Multiple Bounding Boxes Algorithm in Collision Detection and Its Performances in Sequential VS CUDA Parallel Processing

Min Qi
University of Denver

Follow this and additional works at: https://digitalcommons.du.edu/etd

Recommended Citation
Qi, Min, "Multiple Bounding Boxes Algorithm in Collision Detection and Its Performances in Sequential VS CUDA Parallel Processing" (2013). Electronic Theses and Dissertations. 529.
https://digitalcommons.du.edu/etd/529
MULTIPLE BOUNDING BOXES ALGORITHM
IN COLLISION DETECTION AND ITS PERFORMANCES
IN SEQUENTIAL VS CUDA PARALLEL PROCESSING

A Thesis
Presented to
the Faculty of the Daniel Felix Ritchie Engineering and Computer Science
University of Denver

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Min Qi
August 2013

Advisors: Dr. Mario Lopez, Dr. Chris GauthierDickey
Abstract

The traditional method for detecting collisions in a 2D computer game uses a axis-aligned bounding box around each sprite, and checks to determine if the bounding boxes overlap periodically. Using this single bounding box method may result in a large amount of pixel intersection tests, since a sprite may be composed of areas where the pixels are empty and the intersecting bounding box test results in false positives.

Our algorithm analysis shows that the optimal two or three bounding boxes is the best partition we can get for a reasonable time complexity. The results further show significantly diminishing returns for calculating four bounding boxes and above, since it takes a comparably large amount of calculation to find the optimal four bounding boxes or more.

We present a multiple bounding boxes algorithm to show that multiple bounding boxes outperforms the traditional single bounding box method. In addition, we implement the simulation test for the 2-bounding-boxes and 3-bounding-boxes both in serial processing and CUDA parallel processing.

Our simulation result shows that out of the 1,000,000 tests we run, both the 2-bounding-boxes and 3-bounding-boxes partition resulted in far fewer pixel-checks compared to the one bounding box method. Our experiment in serial processing and CUDA parallel processing also shows that GPU programming can be used to significantly reduce the time in calculation and to speed up the process of collision detection when we have a large enough number of pixels to process.
Acknowledgements

First, I would like to thank my advisors Dr. Mario Lopez and Dr. Chris GauthierDickey for the support and guidance you’ve given me over the years. They inspired me in the research of algorithms and analysis. There’s no doubt in my mind that I would not have finished this thesis without your support.

I also would like to thank my committee members, Dr. Nathan Sturtevant, Dr. Chris GauthierDickey and Dr. Mario Lopez for taking the time to review my dissertation and take part in my defense. I could not have asked for a more knowledgeable set of people in my field to review my work.

Special thanks to David Flores. The advice you provided in CUDA implementation is more than helpful, I won’t be able to finish the parallel part of my work so fast without your help.

To my friends Mohamed Al-Bow, Xiao Yin and Ioannis Samonas. You’ve offered a lot of help during the years, and I enjoyed the time that we discussed interesting research problems as well as career problems.

Finally, thank you to my parents for your motivation and encouragement of pursuing higher education. It’s because of you that I made education and research a big part of my life.
# Table of Contents

1 Introduction 1
   1.1 Problem Context ................................. 1
   1.2 GPU introduction .................................. 3
      1.2.1 Brief GPU history and Its Architecture ............ 3
      1.2.2 CUDA Introduction .................................. 4
   1.3 Prior Work ........................................... 5
      1.3.1 Sweep Line ........................................ 5
      1.3.2 Two Minimum Area Rectangle ......................... 6
   1.4 Thesis Statement ..................................... 8
      1.4.1 Objectives ......................................... 8
      1.4.2 Outline ............................................ 8

2 The Multiple Bounding Boxes Algorithm 9
   2.1 General Description ............................... 9
   2.2 Simulation of One bounding box VS multiple bounding boxes .... 10
      2.2.1 One bounding box overlapping area expectation simulation ... 11
      2.2.2 Two bounding boxes overlapping area expectation simulation ... 12
      2.2.3 Three bounding boxes overlapping area expectation simulation ... 15
      2.2.4 Why not four and above ............................... 16
   2.3 Two Boxes Algorithm .................................. 21
   2.4 Three Boxes Algorithm ............................... 25

3 Parallel Implementation in CUDA 29
   3.1 GPU Information ..................................... 29
   3.2 Implementation ....................................... 32
      3.2.1 Bounding Interval ................................... 32
      3.2.2 Accumulated highest and lowest from both directions ....... 36
      3.2.3 Calculate heights .................................... 38
List of Figures

1.1 One bounding box is the smallest axis-aligned rectangle that covers all pixels of an object .......................................................... 2
1.2 GPU architecture ........................................................................ 4
1.3 Staircases of a set of rectangles .................................................. 7
1.4 Type 1 ....................................................................................... 7
1.5 Type 2 ....................................................................................... 7
1.6 Type 3 ....................................................................................... 7
2.1 One Bounding Box colliding into another One Bounding Box .......... 12
2.2 Different Coverage Value (B) ...................................................... 13
2.3 Two Bounding Boxes Overlapping another Two Bounding Boxes .... 14
2.4 Two Bounding Boxes always have a cutting line ............................ 17
2.5 Case 2: one box touches both top and bottom .............................. 19
2.6 Case 2: one box touches top and left ........................................... 19
2.7 Case 2: one box touches top and left ........................................... 19
2.8 One box can be separated from the other three by a line ............... 19
2.9 Two boxes can be separated from the other two by a line ............. 20
List of Algorithms

1. Find Lowest Pixel Index .................................................. 35
2. Hillis and Steele Scan Algorithm in parallel .......................... 36
3. Calculate Height ............................................................ 38
4. Column Highest using sequential implementation .................... 39
5. Column Highest using Parallel implementation ........................ 40
6. Accumulated Highest from left using sequential implementation .. 41
7. Accumulated Highest from left using parallel implementation ...... 41
8. Calculate Heights using sequential implementation .................. 42
9. Calculate Heights using parallel implementation ...................... 42
10. Calculate Areas using sequential implementation ..................... 43
11. Calculate Areas using parallel implementation (M threads) ........ 43
Chapter 1

Introduction

1.1 Problem Context

One of the most common operations in computer games involves detecting the collision between two sprites. Currently, most 2D games use an axis-aligned bounding box which tightly bounds a sprite being rendered [11] to detect possible collisions between a pair of sprites. When an overlap occurs between two boxes, higher fidelity checks are needed, and a per-pixel detection is performed in which each pixel in a sprite is tested against the pixels in another sprite for intersection. With sprites composed of \( n \) pixels, this test can cost up to \( O(n) \) operations, a very expensive cost, and unfavorable towards supporting real-time collision detection when many sprites are involved. We propose a parallel method (by using CUDA\(^1\)) to calculate multiple bounding boxes (2 or 3 boxes) for multiple sprites, thereby achieving better real-time collision detection. The advantage of using bounding boxes for

\(^1\)CUDA is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).
collision detection is that the intersection test is simple and low cost. In our two bounding boxes’ algorithm, it requires only four comparisons (between two pairs of two bounding boxes).

In this thesis, we use the following definitions:

1. One bounding box: One bounding box is the smallest axis-aligned rectangle that covers all pixels of an object. We will use the short term OBB in this paper.

![One Bounding Box (OBB)](image)

**Figure 1.1:** One bounding box is the smallest axis-aligned rectangle that covers all pixels of an object

2. $K$ partitions for an OBB: $K$ axis-aligned rectangles without overlapping inside the OBB.

3. $K$ covering for an OBB: $K$ axis-aligned rectangles that allow overlapping among them inside the OBB.

4. $K$ bounding boxes: $K$ axis-aligned rectangles that covers all pixels of an object without overlapping. The $K$ bounding boxes algorithm is actually looking for the optimal $K$ bounding boxes.
5. Optimal $K$ bounding boxes: The $K$ bounding boxes that have the smallest sum of area.

6. Cutting line: A horizontal or vertical line that separates one or more boxes from the rest of the boxes. A cutting line only crosses a rectangle when it crosses the interior of a rectangle, meaning that if a cutting line is on the boundary of a rectangle, it does not count as crossing that rectangle.

1.2 GPU introduction

1.2.1 Brief GPU history and Its Architecture

Graphic Processing Unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. A single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines, the GPU can achieve a tremendous computation improvement by parallelizing the calculation [9].

The GPU is a highly parallel, multithreaded, manycore processor with high memory bandwidth.[1]. It can provide memory bandwidth and floating-point performances that are orders of magnitude faster than a standard CPU. The reason behind it is that GPU is specialized for compute-intensive, highly parallel computation. In a GPU, the data elements are mapped to parallel processing threads, so that when we encounter large datasets, we do not need to waste time for CPU to do the calculation in serialized way. We chose to implement the multiple bounding boxes algorithm on the GPU for the faster paralleled calculation.
The GPU has a hardware architecture as illustrated in Figure 1.2 [9].

![Figure 1.2: GPU architecture](image)

In conclusion, the GPU is an attractive platform for general-purpose computation.

### 1.2.2 CUDA Introduction

nVidia introduced CUDA in November 2006. CUDA comes with a software environment that allows developers to use C (it also support various other languages and application programming interfaces) as a high-level programming language. The scalable programming model allows the GPU architecture to span a wide market range by simply scaling the number of multiprocessors and memory partitions.
The computation core of the CUDA programming model is the kernel, which is passed onto the GPU and executed by all the processor units, using different data streams. In a GPU, each kernel is launched from the host side (CPU), and it is mapped to a thread grid on the GPU. Each grid is composed of thread blocks [12]. All the threads from a particular block have access to the same shared memory and can synchronize together. On the other hand, threads from different blocks cannot synchronize and can exchange data only through the global (device) memory [8].

