ms]	Speed-up
Step1	3956	93	42.5	16	247.2
Step2	7205	238	30.2	190	39.1
Total	11161	331	33.7	206	54.1

generation CUDA architecture, the Fermi architecture introduces several architectural innovations. For example, in the Fermi architecture, at most 512 CUDA cores can be supported, the global memory is featured by L1/L2 caches, the dual warp scheduler is supported, etc. On the other hand, compared with the previous generation CUDA architecture, the number of memory transactions required by a fully coalesced memory access is also reduced in the Fermi architecture. In the previous generation CUDA architecture, a global memory request for a warp is split into two memory requests, one for each half-warp, that are issued independently. It means that, for a warp, it needs at least two memory transactions to access the global memory, even the global memory accesses are coalesced. However, in the Fermi architecture, a global memory request for a warp is issued into one memory transaction, if the global memory accesses are coalesced. This is reason to why the coalesed access of global memory can achieves more speedups in GTX 580.

It should be clear that the execution time depends on contents of the input images. Therefore, we evaluated the performance for the input images that have the different density of black pixels. We generated input images whose black pixels are randomly distributed such that the density of black pixels is varied from 0% to 100%. Figure 19 shows the performance of the GPU implementation with two different access modes on the GTX 580 system. From the figure, the GPU implementation with VTV-VTV access mode can achieve a higher performance than that with VH-VH access mode for each density of black pixels. The reason is that more global memory accesses can be coalesced in VTV-VTV access mode.

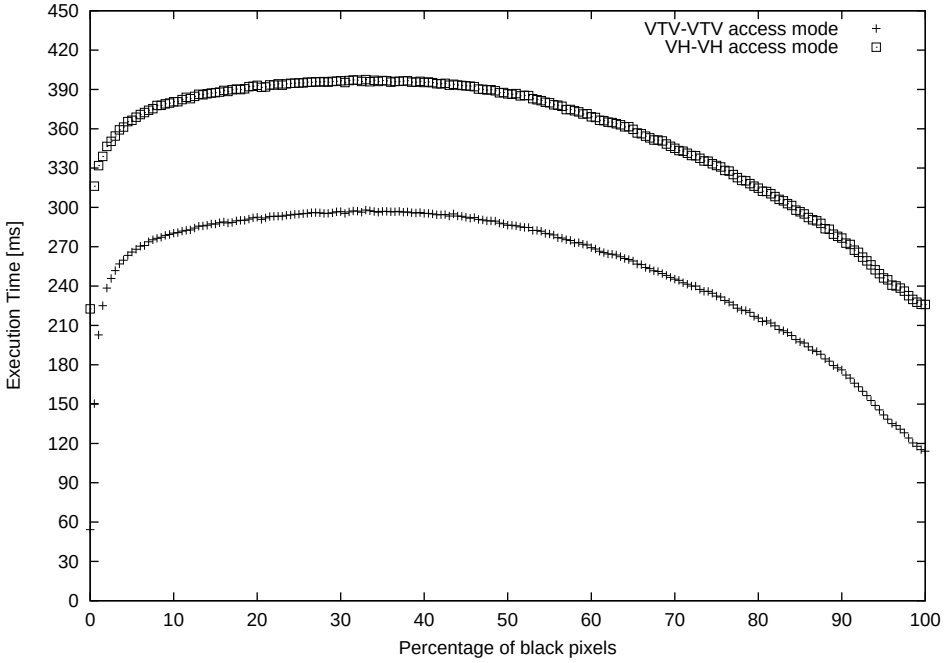


Figure 19. Performance of the GPU implementation with different access modes

Figure 19 also shows how the performance of the GPU implementation is affected by the density of black pixels in the input image. However, the computing time of Step 1 is independent from contents of the input images. The computing time of Step 2 depends only on the contents. Therefore, we focus on the behavior in Step 2. If the density of black pixels is small, pixels of input image have the common nearest black pixel. In other words, each of the black pixels dominates relative large area of the input image. Therefore, the behavior of the threads in each warp is almost the same and computing time becomes shorter. According to the figure, when the percentage of black pixels is close to about 40%, the proposed GPU implementation achieves the worst performance. When the density is the above, many of pixels of input image do not have the common nearest black pixel. Therefore, the behavior of the threads in each warp differs and it causes worse performance. On the other hand, when the percentage of black pixels is larger than 40%, the execution time of the GPU implementation is decreasing along the increase of the percentage of black pixels. The behavior of the threads in each warp is almost the same, which is similar to the lower density of black pixels. Therefore, better performance is achieved. Especially, if the density is close to the 100%, that is almost all the pixels are black, access of stacks assigned to threads in a warp is almost identical. Namely, all the access to the global memory over the whole process reaps the benefit of coalesced access.

