IRAQI Academic Scientific Journals

Al-Rafidain Engineering Journal (AREJ)
Vol.25, No.1, June 2020, pp. 1-11

ISSN (Print) 1813-0526;
ISSN (Online) 2220-1720

Voxelization Parallelism Using CUDA Architecture

Sura Nawfal Alrawy
sura.nawfal@uomosul.edu.iq

Fakhrulddin Hamid Ali
fhali_a@yahoo.com

Computer Engineering Departementand, Collage of Engineering, University of Mosul

Received:1/1/ 2020 Accepted:20/2/ 2020

ABSTRACT

The voxelization process is an essential stage in three dimensional (3D) graphics pipeline. Its implementation should precede displaying objects in the pipeline. In this paper, different Voxelization algorithms are modified and parallelized to accelerate the operation of this stage. The 3D Digital Differential Analyzer (DDA) algorithm is used for line voxelization. This algorithm is utilized in triangle filling using the scan-line and the edge-function algorithms. The first one is designed to produce lines in parallel while the second can produce voxels. All these algorithms are parallelized using CUDA architecture and implemented on GPU processor. The actual implementation of these algorithms is examined and optimized according to the occupancy and block size metrics. The experimental results show that the acceleration amount of 3D DDA was about 4352x max compared to the OpenGL implementation, and the edge function implementation has been executed at a higher speed than the scan-line for object triangles voxelization.

Keywords:

GPU; CUDA; Edge-function; Parallel implementation; voxelization.

https://rengj.mosuljournals.com
Email: alrafidain_engjournal1@uomosul.edu.iq
==========

1. INTRODUCTION

Nowadays, as the continuous growth in 3D graphics field, many theoretical solutions have been proposed to develop and accelerate the overall graphics pipeline and especially the scan conversion or rasterization stage. This stage converts a scene composed of triangle meshes into a regularly spaced grid points, it is a computation-intensive process,NVIDIA considers it as the crown jewel of the hardware graphics pipeline[1].

The 3D extension of this process is called*voxelization*which belongs to *voxel* that is analogous to the*pixel* in 2D image space. The voxelization is accomplished using a 3D scan conversion to generate a discrete surface of a voxelizedobject; it differs from scan conversion that concerns with filling 2D triangles or 2D projection of 3D triangles[2], their algorithms are more abundance in literature than a 3D scan conversion. Therefore we will focus on explaining 3D scan conversion only.In such processes the speed is required in generatingvoxels; some researches accelerated the operation using the standard Bresenham algorithm that has been adopted for a long time[3],[4],[5] and [6]. While other researches tried to use other algorithms like DDA [7],[8]and [9]. In[3], the early 3D version of Bresenham algorithm has been proposed,the algorithm is only implemented as Simulink using C programming and the assembly language. Articles [4] and [5] tried to accelerate the 3D Bresenham algorithm on FPGA platform, A speed of 68M pixels\sec is obtained using Spartan3E FPGA kit [4]. While the authors in [5] have used the Zynq-7000; their methods involve partitioning each line into number of segments drawn simultaneously. However, they test their algorithm to scan few points and they got a good performance among other previous works.

Another set of works accelerated the DDA algorithm for line scan conversion,A multi symmetry, in certain type of lines, is exploited to parallelize the scan conversion algorithms [7], where the line is divided into equal lengths

divisions, The algorithm was proposed to compute only the first division, and the other segments simply repeated from this pre-computation. However, this work has been applied on certain line orientation (endpoints) and the overall performance depends on the probability of a line having multi symmetry.In 2011, a 3D DDA algorithm has been implemented using FPGA[8]. The unit can produce pixels at a speed of 120M pixels\sec. assuming the loss of small time in computing the increment values.

Also, some researches accelerated the ray traversal method that adopted the 3D DDA algorithm [9]. The authors used CUDA architecture on different GPU cards; they get significant acceleration. McGuire and Mars [9] presented an efficient implementation for screen space 3D ray-tracing, their implementation cost 1.2 mille second to render 1920x1080 scene resolution on NVIDIA GeForce Titan.

In terms of the polygon scan conversion; some literature has dealt with filling 2D polygons or the 2D projections of 3D polygons, while the works that dealt with 3D scan-conversion were relatively few. The parallelism in these works has been accomplished using the block-based scan conversion by partitioning the displayed area into blocks and processing them simultaneously on modern multi-core CPU [10],[11]. In [10] the space area of a triangle is partitioned into 8x8 blocks then the 3D scan conversion is applied from the top left corner to the bottom. The researchers implemented the algorithm using Handel-C then translate it into VHDL code to verify the design using ModelSim. In this method some blocks would be empty making the corresponding threads remain idle many times. This problem was dealt with in [11] to reduce the effect of empty blocks and unnecessary calculations by combining two methods adaptive and bisector algorithms. They tested their implementation on Head and Statue models having 600,000 vertices with five different cases according to the distance from the camera, the maximum FPS obtained was about 564 frame\ sec.

NVIDIA proposed an efficient CUDA-based rendering model [1], where a complete software rasterization pipeline has been implemented on a GPU. They tiled the triangle primitive and made each warp rasterize a single triangle; the performance of which is a factor of 2-8x compared to the hardware graphics pipeline.

The 2D scan line method for triangle rasterization has been used in [12] and [13]. The researchers implemented the algorithm on FPGA, the maximum speed obtained in [12] is about 50M pixel\sec., using the classical 2D scan-line algorithm. While the work in [13] has improved the 2D scan conversion by using the midpoint traversal which reduces the number of unnecessary points need to traverse, the performance results on different triangle orientations show the efficiency of this method in comparison with other methods, such as Bounding Box Traversal, Central-line Traversal, and Tiled Traversal.