1.3 Prior Work

1.3.1 Sweep Line

Sweep Line constructions are commonly used in computational geometry. Generally a vertical line sweeps over a collection of objects and keeps track of intersections as it moves from left to right. It has the following steps [5]:

1) Maintain sweep line intersection
2) Maintain priority queue of (possible) event times ( = x coordinates of sweep line)
3) Until queue is empty:
   (A) Delete minimum event time from priority queue
   (B) Update sweep line intersection from \( < t \) to \( > t \)
   (C) Update possible event times in priority queue

One implementation using sweep line is segment intersection [5]. In this implementation, the input data is a list of line segments (two endpoints of a segment define the segment). The sweep line starts from the left (vertical sweep), and halts every time that it
meets with an event point (either the endpoint of a segment or an intersection point). Every
time the sweep line halts, it would change the sweep-line status (insert a new segment, 
remove a segment, determine the above/below relationship between two segments). The 
sweep line for segment intersection algorithm takes a total time $O((n + k) \log n$ where $k$ is 
the number of intersections).

In our application of sweep line, we would use the sweep line construction. The sweep 
line (either horizontal or vertical) would halt at every row/column, and check the possible 
distribution of our optimal two/three boxes using this sweep line as the current cutting line. 
Since the pixels are sorted, the sweep line stops at $O(\sqrt{n})$ locations.

### 1.3.2 Two Minimum Area Rectangle

In Becker et al.’s paper [3], they proposed an algorithm for approximating a set of 
rectangles by two minimum area rectangles. They classified the two rectangles into three 
types [3]: Type1 (Figure 1.4), Type2 (Figure 1.5) and Type3 (Figure 1.6). They proposed a 
sweep line algorithm for Type1, in which the line sweep scans each left and right side of all 
rectangles in ascending order of abscissa values. For Type2 and 3, they build a staircase\(^2\) 
(including NE, NW, SE, SW see Figure 1.3). They examine every active pairs of points in 
opposite staircases (for example NE vs SW), and examine the area of two rectangles (axis 
aligned) to find the optimal two boxes. The algorithm takes $O(n \log n)$ time in which $n$ is 
the number of rectangles. (They have a proof based on choosing to opposite corner points, 
which can be found as the lemma 3.6 of their paper [3]).

\(^2\)Staircase is a monotonically increasing or decreasing stair, it has to either be parallel to x-axis or y-axis.
Their problem differs from our research in the following aspects: first, they are using a set of rectangles as input whereas we are using a digital image with pixels as input. Since pixels are all integer value, we can achieve better running time. Also, their work considers two bounding boxes with overlapping. In the collision detection process, if we use the two boxes with overlapping, the overlapped area can double the cost of calculation. So in our algorithm, we use the partition of a sprite without having any overlapping between/among these boxes.

Figure 1.4: Type 1  
Figure 1.5: Type 2  
Figure 1.6: Type 3
1.4  Thesis Statement

1.4.1  Objectives

We come up with algorithms to find the optimal two and three bounding boxes respectively, show that using two or three bounding boxes outperforms using only one bounding box, argue why using four bounding boxes or more results in high computation complexity and implement the 2 bounding boxes algorithm in parallel using CUDA to achieve better performance for large data inputs.

1.4.2  Outline

In the following chapters, we start by introducing the algorithm for finding the best two bounding boxes and three bounding boxes. Then we will explain why using more bounding boxes is better for collision detection. Also, we explain why using four or more bounding boxes is infeasible due to the high computational costs. We then introduce all the parallel parts of our algorithms that are implemented in CUDA for the two bounding boxes algorithm. We conclude with further research and remaining issues.
Chapter 2

The Multiple Bounding Boxes Algorithm

2.1 General Description

As we shall show later in this chapter, the cost of finding four or more bounding boxes has a large calculation overhead since the sweep line framework would not be sufficient to find the optimal bounding boxes for four and above. Our algorithm uses a sweeping line to find the best cutting line, thus finding the optimal two or three boxes for a sprite. The sweeping line method is an efficient technique that takes $O(n)$ time for an image of $n$ pixels, so using this method will discover the optimal boxes quickly. We scan from left to right as well as top to bottom, and in total spend linear time ($O(n)$) where $n$ is the total pixel count in the sprite for finding an optimal two or three bounding boxes.
2.2 Simulation of One bounding box VS multiple bounding boxes

To demonstrate the motivation for using more than one bounding box per sprite for collision detection, we compared the expected number of per-pixel tests when using one bounding box per sprite versus using two or three bounding boxes per sprite.

The primary reason for using multiple bounding boxes instead of just a single bounding box is that it reduces the asymptotic cost, which we shall demonstrate through mathematical analysis and by measuring the cost reduction through the simulation tests.

First, assume we have two sprites with \( n \) pixels each. Mathematically, we can calculate that a \( k \times n / k \) box overlaps another \( k \times n / k \) box randomly with an area overlapping probability expectation of 0.25 of the total area. This is calculated by the following formula: (WLOG, let the first box be located at (0,0) and the second box randomly placed...
at pixel position \((x, y)^1\). Since \(x\) and \(y\) are independent, we could multiply the expectations in the following equation.

\[
\text{Expectation of Overlapping Area Percentage} = \frac{(\mathbb{E}(k-x)) \times (\mathbb{E}(n/k-y))}{n}
= \mathbb{E}(1 - x/k - y/(n/k) + x \times y/n)
= 1 - 0.5 - 0.5 + 0.25
= 0.25 \quad (2.1)
\]

Next, we conduct the simulation test on one bounding box and two/three boxes to explain why in practice, randomly generated two or three bounding boxes have lower expectation in collision pixels’ check than the one single bounding box.

### 2.2.1 One bounding box overlapping area expectation simulation

In this simulation, we have two \(1000 \times 1000\) boxes, and since we do pixel by pixel calculations, the overlapping area should be an integer rather than a float. We use brute-force to put one box’s left corner at every possible pixel of the other box, then calculate the area in which the two are overlapping.

We ran this simulation for 1,000,000 trials for a generalized area expectation, and averaged the overlapping area to achieve the result 25%, which is statistically close to the theoretical area expectation. This demonstrates that on average, 25% of the pixels of the

\(^1x\) and \(y\) should uniformly range from 0 to \(K\) and 0 to \(n/K\) respectively.
two OBBs will be overlapping. It is therefore possible that on average $25\%$ of the pixels need to be checked to be sure that a collision did not occur.

![Figure 2.1: One Bounding Box colliding into another One Bounding Box](image)

### 2.2.2 Two bounding boxes overlapping area expectation simulation

In this simulation, we use a coverage percentage $B$ as a parameter. The coverage percentage means what percent of the single bounding box is covered by the two bounding boxes. For example, Figure 2.2a represents that the optimal two bounding boxes cover 10 percent of the original one bounding box, Figure 2.2b illustrates that the optimal two bounding boxes cover 60 percent of the original one bounding box and Figure 2.2c represents that the optimal two bounding boxes cover 90 percent of the original one bounding box. Using different set of coverage, we generate randomized two boxes that cover a certain percentage of the one bounding box.
In Figure 2.3 we can see two “two bounding boxes” overlapping. In the simulation step, we need to calculate the expectation of overlapping area by adding the four potential overlapping areas. For example, in Figure 2.3, we need to calculate

\[
\text{overlappingArea} = \text{OverlappingAreaOf}(\text{Rect}(IJKL), \text{Rect}(OPQR)) + \text{OverlappingAreaOf}(\text{Rect}(IJKL), \text{Rect}(STGH)) + \text{OverlappingAreaOf}(\text{Rect}(MNCD), \text{Rect}(OPQR)) + \text{OverlappingAreaOf}(\text{Rect}(MNCD), \text{Rect}(STGH))
\]  

(2.3)
Table 2.1 shows the results from our simulations (the first row indicates the first object’s percentage coverage and the first column indicates the second object’s percentage coverage):

**Table 2.1: Using Two Bounding Boxes: Overlapping Area Expectation (Percentage)**

<table>
<thead>
<tr>
<th>First Object’s B</th>
<th>0.1</th>
<th>0.3</th>
<th>0.5</th>
<th>0.7</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.13%</td>
<td>0.17%</td>
<td>0.18%</td>
<td>0.19%</td>
<td>0.19%</td>
</tr>
<tr>
<td>0.3</td>
<td>0.58%</td>
<td>1.3%</td>
<td>1.5%</td>
<td>1.6%</td>
<td>1.6%</td>
</tr>
<tr>
<td>0.5</td>
<td>1.1%</td>
<td>2.8%</td>
<td>4.1%</td>
<td>4.5%</td>
<td>4.7%</td>
</tr>
<tr>
<td>0.7</td>
<td>1.8%</td>
<td>4.9%</td>
<td>7.4%</td>
<td>9.1%</td>
<td>9.6%</td>
</tr>
<tr>
<td>0.9</td>
<td>2.5%</td>
<td>7.4%</td>
<td>11%</td>
<td>14%</td>
<td>17%</td>
</tr>
</tbody>
</table>

*B means the Percentage Coverage in the One Bounding Box
From Table 2.1 we can see that if we increase the coverage percentage, the result would get worse, which means that the expected overlapping area is increasing. Depending on the shape of the object, we may be able to achieve a significant amount of improvement over one bounding box. Note from the results that we simulated, even the 90 percent coverage case has an expectation of 17%, which is better than the 25% expectation of overlapping area we have in the one bounding box simulation.

2.2.3 Three bounding boxes overlapping area expectation simulation

In the three bounding boxes’ simulation, we generate random three boxes within the one bounding box (100 × 100) without overlapping. Random three boxes process is as following: first, we take a random value from 1 to 99 to generate a horizontal/vertical cutting line; then, we generate one box on one side of the cutting line (either be top side, bottom side, left side or right side) and two boxes on the other side without overlapping; the two boxes on the other side can be generated in two ways: side by side (generate another cutting line two which is parallel to the previous cutting line, or generate another cutting line two which is perpendicular to the previous cutting line. After we have random three boxes, we handle the collision process as follows: First, we generate a random number (integer) within the range of 1 to 99 to decide a cutting line position. Second, we generate one box on one side of the cutting line, two boxes on the other side of the cutting line without overlapping. Then we compare these three boxes against the other three boxes (9 pairs of calculation in total), and calculate the total overlapping area. The result from this simulation is 3.41% against the one bounding box’s 25%, which means more than 80% time saving can be achieved using three bounding boxes on average.
2.2.4 Why not four and above

In our algorithm analysis, we tried to test how much time it takes if we want to find the optimal four bounding boxes partition. It turns out using four boxes can be tricky. It’s easy to argue that if you have a two boxes partition, you can always find a line (parallel to x-axis or y-axis) to separate these two boxes (Figure 2.4a). Also, for the three boxes partition, there always exists a line that separates one box from the other two (See proof below). However, when there are four boxes, there exists instances where no box can be separated from the others using any cutting line within the big single bounding box (Figure 2.10 and Figure 2.11). This makes the cost of finding four optimal boxes too expensive.