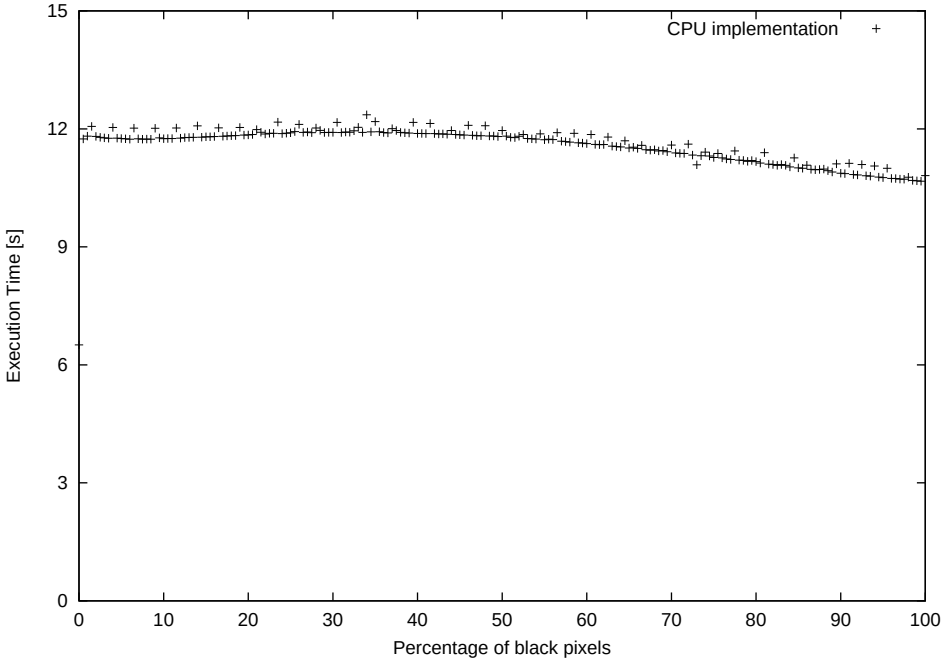


Figure 20. Performance of CPU implementation with HV-HV access mode

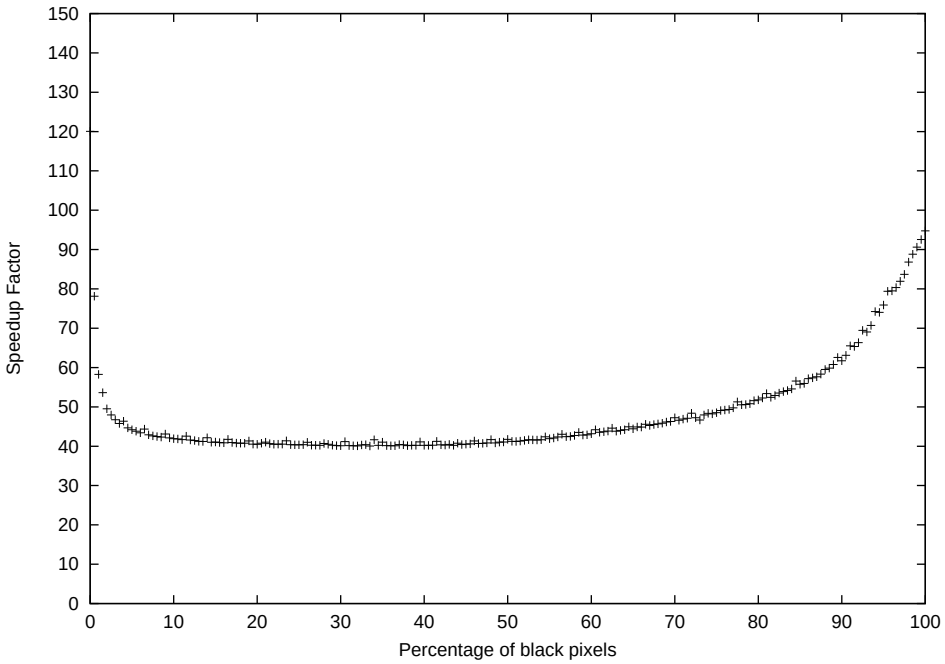


Figure 21. Speedup factor of GPU implementation compared with CPU implementation

Figure 20 shows the performance of the CPU implementation of the sequential algorithm on images with different percentage of randomly distributed black pixels. In our previous paper [5], we have shown that the CPU implementation can achieve the best performance in HV-HV access mode. Therefore, we only show the performance of the CPU implementation with HV-HV access mode. In the figure, it is clear that the density of black pixels has no significant effect on the performance of the CPU implementation. Figure 21 shows the speedup factor of

the GPU implementation with VTV-VTV access mode, compared with the CPU implementation. From the figure, for the input images with different percentage of randomly distributed black pixels, our proposed GPU implementation can achieve at least 40 times speedup compared with the optimal CPU implementation.

On the other hand, experiments show that, the uniform distribution of black pixels (see Figure 22) will result in the worst performance. Since the uniform distribution of black pixels will bring a more complicated global memory access on GPUs. Therefore, in this paper, we just show the performance of the uniform distribution.

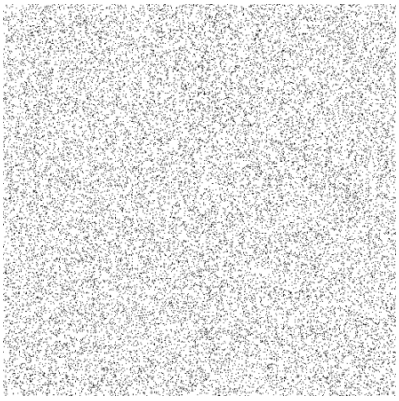


Figure 22. Uniform distribution with 10% black pixels

8. Conclusions

In this paper, we have proposed a simple parallel algorithm for the Euclidean distance map and shown an intuitive GPU implementation of the proposed algorithm. In the GPU implementation, we have considered many programming issues of the GPU system such as coalesced access of global memory and shared memory bank conflicts. We have implemented our parallel algorithm in the following three modern