Although these works can apply rendering algorithms, the performance is still not satisfying compared to the power of the current GPUs and it needs more improvements especially after the graphics vendors began to provide programmability at different stages after they were fixed units on chips.However, more acceleration is needed to improve the performance by implementing the algorithm efficiently in parallel manner.

In addition to this introduction, this paper contains four other sections. Section 2 presents the theoretical bases of the work. The proposed parallel voxelizationis fully explained in section 3. The obtained results and their corresponding discussions are included in section 4. Finally, section 5 concludes this paper.

2. THE THEORETICAL BASES

The voxelization takes advantage of the spatial or coherence property in an object has to be displayed. This property means that one part of the object is related to another part of that object in some way. So this relationship is used in voxel calculation to reduce the processing, where only the end vertices for line, or the three vertices for a triangle, are stored and the scan conversion can create the whole needed voxels for single scan line or between successive scan lines[12].

In this section, the related theory of a straight line and a triangle voxelization is introduced with their sequential algorithms.

2.1 3D line voxelization

There are two standard algorithms for 3D line scan conversion, Bresenham and DDA algorithms[14].In this paper the DDA algorithm is presented. The implementation of itis done by linear interpolation of variables over an interval between start and endpoints. It needs a floating point operation in its computations. The 3D version is accomplished by considering the line segment whose voxels require to be generated in 3D space, so for each voxel, the z value is calculated in addition to the x and y values, hence, the algorithm works in object space rather than in image space.

The sequential DDA scans a line vertex after anotherbased on calculating either *dy* or *dx* differences. It can work on lines with different slopes less or great than one as well as the positive or negative slops. Simply the line equation with slope is exploited here, as (x_n=x_{n-1}+m) so, all the voxels that belong to this line should satisfy this equation. Same calculations are carried out to determine voxel positions at each coordinate then the same process is repeated along the line. Therefore three slopes should be calculated, one for each coordinate and one of them would be equal to one and the others may be less or great than one due to the line type.

2.2Polygonvoxelization

The 3D object is already stored as a polygons mesh, a triangle is considered as a fundamental primitive in most modern GPU cards since any polygon can be divided into many triangles[1],[14].So the triangle scan conversion is treated in this paper. The scan conversion of a triangle involves filling all area bounded by its edges and lie inside its boundaries. Therefore, this stage should be implemented carefully to render the whole object in 3D space in a correct manner. This process can be categorized into two main approaches;scan-line and edge function approaches[12],[14], both are based on the previous line scan conversion algorithms.

2.2.1 Scan-line algorithm

This algorithm is considered a classical version that is still in use, since it offers an efficient triangle traversal by walking through the triangle from top to bottom and digitalizes successive horizontal line by another. Each line is called *scan-line*, and the internal part of a scan line is called *span.* This method is also called *edge walking* or *fast scan conversion* algorithm.
Fig.1 shows two successive spans of a triangle where each one has a constant *y* value.

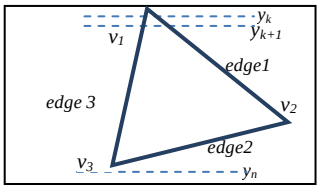


Fig. 1 A polygon scan conversion (up to down) *k*: is a scan line number

The 3D scan-line is extended from the 2D version, where at each span the *z* dimension is also interpolated. We can explain this method by partitioning it into four steps.

i) Vertices sorting

Firstly the triangle vertices (v_1, v_2, v_3) should be sorted according to its *y* values from least to most in a clockwise direction; this is to distinguish the top, middle and bottom vertices. The sorting of the vertices is made independently of the z values and it sorts the triangle sides themselves.

ii) Slopes calculation

After the triangle edges are sorted along the *y*-axis, the slope of each triangle edge can be found from the line formula by[14]:

$$dy_i = \frac{v_{i+1}.y - v_i.y}{v_{i+1}.y - v_i.y} \quad (1)$$

$$dx_i = \frac{v_{i+1}.y - v_i.y}{v_{i+1}.x - v_i.x} \quad (2)$$

$$dz_i = \frac{v_{i+1}.y - v_i.y}{v_{i+1}.z - v_i.z} \quad (3)$$

Where: i :is the triangle vertex counter from 0 to 2.

There are three slopes should be calculated for three edges, these slopes are needed to voxelize the triangle sides, and thereby the whole span points, where these intersections represent the start and end points of the span. The slope transition is needed when the span reaches to v_2 vertex, as in Fig.1, where edge 1 should be dispatched by edge 2 and its slopes will be considered in span end calculations.
In 3D scan conversion, the *z* increment should be calculated for each span line using the equation[14]

$$z_{inc} = \frac{span\_max_k.z - span\_min_k.z}{span\_max_k.x - span\_min_k.x} \quad (4)$$

iii) Finding the scan line intersection

To move down from span to another, we should find the intersection of the scan line with triangle boundaries. The *(x,y,z)* coordinate can be calculated based on the previous slope values. The change in *y* coordinate between two successive spans is one step as the equation:

$$span_{k+1}.y - span_k.y = 1 \quad (5)$$

While the new intersection *x* and *z* values are determined by the *x*, *y* intersection values of the preceding span x_k, z_k as[14]:

$$span_{k+1}.x = span_k.x + \frac{1}{dx_i} \quad (6)$$

$$span_{k+1}.z = span_k.z + \frac{1}{dz_i} \quad (7)$$

Where each successive (*x*, *y*) intercept can thus be calculated by adding the inverse of

the slope, noting that the slope may be negative or positive depending on which is greater (x_{k+1} or x_k) or (z_{k+1} or z_k). So each time we move to a new span, the *x* and *z* values can be incremented or decremented based on the reciprocal value of the slope.