Definition: We say two non-overlapping bounding boxes are optimal if they have the minimal sum of areas that covers all foreground pixels of the sprite.

Theorem 2.2.1. If \( A \) and \( B \) are the optimal two bounding boxes in the OBB which cover all the pixels in the OBB, then all four edges of the OBB must be at least touched by one edge of \( A \) or \( B \).

Proof. We will prove this theorem by contradiction. Let us assume that there is one edge of the OBB (WLOG we can assume it is the top edge) that is not touched by \( A \) or \( B \), then since \( A \) and \( B \) cover all the pixels in OBB, we can decrease the top of the OBB. However, this violates the fact that the OBB is the smallest box that covers all the pixels.

Therefore this contradiction shows us that all four edges of the OBB must be touched by at least one edge of these two boxes (\( A \) or \( B \)).

Theorem 2.2.2. If \( A \) and \( B \) are two non overlapping boxes in the OBB, then there must exist a cutting line (parallel to x-axis or y-axis) that separate \( A \) and \( B \).
Proof. From the definition of $A$ and $B$, we know that $A$ and $B$ can not overlap. There must exist either a line that $A$ and $B$ share, e.g. the mutual edge, or a gap between $A$ and $B$. If there is a line that $A$ and $B$ share as the edge that overlap, then this line itself is a cutting line for $A$ and $B$ (2.4a). Otherwise, the gap between $A$ and $B$ provides several cutting lines. Figure 2.4b shows that all the lines parallel to the y axis (or x axis, depending on the gap) and within the gap will have one box on the line’s left side and the other box on the other side (or top and bottom respectively). Thus the lines within the gap parallel to y axis can all be a cutting line for these two boxes.

Figure 2.4: Two Bounding Boxes always have a cutting line

We shall prove that there always exists a cutting line for three boxes.

Theorem 2.2.3. If $A$, $B$ and $C$ are three non overlapping boxes in the OBB, then there must exist a cutting line (parallel to x-axis or y-axis) that separates one from the other two.

Proof. Assume we have three boxes, such that none of these three boxes would be touching four sides of the OBB, (otherwise such a box leaves no room for the other two boxes). Then,
from the pigeonhole principle, we can be sure that there must exist a box that touches two 
or more sides of the OBB. Since no box can touch four sides of the OBB, only two cases 
remain here:

- **Case 1**: One box touches three sides of the OBB, then the remaining side of this box 
  would be an eligible cutting line that separates itself from the other two boxes.

- **Case 2**: Box 1 touches two sides of the OBB. If Box 1 touches both the top edge 
  and the bottom edge (or left and right edge) of the OBB as in Figure 2.5, it is easy to 
  see that line $j$ or line $k$ (only one of $j$ and $k$ exists if the top box touches three edges: 
  top, bottom and left/right edge) can separate one box from the other two. If Box 1 
  touches the top edge and the left edge (or any two edge that are not parallel) of the 
  OBB (Figure 2.6), then the other two boxes must touch the bottom edge of the OBB. 
  Let’s assume the second box touches the bottom edge and also has line $h$ crossing 
  itself (as in Figure 2.7), then the second box would create two lines $l$ and $m$ that need 
  to be crossed by the third box. There is no way to put the third box which has line $i$, 
  $m$ and $l$ crossing itself. By contradiction, we conclude that there must exist a cutting 
  line to separate a box from the other two boxes for case 2.
Theorem 2.2.4. There are instances of four bounding boxes such that no cutting line can separate one or more boxes from the rest.

Proof. There are generally three cases for the layout of the four bounding boxes. These three cases are listed as follows:

Case 1: There exists a line that can separate one box from the other three (Figure 2.8).

Figure 2.5: Case 2: one box touches both top and bottom

Figure 2.6: Case 2: one box touches top and left

Figure 2.7: Case 2: one box touches top and left

Figure 2.8: One box can be separated from the other three by a line
Case 2: There exists a line that can separate two boxes from the other two (Figure 2.9).

Figure 2.9: Two boxes can be separated from the other two by a line

Case 3: There exists no line that can separate one or more boxes from the rest (Figure 2.10 and Figure 2.11).

Figure 2.10: no box can be separated from the rest by a line

Figure 2.11: no box can be separated from the rest by a line
In case 3, it’s impossible to separate one box or two boxes from the rest. Thus, the sweep line wouldn’t help in getting the optimal four bounding boxes. If we have more than 4 boxes, this condition still exists, therefore we could not use the same sweep line strategy to find the optimal K (K≥3) boxes.

2.3 Two Boxes Algorithm

We begin with the original image of a sprite (Figure 2.12). Assume 1 represents a foreground pixel (which means the sprite itself) and 0 represents a background pixel (the pixels that do not belong to the sprite).

![Image of a sprite with 1s and 0s]

*Figure 2.12: Input Sprite image using 1s & 0s*
When we have the original image, we want to decide an optimal cutting line which can either be horizontal or vertical such that we can have a minimal sum of two boxes on both sides of the line.

**Step1. Find bounding interval in each column and row**

In Step1, we scan every column and row to find the highest and lowest foreground pixel of each column (row). From Figure 2.12, we should have four arrays as a result: the highest pixels of each column, the lowest pixels of each column, the leftmost pixels of each row, and the rightmost pixels of each row.

The results would be: highest = \{1,2,2,0,0,0,1,0\}, lowest = \{5,4,4,5,6,6,5,7\}, leftmost = \{3,0,0,2,0,0,4,7\}, rightmost = \{7,7,7,5,7,6,5,7\}.

If we use the sequential implementation, we should scan from the top of each column downward to find the first foreground pixel (indicated by ‘1’), and decide the highest pixel for this column. Assume our sprite has width \(K (1 \leq K \leq n)\) and height \(n/K\), then the cost for scanning the bounding interval is:

\[
K \times n/K + n/K \times K = O(n)
\]

We can conclude that the scan takes linear time \(O(n)\) for an image of \(n\) pixels. However, in Chapter 3, we will give out a parallel implementation algorithm which only takes \(O(\log n)\).

**Step2. Find the accumulated highest and lowest pixel from left and from right (top and down)**

As you can see from Figure 2.13, if we choose line a as a cutting line, the highest pixel on the left should be 0, because we need the box to cover the top pixel in column 3. Even
though column 0 - 2 do not have any pixel as high as position 0, we have to define the accumulated highest pixel at column 3 to be 0.

![Figure 2.13: We need accumulated highest not the highest pixel in one column](image)

For Step2, we only need to scan the four arrays we got from Step1, and check from left and right (top and down) to get the accumulated value. For the given image in Figure 2.12, we have: 

- **accumulatedHighestFromLeft** = \{1,1,1,0,0,0,0,0\},
- **accumulatedHighestFromRight** = \{0,0,0,0,0,0,0,0\},
- **accumulatedLowestFromLeft** = \{5,5,5,5,6,6,6,7\},
- **accumulatedLowestFromRight** = \{7,7,7,7,7,7,7,7\}. This step takes time:

\[
O(K) + O(K) + O(n/K) + O(n/K) = O(K + n/K)
\]

If \(K\) equals \(\sqrt{n}\), then Step2 takes time \(O(\sqrt{n})\).

**Step3. Calculate heights from left and right (top and down)**
Step 3 is very straightforward. Having a cutting line, we have a box on the left of the cutting line and a box on the right (assume we have a vertical cutting line). The left box has width from 0 to the cutting line position, height of $\text{accumulatedLowestFromLeft} - \text{accumulatedHighestFromLeft}$. The right box has width from the cutting line to $\sqrt{n}$ (which in our case is 8), height of $\text{accumulatedLowestFromRight} - \text{accumulatedHighestFromRight}$.

This step takes also $O(K + n/K)$ time.

![Figure 2.14: Find the box on left and right of the cutting line](image)

**Step 4. Calculate areas**

Once we have the heights in Step 3, we can easily calculate the sum of two boxes’ areas. In Figure 2.14, the cutting line is in position 3, and the left box has width 4, height $(5-0+1) = 6$, so left box has area $4 \times 6 = 24$. The right box has a width $(7-4+1) = 4$, height $(7-0+1) = 8$, and area $= 32$. Thus, the sum of these two boxes is 56.

We will check every cutting line (horizontally and vertically), to find the minimal sum of two boxes’ area. Step 4 takes time $O(K + n/K)$. 

24
From Step1 to Step4, it takes $O(n) + O(K + n/K) + O(K + n/K) + O(K + n/K) = O(n)$ time, therefore this is a linear time algorithm in sequential implementation. For every sprite, we can use this algorithm to find the best two boxes for it, and pay an overhead of $O(n)$.

### 2.4 Three Boxes Algorithm

As in the last section “two boxes algorithm”, we have an input image indicated with 1s and 0s. Now we want to find the optimal three boxes that have the minimal sum of areas. We proved earlier in this chapter that in the three boxes situation, there always exists a line (horizontal or vertical) that separates one box from the other two. Our algorithm begins with the scanning as well, thus we need to scan from left to right, top to bottom, and also in these two scans, we need to check two possibilities: one box is on left (top) or one box is on right (bottom). Let us discuss the situation in which we scan from left to right, and one box is on the right of the separating line.

**Step1. Find bounding interval in each column and row**

As in the algorithm for two boxes, our first step is to get the highest and lowest pixel for every column.

**Step2. Get accumulated highest and lowest from left and right (top and down)**

This step is the same as in last section.

**Step 3. Side by side type scan**

If the three boxes can be separated by 2 vertical (or horizontal) lines as in Figure 2.15a, the scan procedure is almost the same as in two boxes condition. Otherwise, it must be the
case that there is one vertical line and one horizontal line that separate three boxes as in Figure 2.15b. We shall show the solution for the latter in Step 4.