GPU systems: Tesla C1060, GTX 480 and GTX 580, respectively. The experimental results have shown that, for an input binary image with size of 9216×9216, our implementation can achieve a speedup factor of 54 over the sequential algorithm implementation. On the other hand, we have also presented that the density of black pixels in an input image affects the performance of the proposed GPU implementation.

Appendix A.

In the appendix, we show the symbols and definitions appeared in Section 2 and Section 3.

References

[1] NVIDIA Corp., *CUDA ZONE* http://developer.nvidia.com/category/zone/cuda-zone.

Table A1. Symbols appeared in Section 2

Symbol	Definition
p	a point represented by its Cartesian coordinates $x(p)$ and $y(p)$
P	a set of $2n$ points $\{p_1, p_2, ..., p_{2n}\}$ in the plane
P_L	a subset $\{p_1, p_2, ..., p_n\}$ of P
P_R	a subset $\{p_{n+1}, p_{n+2}, ..., p_{2n}\}$ of P
p_i	a point of the point set P
$V(i)$	the voronoi polygen of p_i
I_i	the proximate interval of p_i
I	the collection of proximate intervals : $\{I_1, I_2, ..., I_n\}$
$d(p_i, p_j)$	the Euclidean distance between point p_i and p_j

Table A2. Symbols appeared in Section 3

Symbol	Definition
I	a binary image
n	the number of columns (or rows) of the binary image
$b_{i,j}$	the value of pixel(i, j)
$d((i, j), (i', j'))$	the Euclidean distance between pixels (i, j) and (i', j')
$d_{i,j}$	the distance to the nearest black pixel in the same column
P_j	for a row of image I, the collection of nearest black pixels
$PE(i)$	the processor i

[2] R. Farivar, D. Rebolledo, E. Chan, and R.H. Campbell, *A Parallel Implementation of K-Means Clustering on GPUs*, in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, July, 2008, pp. 340–345.

[3] P. Harish and P.J. Narayanan, *Accelerating large graph algorithms on the GPU using CUDA*, in *Proceedings of the 14th International Conference on High Performance Computing*, 2007, pp. 197–208.