In Fig.2, one can note that the middle span, which passes through the middle vertexv_2, can separate the triangle into two parts top and bottom. So a slope transition is needed at this span where one of the slopes should be changed making the procedure having two different loops, each of them with different span number according to the current active edges. Two cases are produced here according to the orientation of the triangle where the position of the second vertex v_2 may be to the left or to the right of other vertices as shown in Fig.2.

Simply, the number of spans can be represented by the difference between the y-coordinates (*dy*) of the two ends of a triangle edge[2],[14]:

$$dy = y_2 - y_1 \qquad (8)$$

Here, only the two smallest *dy* of a triangle are needed, where the third one represents the sum of these two differences.

$$dy_3 = dy_2 + dy_1 \qquad (9)$$

Whenever the counter value for calculating the spans of the first part becomes equal to or greater than *dy*, we switch to another counter and one of the edges is removed from the active edges and replaced by another. Thereby, we increment or decrement the current *x* and *z* intersection depending on the sign of the new slopes and increment the y coordinate by one, then the counter is increased by one till reaching the second *dy* spans.

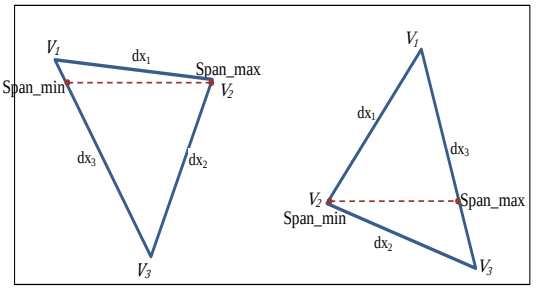


Fig.2 Span line intersections

iv) Filling the span

After finding the intersections of each scan-line with the current active triangle sides, the end and start points of the span are being provided and here, we further decompose the span into pixels using one of line scan conversion algorithms to fill in the span. The 3D DDA is used in this paper,it produces pixels which are filled in between the pairs of intersections horizontally from left to right.

2.2 Edge function testing

Edge function is a linear function that can be used to classify points on a plane according to its location, showing if the voxel lies above, on, or below the vector. Some sources refer to this method as a *half-space function* since it divides the region into two halves based on the considered edge, and others refer to it as a *bounding box* since it checks all the voxels in the bounded box surrounding a triangle [11],[13].

An edge function is defined by computing the perpendicular dot product *PerpDot*between a vector and the perpendicular one of the other vector and passes through the tested point [13].

Fig.3 explains this process where the edge function is the implicit equation, $ax + by + c = 0$, of the vector through the two points A and B.

.

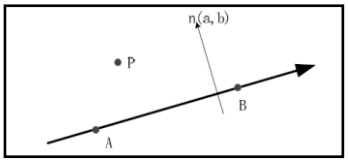


Fig.3 Edge function representation

So the *PerpDot* product can be applied on $\overline{A.B}$ and $\overline{A.P}$ as followed[13]:

$$perpDotP(\overline{AB}, \overline{AP}) = \overline{AB}^{\perp}.\overline{AP} \qquad (10)$$

Where *P(x,y)* is the tested point and the perpendicular vector on AB is shown as the normal $\bar{n}$ which has an inverse slope of AB, therefore the final edge function can be written as[11]:

$$E(x,y) = (x_1 - x_0)(y - y_0) - (y_1 - y_0)(x - x_0) \qquad (11)$$

So the function yields three possible outputs based on the input point *P(x, y)*:

- *E(x, y)* = 0 if point *P* is on the line.
- *E(x, y)*> 0 if point *P* is above the line in the same direction of the normal.
- *E(x, y)*< 0 if point *P* is below the line in the opposite side of the normal.

In triangle rasterization, there are several mathematical approaches to find the inner pixels and they all investigate the question of how to represent edges. Exploiting the edge function in a triangle rasterization involves checking each point inside the triangle against all its three edges as shown in Fig.4.

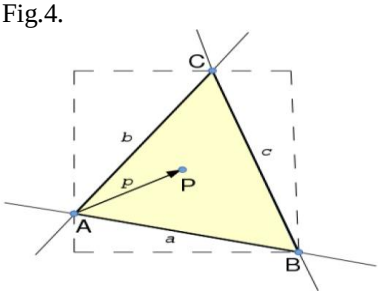


Fig.4 Triangle edge equations

Therefore, for each point there should be three edge equations. In general, a pixel could be inside the triangle if and only if the results of all the three functions are greater or equal than zero, otherwise, the pixel lies outside the triangle.

Surely, the method would not be effectual if all the pixels in the screen are tested, so the bounded box is used to limit the tested voxels as shown in Fig.4. This box is specified according to the minimum and maximum triangle coordinates values.

However, the above rasterizer requires many calculations for each per-pixel, three times of (three multiplications and eight subtractions as a 3D space). So optimization has been added to this approach [13] using the DDA method to find the neighboring pixels; this is to reduce the computations of the three coordinates. Thus, once the edge functions have been created and evaluated for a sample point, any of the four neighboring pixels can be evaluated using simple additions/subtractions operation. Another optimization on this algorithm has been proposed in [15] using the tile-based approach which is considered in some modern graphics card.

3. THE IMPLEMENTATION OF PARALLEL VOXELIZATION

In many 3D scene fillings, the process is performed voxel by voxel. Although mapping the vertex and the process of line traversal seem to be simple, the scan conversion performance largely depends on the implemented algorithm and its optimization level. It has a large number of individual visibility tests between the voxel and a triangle, so the algorithm iterations having many calculations need to be executed many times. Such algorithm requires more efforts to make it parallel and optimized at the same time, taking the possibilities of the GPU hardware into account and how to adapt it onto CUDA (Compute Unified Device Architecture) programming model. In this section, we intend to explore how far our voxelization implementation can be achieved in terms of performance on CUDA.