In Step 3, the two side boxes are easily defined by the accumulated highest and lowest from left and right. However, for the middle box we need to spend up to $O(M)$ time to find its height, where $M$ is the width of the current middle box and $M$ would be from 0 to $K$ for two vertical lines separation. The side by side type scan takes time:

$$O(K) \times O(K) \times O(K) + O(n/K) \times O(n/K) \times O(n/K) = O(K^3 + (n/K)^3)$$

If we are dealing with a sprite whose aspect ratio is close to 1, it takes up to $O(n \sqrt{n})$. However, in the worst case it can take up to $n^3$ when $K$ is $O(1)$ or $O(n)$.

**Figure 2.15: Two cases of three boxes**

- **(a)** Three boxes can be separated by 2 vertical lines
- **(b)** Three boxes can be separated using a vertical and horizontal line

**Step 4. Scan in major direction then in minor direction**

In the beginning of the scan, we can get a partition like Figure 2.16a. The horizontal scan is the major scan, it stops after the first column (column 0). The vertical scan stays on
one side of the major cutting line, in our example, the minor scan is on the left side of the major cutting line. To help find the bound intervals of the two boxes on the left side, we use the data structure priority search tree. The priority search tree can be built in $O(n \log n)$ time and can find the bounding interval in $O(\log n)$ time. Once we have the bounding intervals for all three boxes, we can calculate the sum of areas of these three boxes. In Figure 2.16a, we have the left two boxes’ area sum = $1 \times 2 + 1 \times 2 = 4$. The right box has area $7 \times 8 = 56$ (the right box is just one box, so it follows the same steps as in two boxes algorithm which we won’t spend time to describe again). The total is $4 + 56 = 60$.

The information in the last major scan can be stored for the next. For example, in the Figure 2.16b, we can use the array info in last scan where $topLeft[2] = 0, bottomLeft[2] = 0$ directly because the left side remains the same. For the right side, we just need to check the new column we just passed, if it has any pixel above the horizontal line (by checking the highest pixel in step 1), we increase the $topRight[2]$ by 1, otherwise remain the same value. For the bottom box, it is the same scheme. In Figure 2.16b the left two boxes has sum of area: $2 \times 2 + 2 \times 2 = 8$, and three boxes’ area = $8 + 6 \times 8 = 56$.

So in every minor scan it takes $O(\log n)$ time, there are $n/K$ (or $K$ for horizontal major scan) minor scans in every major scan, and there are $K$ (or $n/K$) major scans in total. Therefore the three boxes algorithm will have a total cost of $O(\log n) + O(\log n) \times O(K) \times O(n/K) + O(\log n) \times O(n/K) \times O(K) = O(n \log n)$.
(a) Initial scan of 3 boxes

(b) Next scan of 3 boxes
Chapter 3

Parallel Implementation in CUDA

There are several functions that are implemented in parallel using CUDA, such as finding the highest and lowest pixel of every column, getting the accumulated highest and lowest position and getting the accumulated heights. Using the parallel implementation can reduce the running time and the CPU burden, thus achieving a better performance.

3.1 GPU Information

The following table describes the profile of the GPU we are using:
Table 3.1: GPU Information Table 1

<table>
<thead>
<tr>
<th>GPU Device Name:</th>
<th>Tesla C2070</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute capability:</td>
<td>2.0</td>
</tr>
<tr>
<td>Clock rate:</td>
<td>1147000</td>
</tr>
<tr>
<td>Device copy overlap:</td>
<td>Enabled</td>
</tr>
<tr>
<td>Kernel execution timeout :</td>
<td>Disabled</td>
</tr>
<tr>
<td>Total global mem:</td>
<td>6442123264</td>
</tr>
<tr>
<td>Total constant Mem:</td>
<td>65536</td>
</tr>
<tr>
<td>Max mem pitch:</td>
<td>2147483647</td>
</tr>
<tr>
<td>Texture Alignment:</td>
<td>512</td>
</tr>
<tr>
<td>Multiprocessor count:</td>
<td>14</td>
</tr>
<tr>
<td>Shared mem per mp:</td>
<td>49152</td>
</tr>
<tr>
<td>Registers per mp:</td>
<td>32768</td>
</tr>
<tr>
<td>Threads in warp:</td>
<td>32</td>
</tr>
<tr>
<td>Max threads per block:</td>
<td>1024</td>
</tr>
<tr>
<td>Max thread dimensions:</td>
<td>(1024, 1024, 64)</td>
</tr>
<tr>
<td>Max grid dimensions:</td>
<td>(65535, 65535, 65535)</td>
</tr>
</tbody>
</table>
We implemented the two bounding boxes algorithm in CUDA, considering the capability of parallelization of GPU, the goal is to achieve better performance via parallelizing processes. We shall discuss all the functions that are implemented in parallel in the following sections. We use an 8×8 sprite sample in Figure 3.1 to help describe each step. We use a deep green color to represent the foreground pixels. In order to check the column for every possible cut line position, we shall need an input as a 1D as input array (we list every 8 pixels per row just for the convenience of reading):

0, 0, 0, 1, 1, 1, 0, 1,
1, 0, 0, 1, 0, 0, 1, 1,
1, 1, 1, 0, 1, 0, 1, 1,
0, 0, 1, 1, 0, 1, 0, 0,
1, 1, 1, 1, 1, 0, 1, 0,
1, 0, 0, 1, 1, 0, 1, 0,
0, 0, 0, 0, 1, 1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 1.
3.2 Implementation

3.2.1 Bounding Interval

Using a sequential method, finding the highest and lowest pixel in one column can take up to $O(M)$ time for an column of M pixels, because we need to scan from the top in one column (for the highest pixel) downward until we reach a point that is occupied. For the lowest pixel we do the exact opposite step by scanning from bottom to top. In the parallel implementation of this step, we refer to this operation as the parallel summation reduction (see Figure 3.2) [10].
In Figure 3.2, we half the size of the array every time, which means there is in total at most \( \log n \) steps. In every step, we compare element \( i \) with element \( (i + j) \), where \( j \) is half of the current size of the array. In Figure 3.2, the \( f(x, y) \) represents the process in which we take the lower pixel that is 1, store the index. If neither \( x \) or \( y \) is 1, we store infinity. This takes \( O(\log n) \) per column.

For example, if we have an image size 1024 \( \times \) 1024, in the first run, the array size would be 1024, so we compare element \( i \) with element \( i+512 \). If we are looking for the highest pixel, we will choose the element which has a bigger value, and store it in position \( i \). Now we have a half sized array with 512 elements. We will do the comparison again,
and leave the bigger value in the first half of array, then half size the array until there is only one element left. And this element array[0] will store the highest pixel position.

The function below findLowest is executed in parallel. We takes parameter image which is an array of 0s and 1s (1 indicating a foreground pixel and 0 indicating a background pixel). We put the array into M blocks where each block has N threads. We preprocess cache[threadsPerBlock] first, giving it value N+1 for background (so that it will be larger than any other threads) and value of the position in column for foreground pixel. After preprocessing, we run a loop for \( O(\log N) \) time. In each loop, we check if the cache[cacheIndex] is greater than cache[cacheIndex+i]. If it is bigger, we need to replace it with cache[cacheIndex+i] because we need to find the lowest position possible. We need to synchronize threads in each step to make sure every thread has finished the processing. At last, if there is only one element left in cache, which means we processed all the pixels and stored the lowest position in cache[0], we will put that value in c[blockIdx.x]. So in this way we calculate every column independently in one block, and put the results back in an array with all the lowest pixel and highest pixel values for each column in a global array result.

For our sample sprite, our result would be find the lowest array \{5, 4, 4, 5, 6, 6, 5, 7\}, and highest array\{1, 2, 2, 0, 0, 0, 1, 0\}.
**input**: An array input of size $M \times N$

**output**: An array lowest of size $M$

shared int cache[threadsPerBlock];

int $tid = threadIdx.x + blockIdx.x \times blockDim.x$;

int $cacheIndex = threadIdx.x$;

if $image[tid] == 0$ then

| $cache[cacheIndex] = N + 1$

else $cache[cacheIndex] = threadIdx.x$;

syncthreads();

int $i = blockDim.x/2$;

while $i! = 0$ do

| if $cacheIndex < i$ then

| | if $cache[cacheIndex] > cache[cacheIndex + i]$ then

| | | $cache[cacheIndex] = cache[cacheIndex + i]$

| end

end

syncthreads();

$i/ = 2$;

end

if $cacheIndex == 0$ then

| $result[blockIdx.x] = cache[0]$

end

**Algorithm 1**: Find Lowest Pixel Index
3.2.2 Accumulated highest and lowest from both directions

However having the highest and lowest per column is far from enough, because every cut line needs the highest point on its left as the highest pixel for left box, and highest point on its right on its right as the highest pixel for right box. So in this step we will generate the accumulated highest/lowest pixel position from both left to right and right to left.

Harris described several algorithms for paralleled prefix sum calculation [7], with the Hillis and Steel algorithm as follows:

```plaintext
input : An array highest input of size M
output: An array accumulatedHighest of size M

for d ← 1 to log_2 M do
    for all thread k in parallel do
        if k ≥ 2^d then
            x[k] ← the smaller of (x[k-2^d-1], x[k])
        end
    end
end
```

**Algorithm 2**: Hillis and Steele Scan Algorithm in parallel
Algorithm 2 performs a total of $\sum_{d=1}^{\log_2 n} n2^{d-1} = O(n \log_2 n)$ addition operations. Which in parallel is achieving $n/\log_2 n$ times better performance. The book [7] also offers an even more efficient parallel scan algorithm, however we haven’t implemented this one yet, and would be one of the future tasks.
3.2.3 Calculate heights

Now we have the accumulated highest and lowest arrays from left and right, we need to combine the highest and lowest to get the height array. This is a very paralleled process since we only need to subtract each highest position to the lowest position.

| **input** | An array highest input of size $M$, an array lowest input of size $M$ |
| **output** | An array height of size $M$ |
| int $i = blockIdx.x$; |
| **if** $i < M$ **then** |
| $\text{height}[i] = \text{high}[i] - \text{low}[i] + 1$ |
| **end** |

Algorithm 3: Calculate Height

3.2.4 Calculate Areas

The calculate area function is just like the calculate heights. We use a sweep line to sweep from left to right, top to bottom, and try to find the best cut line to make the optimal two bounding boxes (the sum of the two boxes’ area is the minimal).