[4] Y. Ito, K. Ogawa, and K. Nakano, *Fast Ellipse Detection Algorithm using Hough Transform on the GPU*, in *Proceedings of International Workshop on Challenges on Massively Parallel Processors*, 2011, pp. 313–319.

[5] 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 1 (2011), pp. 260–276.

[6] K. Nishida, Y. Ito, and K. Nakano, *Accelerating the Dynamic Programming for the Matrix Chain Product on the GPU*, in *Proceedings of International Workshop on Challenges on Massively Parallel Processors*, 2011, pp. 320–326.

[7] K. Ogawa, Y. Ito, and K. Nakano, *Efficient Canny Edge Detection Using a GPU*, in *Proceedings of International Workshop on Advances in Networking and Computing*, 2010, pp. 279–280.

[8] A. Uchida, Y. Ito, and K. Nakano, *Fast and Accurate Template Matching using Pixel Rearrangement on the GPU*, in *Proceedings of International Conference on Networking and Computing*, 2011, pp. 153–159.

[9] S. Wang, S. Cheng, and Q. Wu, *A Parallel Decoding Algorithm of LDPC Codes using CUDA*, in *Proceedings of Asilomar Conference on Signals, Systems, and Computers*, October, 2008, pp. 171–175.

[10] Z. Wei and J. JaJa, *Optimization of Linked List Prefix Computations on Multithreaded GPUs Using CUDA*, in *Proceedings of International Parallel and Distributed Processing Symposium*, 2010.

[11] Y. Mochizuki, A. Imiya, and A. Torii, *Circle-marker detection method for omnidirectional images and its application to robot positioning*, in *Proc. of International Conference on Computer Vison*, 2007, pp. 1–8.

[12] Y. Ito, K. Ogawa, and K. Nakano, *Fast Ellipse Detection Algorithm using Hough Transform on the GPU*, in *Proc. of International Workshop on Challenges on Massively Parallel Processors (CMPP)*, December, 2011, pp. 313–319.

[13] H. Breu, J. Gil, D. Kirkpatrick, and M. Werman, *Linear time Euclidean distance transform algorithms*, IEEE Trans. Pattern Analysis and Machine Intelligence 17 (1995), pp. 529–533.

[14] L. Chen, *Optimal algorithm for complete Euclidean distance transform*, Chinese J. Computers 18 (1995), pp. 611–616.

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

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

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

[18] T. Hayashi, K. Nakano, and S. Olariu, *Optimal parallel algorithm for finding proximate points, with applications*, IEEE Transactions on Parallel and Distributed Systems 9 (1998), pp. 1153–1166.

[19] S. Pavel and S. Akl, *Efficient algorithms for the Euclidean distance transform*, Parallel Processing Letters 5 (1995), pp. 205–212.

[20] Y.H. Lee, S.J. Horng, T.W. Kao, F.S. Jaung, Y.J. Chen, and H.R. Tsai, *Parallel computation of exact Euclidean distance transform*, Parallel Computing 22 (1996), pp. 311–325.

[21] L. Chen, P. Yi, C. Yixin, and X. Xiaohua, *Efficient parallel algorithms for Euclidean distance transform*, The Computer Journal 47 (2004), pp. 694–700.

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

[23] NVIDIA Corp., *NVIDIA CUDA PROGRAMMING GUIDE VERSION 4.1* (2011).

[24] NVIDIA Corp., *Tesla C1060 Computing Processor*, http://www.nvidia.com/object/product_tesla_c1060_us.html.

[25] NVIDIA Corp., *GeForce GTX 480*, http://www.nvidia.com/object/product_geforce_gtx_480_us.html.

[26] NVIDIA Corp., *GeForce GTX 580*, http://www.nvidia.com/object/product-geforce-gtx-580-us.html.

[27] F. Preparata and M. Shamos, *Computational Geometry: An Introduction*, third corrected printing ed., Berlin: Springer-Verlag, 1990.

[28] NVIDIA Corp., *CUDA C BEST PRACTICE GUIDE VERSION 4.1* (2012).

[29] Intel Corp., *INTEL CORE i7 PROCESSOR*, http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html.

[30] NVIDIA Corp., *Fermi White Paper*, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.