3.1 GPU-based line segment algorithm

With a parallel computer, we can find voxel positions along a line path simultaneously by separating the computations among the available streaming processors. This section shows how to modify scan conversion methods for 3D lines so that they can run on a parallel computer.

As discussed previously, there are two main serial algorithms for straight-line scan converting: DDA and Bresenham. The Bresenham's algorithm has become common because of its integer arithmetic operations since oldest processors could not easily do floating-point arithmetic. There were no floating-point math and no multiply\divide. This was thousands of times slower than doing simple integer arithmetic [9].

Nowadays, all processors (GPUs or even CPUs) can do SIMD (Single Instruction Multiple Data) operations and they use floating-point vector extensions. The current GPUs support both integer and floating-point operations, and they consume the same number of clock cycles for both integer and float operation [16]. Hence, the obstacles of unusing the DDA algorithm have vanished, so we expected that the DDA is more suitable for parallel implementation than the Bresenham algorithm.

The rasterization of a 3D line was not easy to implement as initially thought. Firstly, we tried to parallelize the Bresenham algorithm, but we found that this algorithm has more branches and dependency in its instructions. The creation of new voxel depends on the error value which is accumulated from other preceding voxel calculation. It changes under the condition if it is greater or less than zero, therefore, its implementation is difficult as SIMD. This difficulty can be dissolved by partitioning the line into many segments having the same slope and hence each thread could tackle one of these segments. However, this solution increases the sequential execution and costs more calculations in each thread. Many researches have been proposed to parallelize the Bresenham algorithm[4], but their solution lacks for physical hardware or their implementation deals with small data set [6].

Indeed, we exploited the benefits of parallelization in efficient implementation,wherethe problem that can be divided into independent sub-problems is more suitable.Thereby, we prefer the (DDA) algorithm for the implementation on GPU. The arithmetic operations in the DDA are more suited to the GPU because the inner loop, as presented previously, contains independent instructions and need less modification than Bresenham to be parallelized and properly hardware-implemented and scalable.

The parallel 3D DDA algorithm is designed by forming a Multiply-Add operation (FMUL) in its inner loop as follows: the start-end voxel is used in all processors instead of calculating the previous voxel at each step, since in parallel implementation recalculating the value is easier and less expensive than rereading it from global memory. The pseudo code of the implementedalgorithm is listedin Fig.5.

```
Procedure of parallel 3D Line-voxelization
    BEGIN
     blockId = blockIdx.x + blockIdx.y *
               gridDim.x;
     i = blockId * (blockDim.x *
         blockDim.y) + (threadIdx.y *
         blockDim.x) + threadIdx.x;

      Length = abs(x2 – x1)
      if (abs(y2 – y1) > Length) Length =
      abs(y2 – y1)
      incx=(x2 – x1) / Length
      incy=(y2 – y1) / Length
      incz=(z2 – z1) / Length
      For each thread (i)
      Begin
        x = x0 + incx*i
        y = y0 + incy*i
        z = z0 + incz*i
      store_voxel_float4(x, y ,z, 1.0f)
      End
    END
```

Fig.5 The pseudo code for parallel 3D-line voxelization

The indexing configuration of grid and block entries is architected using the built-in variables *threadIdx* and *blockDim*in the CUDA runtime, where the coordinate of each thread can be accessed within kernel function by the variable*i*.

3.2 GPU-based 3D triangle voxelization

The scan-line and edge-function algorithms are optimized based on the parallel implementation; both are executed for triangle primitive. In the sequential procedure of those algorithms we found that there is no dependency in its instructions and thereby their procedure can be parallelized with some modifications. The performance of the filling process largely depends on the executed algorithm and its optimization level. Different configurations are tested in this paper to parallelize these algorithms; each of them is discussed as follow:

3.2.1 Parallel scan-line algorithm

The parallelization of this algorithm is more complex than the edge-function since it has two nested for-loop. So, it can't be parallelized for each pixel, instead, we implemented the parallelism for each scan-line. Each thread is responsible for drawing single span but in this case a sequential loop exists for scanning the horizontal line, and here the largest span will depend on the orientation of a triangle. This can be inefficient for large triangle size where the span will be long and which will increase the consumed time to rasterize it. Actually, in real implementation and for high resolution, all the triangles of an object are of being with small sizes. Locality coherence states that (the triangle size decreases as the number of triangles in a scene increases). Therefore, this algorithm is still being in use especially in low end consumer devices like handhelds.

The following pseudo-code in Fig.6 illustrates the parallel triangle voxelization,

```
Procedure parallelvoxelizationscan-line
algorithm
BEGIN
    Enter the clockwise vertices of the
    Triangle
    blockId = blockIdx.x + blockIdx.y *
          gridDim.x;
    i = blockId * (blockDim.x *
        blockDim.y) + (threadIdx.y *
        blockDim.x) + threadIdx.x;

    For each thread (i)<dy1
    Begin
       //compute the intersections of the
        current scan line (spani.x,spani.z)
        with the left and right sides of
        the triangle;

        span_min=span_max=v;

        span_min.x=v.x+ 1\dx1*i
        span_min.z=v.z + 1\dz1*i
        span_max.x=v.x + 1\dx2*i
        span_max.z=v.z + 1\dz2*i

        zinc = (span_max.z – span_min.z)/(
        span max.x – span min.x);

        for( ; span_min.x<=span_max.x ;
        span_min.x+=1)
     store_voxel_float4(span_min.x,v.y + i
          , span min.z +  zinc*i, 1.0f)
    End
 END
```

Fig.6 The Pseudo code for parallel 3D-triangle voxelization using scan-line approach

The important issue that should be mentioned here is that there are two configurations after sorting the sides of the triangle, as previously shown in Fig.2, one of them if the middle vertex is on the right and the other when it is on the left. A simple condition is added to the algorithm to distinguish these two configurations where the *dx* values of the active sides are checked to begin the span with the smallest *dx* and end it with the largest one. The procedure below takes one case when *dx1* is smaller than *dx2*. One part is introduced below if one of the triangle sides is a horizontal line or parallel to the scan line.