3.3 Performance Improvements

3.3.1 Theoretical Performance Improvements

3.3.1.1 Bounding Interval

The sequential implementation needs to go from the top to find the highest point and stop when it reaches a point that is occupied (a foreground pixel). Thus, the sequential
implementation would take $O(n)$ running time in average (for a sprite size of width $K$ and height $n/K$). In the parallel implementation, we compare one half of the column (or row) with the other half, and keep the first half as the result. Also, for multiple columns/rows, we can take advantage of parallelism and compute at the same time. Thus, the steps required are $O(\log K + \log n/K)$ in parallel. Compared to $O(n)$, it is a big improvement.

A

<table>
<thead>
<tr>
<th>Algorithm 4: Column Highest using sequential implementation</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>input</th>
<th>An array input of size $M \times N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>output: An array $\text{highest}$ of size $M$</td>
<td></td>
</tr>
</tbody>
</table>

for $i \leftarrow 0$ to $M$ do
  for $j \leftarrow N-1$ to 0 do
    if $\text{input}[i \times M + j] == 1$ then $\text{highest}[i] \leftarrow j$ and break ;
  end
end

Algorithm 4: Column Highest using sequential implementation
**input**: An array input of size $M \times N$

**output**: An array highest of size $M$

put input[i][j] in block $i$ thread $j$;

```
if input[i][j] == 1 then
    // cache store the column number
    cache[i] ← threadIdx.x;
else
    cache[i] ← -1;
```

$i ← N/2$;

```
while $i \geq 1$ do
    // in every block, do in parallel
    for all thread $k < i$ in parallel do
        if cache[k] < cache[i+k] then
            cache[k] ← cache[i + k];
        end
    end
    $i ← i/2$;
end
```

**Algorithm 5**: Column Highest using Parallel implementation

### 3.3.1.2 Accumulated highest and lowest from both direction

If we run the accumulated highest and lowest step in a sequential manner, it would take $O(K + n/K)$ time, because we need to go through all the columns/rows to find the accumulated highest/lowest pixel position (6).
However, if we implement the accumulated function in parallel, we can achieve $O(\log K + \log n/K)$ time by adding calculating multiple threads at the same time (for an image of size $K \times n/K$ we put $\log K$ (or $\log n/K$) threads in parallel).

```
input : An array highest of size $M$
output: An array highFromLeft of size $M$

for $i \leftarrow 0$ to $M$
do

  highFromLeft$[i] \leftarrow$ highest$[i]$;

  if highFromLeft$[i] <$ highFromLeft$[i-1]$ && $i > 0$
  then highFromLeft$[i] \leftarrow$ highFromLeft$[i-1]$;

end
```

**Algorithm 6:** Accumulated Highest from left using sequential implementation

```
input : An array highest of size $M$
output: An array highFromLeft of size $M$

for $i \leftarrow 1$ to log$_2$ $M$
do

  for $k \in$ parallel threads do

    if $k \geq 2^i$
    then highFromLeft$[k] \leftarrow$ (the bigger of (highFromLeft$[k]$,
                                                  highFromLeft$[k - 2^{i-1}]$);

  end

end
```

**Algorithm 7:** Accumulated Highest from left using parallel implementation
3.3.1.3 Calculate heights

Having the accumulated highest and lowest, it is easy to calculate the accumulated height by simply subtracting every element in lowest array from highest array (8). The running time of this calculation can easily be seen to be \( O(\log K + \log n/K) \).

Running this algorithm in parallel would simply take these two arrays and do the subtraction in parallel which takes \( O(1) \) time. For a large set of data, this is a huge improvement in running time.

```
for i ← 0 to M do
    heightFromLeft[i] ← highFromLeft[i] − lowFromLeft[i];
end
```

**Algorithm 8:** Calculate Heights using sequential implementation

```
for i ← 0 to M do
    // do all in parallel
    heightFromLeft[i] ← highFromLeft[i] − lowFromLeft[i];
end
```

**Algorithm 9:** Calculate Heights using parallel implementation
### 3.3.1.4 Calculate Areas

The calculation of areas step is just like calculating heights, where the sequential implementation takes $O(\log K + \log n/K)$ while in parallel we can complete in constant time.

**Algorithm 10: Calculate Areas using sequential implementation**

**input**: An array $heightFromLeft$ of size $M$, array $heightFromRight$ of size $M$

**output**: An array areas of size $M$

for $i \leftarrow 0$ to $M$

| areas[$i$] $\leftarrow heightFromLeft[i] \times i + lowFromLeft[M - 1 - i] \times (M - i)$; |

end

**Algorithm 11: Calculate Areas using parallel implementation (M threads)**

**input**: An array $heightFromLeft$ of size $M$, array $heightFromRight$ of size $M$

**output**: An array areas of size $M$

for $i \leftarrow 0$ to $M$

| areas[$i$] $\leftarrow heightFromLeft[i] \times i + lowFromLeft[M - 1 - i] \times (M - i)$; |

end
Chapter 4

Performance Analysis

In Chapter 2, we analyzed the 2 boxes and 3 boxes algorithms, it is shown that we can pay $O(n)$ overhead to find the optimal 2 boxes or $O(n \log n)$ 3 boxes. In the process of checking collisions between two sprites, we can eliminate many per-pixel checks using 2 boxes or 3 boxes (depending on the image, the improvement varies, however we can always reduce the calculation of using just one bounding box). Considering that collision detection can cost up to $O(n)$, and it occurs many times in the game process, it is better to pay the $O(n)$ (or $O(n \log n)$) overhead cost and use the optimal 2 boxes or 3 boxes.

Also, in Chapter 3, we presented a parallel implementation for the 2 boxes which can achieve $O(\log n)$ running time, quite an improvement over $O(n)$ overhead cost.

We ran both the CUDA program and the sequential program for 10000 times to get an average running time. From Table 4.2, we can see that even for an image size of $1024 \times 1024$, it takes around 40 seconds in CUDA to find the optimal two boxes, and we can then store the partition in memory and incur less calculation in the then coming collision
checking. However for an image size of $64 \times 64$, CUDA actually performs worse than the CPU sequential program. It’s easy to predict if we have a bigger size of image (it can be a series of small images), CUDA would further improve its running time in comparison to the sequential implementation.

In conclusion, it is proven by algorithm analysis and statistical simulation that using 2 boxes or 3 boxes can have the benefit of reducing realtime calculation and thus achieve better performance.

**Table 4.1: CUDA Program VS Sequential Program Performance Using $64 \times 64$ Image**

<table>
<thead>
<tr>
<th>Performance</th>
<th>CUDA Program</th>
<th>Sequential Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Clock Time:</td>
<td>21.670000s</td>
<td>0.290000s</td>
</tr>
<tr>
<td>Real Running Time:</td>
<td>0m22.190s</td>
<td>0.298s</td>
</tr>
<tr>
<td>User Time:</td>
<td>0m6.076s</td>
<td>0.296s</td>
</tr>
<tr>
<td>System Time:</td>
<td>0m16.649s</td>
<td>0m0.00s</td>
</tr>
</tbody>
</table>

**Table 4.2: CUDA Program VS Sequential Program Performance Using $1024 \times 1024$ Image**

<table>
<thead>
<tr>
<th>Performance</th>
<th>CUDA Program</th>
<th>Sequential Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Clock Time:</td>
<td>39.520000s</td>
<td>61.610001s</td>
</tr>
<tr>
<td>Real Running Time:</td>
<td>0m41.829s</td>
<td>1m1.725s</td>
</tr>
<tr>
<td>User Time:</td>
<td>0m27.302s</td>
<td>1m1.612s</td>
</tr>
<tr>
<td>System Time:</td>
<td>0m12.297s</td>
<td>0m0.00s</td>
</tr>
</tbody>
</table>
Chapter 5

Future Work and Conclusion

As you might have noticed, our algorithm is using axis-aligned bounding boxes instead of arbitrary oriented bounding boxes. The next step of the research could be having multiple arbitrary oriented bounding boxes. Garcia-Alonso et al. has proposed an method of checking two arbitrary boxes’ overlapping areas by using basic the x and y axis and the transformation matrix [6].

The second direction could be to apply the algorithm for the 3D applications. Aggarwal et al. proposed an algorithm of running time $O(n^3)$ for a point set size of $n$ [2]. Furthermore in Clingman et al.’s book [4], they mentioned the bounding sphere, bounding box, bounding cylinder and even bounding polytope to detect collisions. As 3D games and devices are more and more popular, it is a potential research direction for us to determine an algorithm for optimal multiple 3D boxes. Also, there might be a possibility of using other polygon shapes other than rectangles for collision detection, such as parallelogram and trapezoid.
Last, for the CUDA parallelism, there are plenty of algorithms that can be transformed from sequential implementation to paralleled implementation. For a large set of data, it is worthwhile to find a way to transform some of these algorithms into CUDA implementation. For example, in this paper we did not create a parallel implementation for 3 boxes, and we can very likely gain better performance using a CUDA implementation for it.

In this thesis, we first proved that using 2 boxes or 3 boxes can achieve less calculation than using one bounding box which is normally used (Chapter 2). Then, we introduced the algorithm to find the optimal 2 boxes or 3 boxes, and analyzed the performance of these algorithms. It turns out that finding the optimal 2 boxes and 3 boxes takes $O(n)$ and $O(n \log n)$ respectively for an image size of $n$ pixels.

In Chapter 3, we implemented the algorithm for 2 boxes in parallel using CUDA. The parallel algorithm covers all of the four steps of our sequential algorithm, and gets the running time down to $O(\log n)$. Finally, in the simulation test, the parallel implementation achieved the better running time than the sequential implementation (4.2).
Bibliography


Appendix A

Appendix A: Source Code

/*****
Author: Min Qi
Date: Nov 28, 2012
Description: This is a program that does a random simulation to
find the collision pixel expectation for one bounding box

******/

#define NUM_OF_TIME 1000000
#include<stdio.h>
#include<stdlib.h>
int gen_rand(int, int);
void main(void)
{
    double x,y;
    double total=0.0;
for(int i=0;i<NUM_OF_TIME;i++){
    x=gen_rand(0,100);  // we put the second OOB randomly on the 
    // first OOB (second OOB’s top left corner is inside the first 
    // OOB)
    y=gen_rand(0,100);
    total+=x*y;
}

printf("Expected Collision Area is %f \n",total/NUM_OF_TIME);

Listing A.1: Random One Box Calculation Expectation

/******
Author: Min Qi
Date: Nov 30, 2012
Description: This is a program that uses random simulation to find 
the collision pixel expectation for two bounding boxes
#define NUM_OF_TIME 1000000

#include<stdio.h>
#include<stdlib.h>

int gen_rand (int, int);

double min (double, double);  //find the smaller of two values

double max (double, double);  //find the bigger of two values

double intersectArea (double, double, double, double, double, double, double, double, double);  //define the
//top left corner position of two rectangles with two width and
//height
//values, calculate the intersection area

int

main (int argc, char **argv)
{
  int aCoverage, bCoverage;  //aCoverage is the percentage
   coverage for first box, and bCoverage is the percentage
   coverage for the other box

  double ax1, ay1, ay2;
  double total = 0.0;

  aCoverage = 1000000 * atof (argv[1]);  // we pass the percentage
   coverage as two arguments
  bCoverage = 1000000 * atof (argv[2]);

for (int time = 0; time < NUM_OF_TIME; time++)
{
    ax1 = (double) gen_rand (0, (int) (aCoverage / 1000));
    ay1 = (double) gen_rand (0, 1000 - (int) ((aCoverage - 1000 * ax1) / (1000 - ax1)));
    ay2 = ay1 + (double) (aCoverage - 1000 * ax1) / (double) (1000 - ax1);

    int topLeftCornerX, topLeftCornerY;
    double bx1, by1, by2;

    topLeftCornerX = gen_rand (0, 1000); // we generate the second box’s top left corner as a random position inside the first box
    topLeftCornerY = gen_rand (0, 1000);

    bx1 = (double) gen_rand (0, (int) (bCoverage / 1000));
    by1 = (double) gen_rand (0, 1000 - (int) ((bCoverage - 1000 * bx1) / (1000 - bx1)));
    by2 = by1 + (double) (bCoverage - 1000 * bx1) / (double) (1000 - bx1);

    total += intersectArea (0, 1000, ax1, 1000, (double) topLeftCornerX,
                            (double) topLeftCornerY, bx1, 1000);
    total += intersectArea (0, 1000, ax1, 1000, (double) topLeftCornerX + bx1,
                            (double) topLeftCornerY - 1000.0 + by2, 1000.0 - bx1,
(double) (bCoverage - 1000 * bx1) / (double) (1000 - bx1));

total += intersectArea (ax1, ay2, 1000 - ax1, ay2 - ay1,
(double) topLeftCornerX, (double) topLeftCornerY, bx1, 1000);

total += intersectArea (ax1, ay2, 1000 - ax1, ay2 - ay1,
(double) topLeftCornerX + bx1,
(double) topLeftCornerY - 1000.0 + by2, 1000.0 - bx1,
(double) (bCoverage - 1000 * bx1) / (double) (1000 - bx1));

printf ("Expected Collision Pixel number is %6.4f\n", total / NUM_OF_TIME);

return 0;

int gen_rand (int min, int max)
{
    int n;
    int bucket_size = max - min + 1;
    n = (rand () % (bucket_size)) + min;
    return (n);
}
double min (double a, double b) {
    if (a > b)
        return b;
    else
        return a;
}

double max (double a, double b) {
    if (a > b)
        return a;
    else
        return b;
}

double intersectArea (double x1, double y1, double w1, double h1, double x2,
```c
    double y2, double w2, double h2)
{
    double area = 0.0;

    if (x2 >= x1 && x2 <= (x1 + w1) && y2 <= y1 && y2 >= (y1 - h1))
        area = (min ((x2 + w2), (x1 + w1)) - x2) * (y2 - max (y2 - h2, y1 - h1));

    else if (x2 >= x1 && x2 <= (x1 + w1) && (y2 - h2) <= y1
            && (y2 - h2) >= (y1 - h1))
        area = (min ((x2 + w2), (x1 + w1)) - x2) * (y1 - y2 + h2);

    else if ((x2 + w2) >= x1 && (x2 + w2) <= (x1 + w1) && y2 <= y1
            && y2 >= (y1 - h1))
        area = (x2 + w2 - x1) * (y2 - max ((y2 - h2), (y1 - h1)));

    else if ((x2 + w2) >= x1 && (x2 + w2) <= (x1 + w1) && (y2 - h2)
            && y2 >= (y1 - h1))
        area = (x2 + w2 - x1) * (y1 - y2 + h2);

    else
        area = 0.0;

    return area;
}
```

Listing A.2: Random Two Boxes Calculation Expectation
/*----------*/

Author: Min Qi

Date: Nov 30, 2012

Description: This is a program that generates random locations for three non-overlapping boxes within a range of 100X100 bounding box. Then we put the three-boxes number2 to a random position on three-boxes number1, to check the estimated calculation of pixels.

Notice: in this program, the avatar is considered to be connected graph, so the three boxes have to be connected

*/

#define NUM_OF_TIME 100000

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int min(int, int);
int max(int, int);
int gen_rand(int, int);

int intersectArea (int, int, int, int, int, int, int, int);
    //define the
    //top left corner position of two rectangles with two width and height
    //values, calculate the intersection area

int* random_three_boxes(int); //this function will return an array of
    //locations, which are Ax1, Ay1, Ax2, Ay2, Bx1, By1, Bx2, By2, Cx1,
int main(){
clock_t start, end;
double runTime;
long totalArea=0;
long totalArea_prev=0;
start = clock();
int times=0;
while(times<NUM_OF_TIME){

int (*threebox1)[13] = random_three_boxes(gen_rand(1,3));

int (*threebox2)[13] = random_three_boxes(gen_rand(1,3));
    int rand_x = gen_rand(0,100);
    int rand_y = gen_rand(0,100);
    totalArea_prev +=(100- rand_x)*(100- rand_y);
    totalArea += intersectArea((*threebox1)[0], (*threebox1)[1],
                                (*threebox1)[2]-(*threebox1)[0],
                                (*threebox1)[3]-(*threebox1)[1],
                                (*threebox2)[0] + rand_x, (*threebox2)[1] + rand_y,
                                (*threebox2)[2]-(*threebox2)[0],
                                (*threebox2)[3]-(*threebox2)[1]);
    totalArea += intersectArea((*threebox1)[0], (*threebox1)[1],
                                (*threebox1)[2]-(*threebox1)[0],
                                (*threebox1)[3]-(*threebox1)[1],
                                (*threebox2)[0] + rand_x, (*threebox2)[1] + rand_y,
                                (*threebox2)[2]-(*threebox2)[0],
                                (*threebox2)[3]-(*threebox2)[1]);
}
}
45 (*threebox2)[4] + rand_x, (*threebox2)[5] + rand_y,
    (*threebox2)[6]-(*threebox2)[4],
    (*threebox2)[7]-(*threebox2)[5]);
46    totalArea += intersectArea((*threebox1)[0], (*threebox1)[1],
                    (*threebox1)[2]-(*threebox1)[0],
                    (*threebox1)[3]-(*threebox1)[1],
47 (*threebox2)[8] + rand_x, (*threebox2)[9] + rand_y,
    (*threebox2)[10]-(*threebox2)[8],
    (*threebox2)[11]-(*threebox2)[9]);
48    totalArea += intersectArea((*threebox1)[4], (*threebox1)[5],
                    (*threebox1)[6]-(*threebox1)[4],
                    (*threebox1)[7]-(*threebox1)[5],
49 (*threebox2)[0] + rand_x, (*threebox2)[1] + rand_y,
    (*threebox2)[2]-(*threebox2)[0],
    (*threebox2)[3]-(*threebox2)[1]);
50    totalArea += intersectArea((*threebox1)[4], (*threebox1)[5],
                    (*threebox1)[6]-(*threebox1)[4],
                    (*threebox1)[7]-(*threebox1)[5],
51 (*threebox2)[4] + rand_x, (*threebox2)[5] + rand_y,
    (*threebox2)[6]-(*threebox2)[4],
    (*threebox2)[7]-(*threebox2)[5]);
52    totalArea += intersectArea((*threebox1)[4], (*threebox1)[5],
                    (*threebox1)[6]-(*threebox1)[4],
                    (*threebox1)[7]-(*threebox1)[5],
53 (*threebox2)[8] + rand_x, (*threebox2)[9] + rand_y,
    (*threebox2)[10]-(*threebox2)[8],
    (*threebox2)[11]-(*threebox2)[9]);
totalArea += intersectArea((*threebox1)[8], (*threebox1)[9],
    (*threebox1)[10]-(*threebox1)[8],
    (*threebox1)[11]-(*threebox1)[9],
(*threebox2)[0] + rand_x, (*threebox2)[1] + rand_y,
    (*threebox2)[2]-(*threebox2)[0],
    (*threebox2)[3]-(*threebox2)[1]);
totalArea += intersectArea((*threebox1)[8], (*threebox1)[9],
    (*threebox1)[10]-(*threebox1)[8],
    (*threebox1)[11]-(*threebox1)[9],
(*threebox2)[4] + rand_x, (*threebox2)[5] + rand_y,
    (*threebox2)[6]-(*threebox2)[4],
    (*threebox2)[7]-(*threebox2)[5]);
totalArea += intersectArea((*threebox1)[8], (*threebox1)[9],
    (*threebox1)[10]-(*threebox1)[8],
    (*threebox1)[11]-(*threebox1)[9],
(*threebox2)[8] + rand_x, (*threebox2)[9] + rand_y,
    (*threebox2)[10]-(*threebox2)[8],
    (*threebox2)[11]-(*threebox2)[9]);
//printf("Area is now %ld vs %ld\n", totalArea, totalArea_prev);
free(threebox1);
free(threebox2);
times++;
}
printf("Area Previous is now %ld\n", totalArea_prev/NUM_OF_TIME);
printf("Area is now %ld\n", totalArea/NUM_OF_TIME);
end = clock();
runTime = (end-start) / (double)CLOCKS_PER_SEC ;
printf("run time is %.2f seconds", runTime);
int gen_rand(int min, int max) {
    int n;
    int bucket_size = max - min + 1;
    if (bucket_size > 1)
        { n = (rand () % (bucket_size)) + min; }
    else if (bucket_size == 1)
        { n = min; }
    else
        { n = max; }
    return n;
}

int* random_three_boxes(int type) {

    int *result=malloc(13*sizeof(int));
    int Ax1,Ay1,Ax2,Ay2,Bx1,By1,Bx2,By2,Cx1,Cy1,Cx2,Cy2,Area;
    switch (type)
        { case 1:

        }
Ax1 = 0;
Ay1 = 0;
Ax2 = gen_rand(1, 100);
Ay2 = gen_rand(Ay1+1, 99);
    Bx1 = Ax2;
By1 = gen_rand(Ay1, Ay2);
Bx2 = 100;
By2 = gen_rand(By1+1, 99);
Cx1 = gen_rand(Bx1, Bx2);
Cy1 = By2;
Cx2 = gen_rand(Cx1+1, 100);
Cy2 = 100;
Area = (Ax2-Ax1) * (Ay2 - Ay1) + (Bx2 - Bx1) * (By2 - By1) + (Cx2 - Cx1) * (Cy2 - Cy1);
result[0] = Ax1;
result[1] = Ay1;
result[2] = Ax2;
result[3] = Ay2;
result[4] = Bx1;
result[5] = By1;
result[6] = Bx2;
result[7] = By2;
result[8] = Cx1;
result[9] = Cy1;
result[10] = Cx2;
result[11] = Cy2;
result[12] = Area;
return result;

case 2:
    Ax1 = 0;
    Ay1 = gen_rand(0, 100);
    Ax2 = gen_rand(1, 100);
    Ay2 = gen_rand(Ay1+1, 100);
    Bx1 = Ax2;
    By1 = 0;
    Bx2 = 100;
    By2 = gen_rand(1, 100);
    Cx1 = gen_rand(Bx1, Bx2-1);
    Cy1 = By2;
    Cx2 = gen_rand(Cx1+1, 100);
    Cy2 = 100;
    Area = (Ax2-Ax1) * (Ay2 - Ay1) + (Bx2 - Bx1) * (By2 - By1)
    + (Cx2 - Cx1) * (Cy2 - Cy1);
    result[0] = Ax1;
    result[1] = Ay1;
    result[2] = Ax2;
    result[3] = Ay2;
    result[4] = Bx1;
    result[5] = By1;
    result[6] = Bx2;
    result[7] = By2;
    result[8] = Cx1;
    result[9] = Cy1;
    result[10] = Cx2;
result[11] = Cy2;
result[12] = Area;

return result;

case 3:
    Ax1 = 0;
    Ay1 = 0;
    Ax2 = gen_rand(0, 98);
    Ay2 = gen_rand(1, 100);
    Bx1 = Ax2;
    By1 = gen_rand(Ay1, Ay2);
    Bx2 = gen_rand(Bx1+1,100);
    By2 = 100;
    Cx1 = Bx2;
    Cx2 = 100;
    Cy2 = gen_rand(By1, 100);
    Cy1 = gen_rand(0, Cy2);
    Area = (Ax2-Ax1) * (Ay2 - Ay1) + (Bx2 - Bx1) * (By2 - By1)
        + (Cx2 - Cx1) * (Cy2 - Cy1);
    result[0] = Ax1;
    result[1] = Ay1;
    result[2] = Ax2;
    result[3] = Ay2;
    result[4] = Bx1;
    result[5] = By1;
    result[6] = Bx2;
    result[7] = By2;
    result[8] = Cx1;
result[9] = Cy1;
result[10] = Cx2;
result[11] = Cy2;
result[12] = Area;
return result;
default:
  Ax1 = 0;
  Ay1 = 0;
  Ax2 = gen_rand(0, 98);
  Ay2 = gen_rand(1, 100);
  Bx1 = Ax2;
  By1 = gen_rand(Ay1, Ay2);
  Bx2 = gen_rand(Bx1+1,100);
  By2 = 100;
  Cx1 = Bx2;
  Cx2 = 100;
  Cy2 = gen_rand(By1, 100);
  Cy1 = gen_rand(0, Cy2);
  Area = (Ax2-Ax1) * (Ay2 - Ay1) + (Bx2 - Bx1) * (By2 - By1) + (Cx2 - Cx1) * (Cy2 - Cy1);
  result[0] = Ax1;
  result[1] = Ay1;
  result[2] = Ax2;
  result[3] = Ay2;
  result[4] = Bx1;
  result[5] = By1;
  result[6] = Bx2;
result[7] = By2;
result[8] = Cx1;
result[9] = Cy1;
result[10] = Cx2;
result[11] = Cy2;
result[12] = Area;
return result;

int min(int a, int b) {
    if (a > b) return b;
    else return a;
}

int max(int a, int b) {
    if (a < b) return b;
    else return a;
}

int intersectArea(int x1, int y1, int w1, int h1, int x2, int y2, int w2, int h2) {
    int area = 0;

    if (x2 >= x1 && x2 <= (x1 + w1) && y2 <= y1 && y2 >= (y1 - h1))
area = (min ((x2 + w2), (x1 + w1)) - x2) * (y2 - max (y2 - h2, y1 - h1));

else if (x2 >= x1 && x2 <= (x1 + w1) && (y2 - h2) <= y1 && (y2 - h2) >= (y1 - h1))
area = (min ((x2 + w2), (x1 + w1)) - x2) * (y1 - y2 + h2);

else if ((x2 + w2) >= x1 && (x2 + w2) <= (x1 + w1) && y2 <= y1 && y2 >= (y1 - h1))
area = (x2 + w2 - x1) * (y2 - max ((y2 - h2), (y1 - h1)));

else if ((x2 + w2) >= x1 && (x2 + w2) <= (x1 + w1) && (y2 - h2) <= y1 && (y2 - h2) >= (y1 - h1))
area = (x2 + w2 - x1) * (y1 - y2 + h2);

else
area = 0.0;

return area;
}

Listing A.3: Random Three Boxes Calculation Expectation

/*********
Author: Min Qi
Date: Dec 12, 2012
Description: This is a program that uses a serialized way to find the optimal two bounding boxes for a sprite
///
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define M 1024
#define N 1024

#define blocksPerGrid M
#define threadsPerBlock N

int findLowest(int *b){
    int index=M-1;
    for(int i=M-1;i>-1;i--)
    {
        if(b[i]!=0)
            index=i;
    }
    return index;
}

int findHighest(int *b){
    int index=0;
    for(int i=0; i<M; i++)
    {
        if(b[i]!=0)
            index=i;
    }
    return index;
}
int main()
{
    clock_t t_ini, t_fin;
    int b[M*N];
    for(int i = 0; i < M*N; i++)
    {
        b[i] = 1;
    }

    int *highest_c;
    int *lowest_c;
    int highest_left[blocksPerGrid], lowest_left[blocksPerGrid],
        highest_right[blocksPerGrid], lowest_right[blocksPerGrid];
    int height_left[blocksPerGrid],
        height_right[blocksPerGrid], areas[blocksPerGrid];

    t_ini = clock();
    //b=(int*) malloc(M*N*sizeof(int));
    lowest_c = (int*) malloc(blocksPerGrid*sizeof(int));
    highest_c = (int*)malloc(blocksPerGrid*sizeof(int));

    /* for(int i=0;i<N*M;i++){
        if((int)(random()%2)==0)
            b[i]=0;
    }*/

    return index;
}
else
  b[i]= 1;
*/

for (int i=0;i<M;i++){
  highest_c[i] = findHighest(&b[i*M]);
  printf("%d ", highest_c[i]);
}
printf("\n");

for(int i=0;i<blocksPerGrid;i++){
  lowest_c[i] = findLowest(&b[i*M]);
  printf("%d ", lowest_c[i]);
}

for(int i=0;i<blocksPerGrid;i++){
  highest_left[i]=highest_c[i];
  lowest_left[i]=lowest_c[i];
  highest_right[i]=highest_c[blocksPerGrid-1-i];
  lowest_right[i]=lowest_c[blocksPerGrid-1-i];

  if(i!=0&&highest_left[i]<highest_left[i-1])
    highest_left[i]=highest_left[i-1];
  if(i!=0&&lowest_left[i]>lowest_left[i-1])
    lowest_left[i]=lowest_left[i-1];
  if(i!=0&&highest_right[i]<highest_right[i-1]) // get the accumulated highest/lowest
    highest_right[i]=highest_right[i-1];
highest_right[i]=highest_right[i-1];
if (i!=0&&lowest_right[i]>lowest_right[i-1])
    lowest_right[i]=lowest_right[i-1];
}
for (int i=0;i<blocksPerGrid; i++){
    height_left[i]=highest_left[i]-lowest_left[i]+1;  //calculate
    // the height by subtract lowest from highest
    height_right[i]=highest_right[i]-lowest_right[i]+1;
}
areas[0]=M*N;
for (int i=1;i<blocksPerGrid;i++){
    areas[i]=i*height_left[i] + (N-i)*height_right[M-i];
    //calculate area by width*height (left box + right box)
}
t_fin=clock();
printf("the total click is %d and time spent is %f seconds",
t_fin-t_ini, ((float)(t_fin-t_ini))/CLOCKS_PER_SEC);
free(highest_c);
free(lowest_c);
}