3.2.2 Parallel edge-function algorithm

The edge function implementation is accomplished by distributing the voxels among the available threads. It is more suitable for parallel implementation than the previous approach, since it can be applied on each voxel as SIMD, but in such a case more threads are needed, than the previous algorithm, due to the bounded box, where only the voxels inside the triangle are drawn and those voxels outside the triangle and inside the box aren't drawn although they seize threads and consume an extra processing time.

The following pseudo code in Fig.7 explaines the implementation of this algorithm,where the threads configuration is arranged as *(x,y,z)* so, each thread coordinate should be compared in the three edge functions, that are extended here as a 3D space, to return the decision of display the voxel or not.

```
Procedure parallel voxelizationedge-
function algorithm
    Enter the clockwise vertices of the
    Triangle (A,B,C)
BEGIN
x = blockIdx.x*blockDim.x +threadIdx.x;
 y = blockIdx.y*blockDim.y + threadIdx.y;
  z = blockIdx.z*blockDim.z + threadIdx.z;
  map the threads coordinates to the
  triangle coordinates
  for each thread
  Begin
     e1=(A.x-B.x)*(y-A.y)-(A.y-B.y)*(x-
        A.x)-(A.z-B.z)*(z-A.z);
     e2=(B.x-C.x)*(y-B.y)-(B.y-C.y)*(x-
        B.x)-(B.z-C.z)*(z-B.z);
     e3=(C.x-A.x)*(y-C.y)-(C.y-A.y)*(x-
        C.x)-(C.z-A.z)*(z-C.z);
     if (e1>=0 && e2>=0 && e3>=0)
          store_voxel_float4(x, y ,z,
          1.0f)
  End
END
```

Fig.7 The pseudo code for parallel 3D-triangle voxelization using scan-line approach

In general, a voxel could be inside the triangle if and only if the results of all the three functions are greater or equal than zero, otherwise, the voxel lies outside the triangle.

The advantages of this algorithm include simple logic and high precision, yet it has relatively low efficiency since it needs more threads.

4. RESULTS AND DISCUSSIONS

To evaluate the performance of parallel voxelization, the implementation isrealized on GPU using NVIDIA CUDA architecture, this architecture is chosen due to hardware availability and experience with this technology.

The parallel testsare implemented on personal computer having Intel® CoreTM i7 processor with one NVIDIA GeForce 1050 GTX of compute capability 6.1.This processor contains five SMs (Streaming Multiprocessors), with each of them 128 SPs (Stream Processors) and the maximum memory data rate (112 GB/s).All applications are designed in CUDA version 9.2 using CUDA/C++ languages on visual studio environment.

The obtained results of parallel voxelization algorithms are discussed in this section. They are partitioned into two parts: the first one for line and the other for triangle voxelization, each of which iscomparedwith sequential and other previous worksresults in addition to the OpenGL implementation on GPU itself. These results of parallel implementation are optimized according to the block sizes and the occupancy metrics using the NIVIDIA profile and Nsight tools.

4.1. GPU-based 3D DDA results

In this section, we implement the parallel 3D DDA algorithm as described in section (3.1). The algorithm is applied on different sizes of 3D data from a starting point (0,0,0) to a maximum length as an endpoint where different lengths are used. The resulting line in the 3D space can be moved by the mouse in different orientations.

In the three different implementations sequential, OpenGL and parallel,the measuredtime and FPS metrics are listed in Table.1. These times are measured in microseconds and the block size for CUDA implementation is set to 8x8 in this test. As can be seen from the table, the calculation time depends on the data size. Also the sequential implementation (voxel by another) could not be able to process large data set but the openGL could, while the CUDA program can implement more than four million voxel, it is suitable for

intensively high computing program. The table also shows the speedup of CUDA over OpenGL implementation where a significant factor is gained.

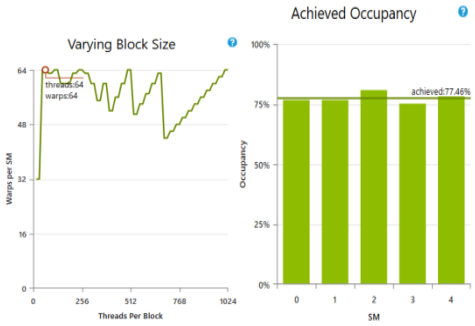
Table 1:Execution times in (µ seconds) and speedup for 3D DDA implementations

Voxels number	Seq.	OpenGL	CUDA	Speedup CUDA\ Seq.	Speedup CUDA\ OpenGL
1 024	4 720	8 978	2.06	847.70X	4352X
4 096	16 971.4	9 920	2.372	1756.14X	4132X
65 536	256 359s	16 987	11.616	3864.55X	1462X
262 144	8 829 333	44 914	42.464	36 133X	1058X
1 048 576	---	158 575	165.21	-----	960X
4 194 304	---	598 399	656.35	---	912X

From the performance analysis in the profile (nvprof), we found that the executable kernel hits the peak theoretical bandwidth, where the achievable bandwidth for this kernel is about 94.6GB/sec that is close to the effective bandwidth of this device (112 GB\sec) and it hits the reasonable bandwidth target which is about 94.616 GB\sec. So there is no more optimization could be done to improve the performance since the kernel reaches to memory bounding.

4.1.1 Effect of varying the block size

In this work, a block size is chosen due to some experience and considering NVIDIA profile (`nvprof`). The benchmarking of this factor is explained here to enhance the overall GPU performance and to investigate how threads are distributed among the available processors. The results in Fig.8are extracted from the profile report when implementing the 3D DDA on data size of 65536 vertices. The left figures show varying theoretical active warp with the block size. It is evident from the figure that the red circle points to the current block size (8x8), if the chart goes higher than this circle point, this means that the selected value is not suitable, and the performance can be improved by increasing the block size. This increasing could subsequently increase the active warp per SM that in turn increases the occupancy.



(a) (b)

Fig.8 Parallel performance for 65536 voxels using 8x8 threads (a) Effect of varying block size on warps per SM (b)The achieved occupancy

As can be seen from Fig.8, the 8x8 threads achieved the highest warp per SM (64 warp or 2048 threads for this type of GPU) so it is a good choice. While the 4x4 block size, holding other parameters constant, made the max warp per SM equal to 32 only, resulting in low occupancy. We will explain the occupancy factor in details next section.

The other block sizes of the 512, 256, 128 threads are also tested (as 1D, 2D or 3D configurations). They approximately gave the same performance and nearly achieved identical occupancy as 64 block size, whereas the 1024 threads operate with less occupancy. Although this larger size reduces memory fetches, but the reason is that the SM cannot process more than two blocks at a time. However, the execution time just slightly increases as benchmarked in Table.2. In general, the highest block size needs too many resources to execute a block, so, it consumes more time. The new thread block may wait to get a resource; thereby the SM may be unable to cover the latency of memory accesses.

Table.2: Effect of different block sizes on the execution time

Data size	Exe. Time (µ sec.)		
	Block size (1024 threads)	Block size (256 threads)	Block size (16 threads)
1024	2.240	1.952	2.208
4096	2.240	2.270	3.168
16384	3.648	3.616	7.040
65536	11.808	11.649	22.112
262144	42.592	42.496	81.792
1048576	165.248	165.184	320.023
4194304	656.351	656.255	1274.91

In the previous table, the execution times for few voxels approximately remained constant till the voxels occupy all the SMs in the GPU. For

example, when the number of voxels is 1024, only one block of the 32x32 threads can reside on one SM. Therefore, the GPU is not a suitable target for small data size compared to other platforms; it needs more data to exploit the power of GPU. While when a (16x16) block size is used, it means that there are enough threads per block to provide hardware with many warps to switch between,

4.1.2 Effect of the occupancy factor

The occupancy metric is also measured for different block and data sizes as shown in Fig.9, where three different block sizes are examined. This factor describes the ratio of executed active warps on the SM to the maximum possible number of active warps that SM can support[17].The 1050 GPU type has five SMs and each one can operate on 2048 threads. So the work of the voxelizing is partitioned by how each SM could be occupied by active blocks[18]. As can be seen from Fig.9, increasing the block size could subsequently increase the active warp per SM that in turn increases the occupancy to about 80%. Whereas the (32,32) threads operate with less occupancy, because the SM cannot process more than two blocks at a time and needing more resources reduced the active warps.In general, the occupancy metric is affected by two factors, the block size and the workload given by the data amount.

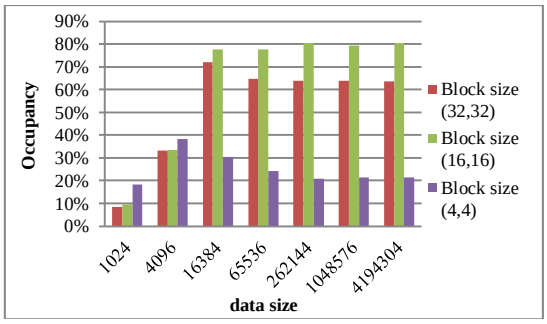


Fig.9 Percentage of achieved occupancy using different block size on various data size

After explaining this optimization, the optimal amount of threads per block can be 64 or 256 threads configurations which gave the higher performance. Using the (16x16) 256 threads, the average warps per each SM are indicated in Fig.10 as 65536/2048/5SMs =409.6 threads. SM1 and SM3 launched fewer warps compared to others. The utilization of the SMs (SM activity) according to this configuration is nearly 100% and the hardware waste is low.

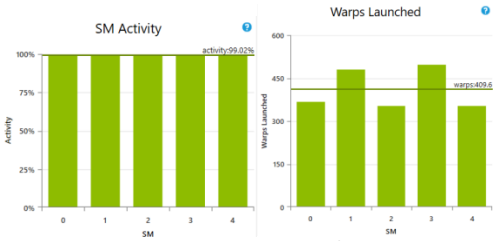


Fig.10 3D DDA kernel statistics of warps launched and SM activity (16x16 block size, 65536 voxels)

4.2. Parallel 3D triangle filling results

In this experiment, the 3D triangle filling is tested on fixed size triangle using both scan-line and edge function method. In scan-line method (start and endpoints approach) each thread could compute single span of different number of voxels, while in the other method, more threads are occupied to fill the same area of triangle thereby it consumes less time. Not all of these threads achieve the three conditions of the edge function that determines which location to fill, so the number of displayed voxels is less than the number of the executed threads. The 3D indexing is used here (GPU 1050 supports this feature) where the thread coordinates *x*, *y* and *z* direction are checked in the edge function equations.