Listing A.4: Random Three Boxes Calculation Expectation
Description: This is a program that parallelize the procedure of finding optimal two boxes for an avatar

```c
#include "../header.h"
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <limits.h>

#define M 8
#define N 8

const int threadsPerBlock = N;
const int blocksPerGrid = M;

__global__ void calcHeight(int *lowest, int *highest, int *heights) {
    int i = blockIdx.x;
    if (i < M)
        heights[i] = highest[i] - lowest[i] + 1;
}

__global__ void scanHighest(int *outdata, int *indata)
```
```c
int tid = blockIdx.x;
outdata[tid] = indata[tid];
for (int offset = 1; offset < N; offset *= 2) {
    if (tid >= offset)
        outdata[tid] = ((outdata[tid] > outdata[tid - offset]) ?
                        outdata[tid]: outdata[tid - offset]);
    __syncthreads();
}
}

__global__ void scanLowest(int *outdata, int *indata) {
    int tid = blockIdx.x;
    outdata[tid] = indata[tid];
    for (int offset = 1; offset < N; offset *= 2) {
        if (tid >= offset)
            outdata[tid] = ((outdata[tid] < outdata[tid - offset]) ?
                            outdata[tid]: outdata[tid - offset]);
        __syncthreads();
    }
}

__global__ void findArea(int *leftHeight, int *rightHeight, int *result) {
```
int tid=threadIdx.x;
result[0]=M*N;
if(tid>0&&tid<N){
result[tid]= tid*leftHeight[tid]+ (N-tid)*rightHeight[N-1-tid];
}

__global__ void findLowest(int *b, int *c){
__shared__ int cache[threadsPerBlock];
int tid= threadIdx.x + blockIdx.x *blockDim.x;
int cacheIndex= threadIdx.x;
if(b[tid]==0) cache[cacheIndex]=N+1;
else cache[cacheIndex]=threadIdx.x;
__syncthreads();

int i = blockDim.x/2;
while(i!= 0){
    if(cacheIndex<i)
    { if(cache[cacheIndex]>cache[cacheIndex+i])
        cache[cacheIndex]=
        cache[cacheIndex+i];
    }
    __syncthreads();
}
i /=2;
}

if(cacheIndex ==0)
{

c[blockIdx.x] = cache[0];
}

__global__ void findHighest(int *b, int *c){

__shared__ int cache[threadsPerBlock];

int tid= threadIdx.x + blockIdx.x *blockDim.x;
int cacheIndex= threadIdx.x;

if(b[tid]==0) cache[cacheIndex]=-1;
else cache[cacheIndex]=threadIdx.x;

int i = blockDim.x/2;
while(i!= 0){
    if(cacheIndex<i)
    {
        if(cache[cacheIndex]<cache[cacheIndex+i])
        
            cache[cacheIndex]=cache[cacheIndex+i];
    
    }
__syncthreads();
i /=2;
}
if(cacheIndex ==0)
{
    c[blockIdx.x] = cache[0];
}
}

int main(){
clock_t t_ini, t_fin;

int b[M*N]
    ={1,0,0,0,1,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,
    0,1,0,0,1,1,1,0,0,1,0,0,0,0,1,1,1,0,1,0,1,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0};  //give an 8 X 8 sample image

int highest_c_fromright[N], lowest_c_fromright[N];

int *dev_b;
int *dev_c_low, *dev_c_high;
int *highest_c;
int *lowest_c;

int highest_left[blocksPerGrid], lowest_left[blocksPerGrid], highest_right[blocksPerGrid], lowest_right[blocksPerGrid];
int height_left[blocksPerGrid],
    height_right[blocksPerGrid], areas[blocksPerGrid];

int *dev_c;
lowest_c = (int*) malloc(blocksPerGrid*sizeof(int));
highest_c = (int*) malloc(blocksPerGrid*sizeof(int));

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

HANDLE_ERROR(cudaMalloc((void**)&dev_b, M*N*sizeof(int)));
HANDLE_ERROR(cudaMalloc((void**)&dev_c_low, M*sizeof(int)));
HANDLE_ERROR(cudaMalloc((void**)&dev_c_high, M*sizeof(int)));

HANDLE_ERROR(cudaMemcpy(dev_b, b, M*N*sizeof(int), cudaMemcpyHostToDevice));
findHighest<<<blocksPerGrid, threadsPerBlock>>>(dev_b, dev_c_high);
findLowest<<<blocksPerGrid, threadsPerBlock>>>(dev_b, dev_c_low);
HANDLE_ERROR(cudaMemcpy(highest_c, dev_c_high, blocksPerGrid*sizeof(int), cudaMemcpyDeviceToHost));
HANDLE_ERROR(cudaMemcpy(lowest_c, dev_c_low,
        blocksPerGrid*sizeof(int), cudaMemcpyDeviceToHost));

t_fin = clock();
for(int i=0; i<blocksPerGrid; i++){
    printf("%d ", highest_c[i]);
}
printf("\n");
for(int i=0; i<blocksPerGrid; i++){
    printf("%d ", lowest_c[i]);
}

HANDLE_ERROR(cudaEventRecord(stop,0));
HANDLE_ERROR(cudaEventSynchronize(stop));

    float elapsedTime;

    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start,
        stop));

    printf("Time to generate: %3.1f ms\n",elapsedTime);

HANDLE_ERROR(cudaEventDestroy(start));
HANDLE_ERROR(cudaEventDestroy(stop));
HANDLE_ERROR(cudaFree(dev_b));

printf("The time in system clock takes %d clicks %f seconds\n",
        (t_fin-t_ini), ((float)(t_fin-t_ini))/CLOCKS_PER_SEC);

for(int i=0; i<blocksPerGrid; i++){
    highest_c_fromright[i]=highest_c[blocksPerGrid-1-i];
lowest_c_fromright[i]=lowest_c[blocksPerGrid-1-i];

HANDLE_ERROR(cudaMalloc((void**)&dev_c,N*sizeof(int)));

HANDLE_ERROR(cudaMemcpy(dev_c_low, lowest_c, N*sizeof(int),
              cudaMemcpyHostToDevice));

scanLowest<<<N,1>>>(dev_c, dev_c_low);

HANDLE_ERROR(cudaMemcpy(lowest_left, dev_c, N*sizeof(int),cudaMemcpyDeviceToHost));

cudaFree(dev_c);

HANDLE_ERROR(cudaMalloc((void**)&dev_c,N*sizeof(int)));

HANDLE_ERROR(cudaMemcpy(dev_c_high, highest_c, N*sizeof(int),
              cudaMemcpyHostToDevice));

scanHighest<<<N,1>>>(dev_c, dev_c_high);

HANDLE_ERROR(cudaMemcpy(highest_left, dev_c, N*sizeof(int),cudaMemcpyDeviceToHost));

cudaFree(dev_c);

HANDLE_ERROR(cudaMalloc((void**)&dev_c,N*sizeof(int)));

HANDLE_ERROR(cudaMemcpy(dev_c_low, lowest_c_fromright,
              N*sizeof(int), cudaMemcpyHostToDevice));

scanLowest<<<N,1>>>(dev_c, dev_c_low);

HANDLE_ERROR(cudaMemcpy(lowest_right, dev_c, N*sizeof(int),cudaMemcpyDeviceToHost));

cudaFree(dev_c);
HANDLE_ERROR(cudaMalloc((void**)&dev_c, N*sizeof(int)));  
HANDLE_ERROR(cudaMemcpy(dev_c_high, highest_c_fromright, N*sizeof(int), cudaMemcpyHostToDevice));  
scanHighest<<<N,1>>>(dev_c, dev_c_high);  
HANDLE_ERROR(cudaMemcpy(highest_right, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost));  
cudaFree(dev_c);  
HANDLE_ERROR(cudaMalloc((void**)&dev_c, N*sizeof(int)));  
HANDLE_ERROR(cudaMemcpy(dev_c_low, lowest_left, N*sizeof(int), cudaMemcpyHostToDevice));  
HANDLE_ERROR(cudaMemcpy(dev_c_high, highest_left, N*sizeof(int), cudaMemcpyHostToDevice));  
calcHeight<<<M,1>>>(dev_c_low, dev_c_high, dev_c);  
HANDLE_ERROR(cudaMemcpy(height_left, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost));  
cudaFree(dev_c);  
cudaFree(dev_c_low);  
cudaFree(dev_c_high);  
HANDLE_ERROR(cudaMalloc((void**)&dev_c, N*sizeof(int)));  
HANDLE_ERROR(cudaMalloc((void**)&dev_c_low, N*sizeof(int)));  
HANDLE_ERROR(cudaMalloc((void**)&dev_c_high, N*sizeof(int)));
HANDLE_ERROR(cudaMemcpy(dev_c_low, lowest_right, N*sizeof(int), cudaMemcpyHostToDevice));
HANDLE_ERROR(cudaMemcpy(dev_c_high, highest_right, N*sizeof(int), cudaMemcpyHostToDevice));
calcHeight<<<N,1>>>(dev_c_low, dev_c_high, dev_c);
HANDLE_ERROR(cudaMemcpy(height_right, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost));
cudaFree(dev_c);
cudaFree(dev_c_low);
cudaFree(dev_c_high);
printf("the height from left to right:\n");
for(int i=0;i<blocksPerGrid;i++){
    printf("%d\t", height_left[i]);
}
HANDLE_ERROR(cudaMalloc((void**)&dev_c, N*sizeof(int)));
HANDLE_ERROR(cudaMalloc((void**)&dev_c_low, N*sizeof(int)));
HANDLE_ERROR(cudaMalloc((void**)&dev_c_high, N*sizeof(int)));
HANDLE_ERROR(cudaMemcpy(dev_c_low, height_left, N*sizeof(int), cudaMemcpyHostToDevice));
HANDLE_ERROR(cudaMemcpy(dev_c_high, height_right, N*sizeof(int), cudaMemcpyHostToDevice));
findArea<<<1,N>>>(dev_c_low, dev_c_high, dev_c);
HANDLE_ERROR(cudaMemcpy(areas, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost));
for(int i=1;i<blocksPerGrid;i++){
    printf("%d \t",areas[i]);
}
cudaFree(dev_c);
cudaFree(dev_c_low);
cudaFree(dev_c_high);
free(highest_c);
free(lowest_c);