The results are recorded in Table.3, where the execution time and the FPS for these methods are compared. The block size is set to 16x16 threads and the mesh size (number of threads) is set according to the *dy* value of a triangle approximated to multiple of 32. The output triangle has about 16384 voxels. Eventually, the parallel implementation outperforms the OpenGL and the sequential execution, the acceleration factor is about 247x for the scan-line and 4238x for the edge function method with respect to OpenGL execution. This demonstrates the more parallelism that inspired the second method and how it was adapted well as parallel configuration.The maximum throughput is about 4500 M voxel/sec.

On the contrary, the scan-line filling is executed with a sequential portion to draw a single span. Hence, the execution depends on the thread with longest span since the spans are different in their lengths, so the throughput is decreased to about 69M voxel\sec.

Table 3: Performance of triangle filling methods

	Scan line		Edge function	
	Exe time (µ sec.)	FPS	Exe time (µsec.)	FPS
Sequential	303 878.720	3.3	90 654.912	10.1
OpenGL	58 438.23	30.2	16 954.112	59.4
Parallel	236.256	929.2	4.001	1068.2

In 3D space, the orientation of a triangle is an important factor that determines the number of voxelized vertices even if the triangulation process for volumetric representation forms the surface as equilateral triangles. The maximum filling voxels are displayed when the viewing angle is perpendicular on the triangle plane. Therefore the execution time is changed according to this viewing angle.

4.3 Comparison with previous works

Although not many resources that relate to this work, the results of the voxelization are compared to the algorithms that are implemented on other platforms even if it used a 2D version scan conversion. These algorithms are also tested to render only few pixels. Table.4 presents comparison of line drawing with other previous works. These works tested their algorithms on small data size of only one thousand vertices. In our implementation the rasterization of up to 10240 voxels consumes about 2.41 micro seconds making the throughput of about 4 G voxel\sec.

For triangle rasterization, the best implemented algorithm (edge function) is compared with other accelerated algorithms that have been realized on different platforms as FPGAs. The same triangle size is used for comparison although these algorithms are implemented as 2D version. Table.5 shows the execution times in (µ sec) consumed for triangle rasterization. Our GPU voxelization costs the least time compared with other algorithms. It approximately remains constant for all the three compared sizes, since these sizes are considered so small to fill the GPU processors and to utilize its efficiency.

Table 4: Execution times of line drawing compared to other works

	End coordinates	Operating environment	exe.time	throughput voxel\sec
[6] 3D 2018	(992, 992, 992)	ZyboSoC board	0.31µs	3.20 G
[5] 3D 2013	(1000, 1000, 1000)	Spartan 3E FPGA	13.16µs	73.5 M
[5] 3D 2013	(1000, 1000, 1000)	Spartan 3E FPGA	14.71µs	68.0 M
[4] 2D 2011	(1000, 1000)	Personal computer	1.25ms	0.8 M
[19] 2D 2010	(1000, 1000)	Personal computer	1.36ms	0.73 M
[Current] 2019	(10240, 10240, 10240)	GeForce 1050	2.41 µs	4.24 G

Table.5: Triangle filling time (µ sec.) comparison with previous works

Algorithm	v_0(100,30) v_1(10,100) v_2(20,80)	v_0(50,20) v_1(100,100) v_2(10,100)	v_0(10,20) v_1(80,50) v_2(50,100)
[15] 2009 Tiled traversal	18.895	59.995	41.665
[20] 2010 Central traversal	9.204	40.332	29.300
[21] 2011 Midpoint traversal	8.764	40.112	25.745
Current work 3D	2.419	2.461	2.413

5. CONCLUSION

In this paper, the acceleration of voxelization unit has been achieved using parallel techniques, where many voxels have been produced at a time. The acceleration amount of the parallelized 3D DDA was about 4352x max compared to the OpenGL implementation that also uses parallel technique in its execution. In triangle filling, the scan-line algorithm cannot easily parallelize to compute many voxels at a time. It is more difficult than the edge-function approach since it operates per line instead of per voxel parallelization. However, the maximum throughput of parallel 3D DDA implementation was about 4.24 G voxel \sec.In terms of triangle voxelization, 4.5G vertex\sec. was obtained in edge-function parallel implementation and about 69M vertex\sec. was the throughput for the scan-line implementation.

Finally, the parallel implementation has been optimized many times to get these results.We can conclude that more threads in blocks donot lead to higher occupancy; there must be other factors that restrict performance. At the same time, a higher occupancy does not always achieve good performance since it may increase the memory controller connections, therefore the benchmarks is needed to choose the best for a given case and a given hardware type.

REFERENCES

[1] S. Laine and T. Karras, "High-performance software rasterization on GPUs," in *Proceedings of the ACM SIGGRAPH Symposium*, 2011, p. 79.

[2] J. Vince, *Mathematics for Computer Graphics*, Second., vol. 53, no. 9. UK: Springer-Verlag London Limited, 2013.

[3] X. W. Liu and K. Cheng, "Three-dimensional extension of Bresenham's algorithm and its application in straight-line interpolation," *Part B J. Eng. Manuf. SAGE Journals*, vol. 216, no. 3, pp. 459–463, 2002.

[4] L. Yan-Cui, H. Jin-Yan, L. Shi-Yong, and Y. Xia, "A straight line generation algorithm based on line pixels," *Proc. - 2011 IEEE Int.*

Conf. Comput. Sci. Autom. Eng. CSAE 2011, vol. 4, pp. 466–469, 2011.

[5] B. Mohammed and K. Younis, "Hardware Implementation of 3D-Bresenham ' s Algorithm Using FPGA," *Tikrit J. Eng. Sci.*, vol. 20, no. 2, pp. 37–47, 2013.

[6] S. Ismae, O. Tareq, and Y. T. Qassim, "Hardware / software co-design for a parallel three-dimensional bresenham ' s algorithm," vol. 9, no. 1, pp. 148–156, 2019.

[7] M. Khairullah, "An Analysis of Scan Converting a Line with Multi Symmetry," *Int. J. Comput. Appl.*, vol. 61, no. 15, pp. 30–33, 2013.

[8] F. H. Ali, "Depth Buffer DDA Based on FPGA," *Al-Rafidain Eng.*, vol. 19, no. 5, pp. 10–12, 2011.

[9] M. Mcguire, "Efficient GPU Screen-Space Ray Tracing," *J. Comput. Graph. Tech.*, vol. 3, no. 4, pp. 73–85, 2014.

[10] Y. Yan, J. Zhou, and C. Zheng, "Design and implementation of 3D scan conversion algorithm based on Handel-C," *Proc. - 2010 Int. Conf. Anti-Counterfeiting, Secur. Identification, 2010 ASID*, pp. 146–149, 2010.

[11] P. Mileff, K. Nehéz, and J. Dudra, "Accelerated Half-Space Triangle Rasterization," *Acta Polytech. Hungarica*, vol. 12, no. 7, pp. 217–236, 2015.

[12] Fakhraldeen H. Ali Amar I. Dawod, "Fpga Design And Implementation Of A Scan Conversion Graphical Sub-System," *J. Al-Rafidain Eng.*, vol. 16, no. 4, 2007.

[13] X. Wang, F. Guo, and M. Zhu, "A More Efficient Triangle Rasterization Algorithm Implemented in FPGA," in *Proc. ICALIP IEEE*, 2012, pp. 1108–1113.

[14] E. Lengyel, *Mathematics for 3D Game Programming and Computer Graphics*, Third Edit. Course Technology PTR, 2012.

[15] R. Rd, "Universal Rasterizer with Edge Equations and Tile-Scan Triangle Traversal Algorithm for Graphics Processing Units," in *ICME 2009 IEEE*, 2009, pp. 1358–1361.

[16] NVIDIA, "NVIDIA CUDA C Programming Guide Version 3.2," *httpdeveloperdownload nvidiacomcomputecuda 31toolkitdocs NVIDIACUDACProgrammingGuide 31pdf*, pp. 1–170, 2010.

[17] NVIDIA, "NVIDIA Nsight Visual Studio Edition 2019.3 User Guide," 2019. [Online]. Available: https://docs.nvidia.com/nsight-visual-studio-edition/Nsight_Visual_Studio_Edition_User_Guide.htm.

[18] NVIDIA, "GeForce GTX 1050 Graphics Cards | NVIDIA GeForce." 2018.

[19] S. Zhong, Niu Lianqiang, "A Fast Line Rasterization Algorithm Based on Pattern Decomposition," *J. Comput. Des. Comput. Graph.*, vol. 53, no. 9, pp. 1689–1699, 2013.

[20] Y. Ma, X. Wang, M. Zhu, and W. Wan, "Rasterization of geometric primitive in graphics based on FPGA," *ICALIP 2010 - 2010 Int. Conf. Audio, Lang. Image Process. Proc.*, pp. 1211–1216, 2010.

[21] H. Jiang, X. Wang, M. Zhu, W. Wan, and Y. Ma, "A novel triangle rasterization algorithm based on edge function," in *Proceedings of 2011 Cross Strait Quad-Regional Radio Science and Wireless Technology Conference, CSQRWC 2011*, 2011, vol. 2, pp. 1235–1238.

التنفيذ المتوازي للتنقيط الثلاثي الابعاد باستخدام معمارية كودا

سرى نوفل عبد الرزاق sura.nawfal@uomosul.edu.iq

فخرالدين حامد علي fhali_a@yahoo.com

جامعة الموصل - كلية الهندسة - قسم هندسة الحاسوب

الملخص

تعتبر عملية تجسيم الاشكال من المراحل المهمة في خط نقل الرسومات الثلاثية الابعاد تُنفذ هذه المرحلة قبل عرض الاشكال في خط النقل الخاص بوحدة المعالجة الرسومية (GPU)في هذا البحث ، تم تصميم الخوارزميات الخاصة بتوليد النقاط وذلك بأستخدام التنفيذ المتوازي لتسريع العمل. تم استخدام خوارزمية المحلل التفاضلي الرقمي ثلاثي الأبعاد (DDA). وأُستغلت هذه الخوارزمية في ملء المثلث الذي يعتبر العنصر الاساسي للعمل في الانظمة الصورية, وذلك باستخدام طريقتين اساسيتين: خط المسح و دالة الحافة. تم تصميم الطريقة الاولى بحيث يتم توليد خطوط المسح بصورة متوازية. اما في الطريقة الثانية فتم توزيع العمل بحيث ان كل خيط يولد نقطة صورية واحدة. جميع هذه الخوارزميات صُممت ونُفذت بناءاًعلى معمارية CUDA وبأستخدامالمعالج GPU. اظهرت النتائج التجريبية أن مقدار التسارع لخوارزمية 3D DDA كان حوالي 4352x كحد أقصى مقارنةً بتنفيذ الـOpenGL ، اما خوارزمية دالة الحافة فكانت افضل من خط المسح وذلك بسرعة توليد 4.5 مليار نقطة في الثانية الواحدة.

الكلمات الداله :

وحدة المعالجة الرسومية ،معمارية كودا ، دالة الحافة ، التنفيذ المتوازي،تجسيم الاشكال.