2011

Line and circle clipping based on high performance GPGPU and multithreaded computing

Shailesh Khot

LINE AND CIRCLE CLIPPING BASED ON HIGH PERFORMANCE GPGPU AND MULTITHREADED COMPUTING

by

SHAILESH KHOT

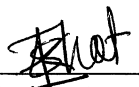
A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in
The Department of Computer Science
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2011

In presenting this thesis in partial fulfillment of the requirements for a master's degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.



(student signature: Shailesh Khot)

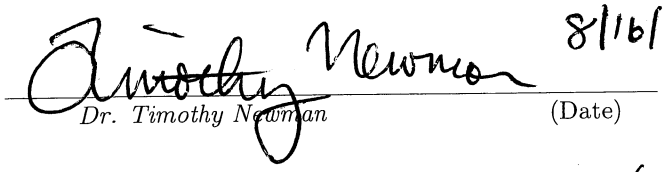
08 | 16 | 2011

(date)

THESIS APPROVAL FORM

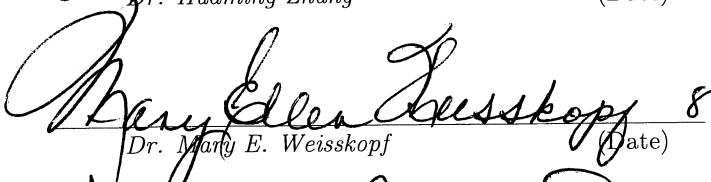
Submitted by Shailesh Khot in the partial fulfillment of the requirements for the degree of Master of Science in Computer Science and accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate of the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements of the degree of Master of Science in Computer Science.



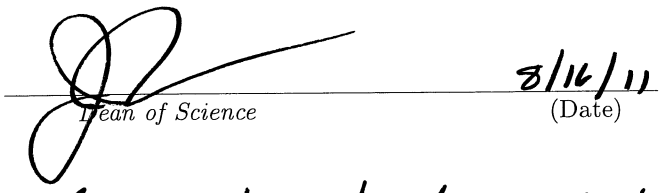
Dr. Timothy Newman (Date) Committee Chair

Dr. Huaming Zhang (Date)



Dr. Mary E. Weisskopf (Date)

Dr. Heggere S. Ranganath (Date) Department Chair



Dean of Science (Date) College Dean

Dr. Rhonda Gaede (Date) Graduate Dean

ABSTRACT

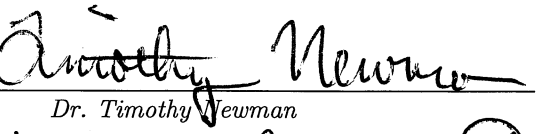
School of Graduate Studies
The University of Alabama in Huntsville

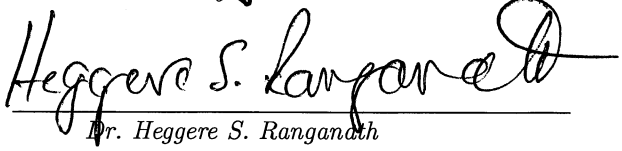
Degree Master of Science College/Dept. Science/ Computer Science

Name of Candidate Shailesh Khot

Title Line and Circle Clipping based on High Performance GPGPU and Multithreaded Computing

New high performance (parallelized) approaches to line and circle clipping are described. The approaches involve breaking down the basic steps of known line and circle clipping algorithms and executing independent steps in parallel. Multiple approaches are described, including ones for GPUs and for multi-processor computers. Performance is also considered. One of the parallel line clipping approaches achieves speedup of 34 times on the GPU and 11 times on the multi-processor computer (over the execution time on a uniprocessor system). One of the parallel techniques for the circle clipping achieves a speedup of 95 times on the GPU and 3 times on the multi-processor computer.

Abstract Approval: Committee Chair 
Dr. Timothy Newman

Department Chair 
Dr. Heggere S. Ranganath

Graduate Dean 
Dr. Rhonda Gaede

ACKNOWLEDGMENTS

I dedicate this thesis to my advisor, Dr. Timothy Newman, for his help, valuable guidance, and constant motivation. I would like to thank him for being a pillar of support and being patient with me throughout the thesis work, without him this thesis would not have been possible.

I want to thank my father for providing financial support towards my education.

I would like to thank my friends Lavelle Houston and Betty Nelson for their constant motivation all the way long especially in my bad times.

Special thanks to Lamardra Juliana and Indjomar Juliana for their help and spiritual support, which I will never forget.

I want to thank my friends, Ayesha Bhatnagar and Swetha Govindaiah for their help and support in my troubled times.

A special thanks to my friend Thomas Adams for his help and being there with me in every situation.

I want to thank my committee members. I also want to thank Dr. Earl Wells for his course in Parallel Programming which helped build a sound foundation for my thesis.

I would like to thank the faculty and staff of the Department of Computer Science at UAHuntsville for their support in all possible ways.

TABLE OF CONTENTS

CONTENT PAGE

LIST OF FIGURES

FIGURE PAGE

LIST OF TABLES

TABLE PAGE

LIST OF SYMBOLS

SYMBOL	DEFINITION
β	probability of non-detecting bad data
δ	Transition Coefficient Constant for the Design of Linear-Phase FIR Filters which are used to take up space when testing the list of symbols
ζ	Reflection Coefficient Parameter
σ	Electrical Conductivity, typically given in units of S/m, which is also a nice way to test the list of symbols

This is a dedication

I drank what?

—Socrates

CHAPTER 1

INTRODUCTION

Research is what I'm doing
when I don't know what I'm doing.

—Wernher von Braun

Clipping is a graphical process involving finding the portions of a scene that are inside a specified region of space [1]. In most applications, the region is a rectangle-like region (2D) or cube-like view volume (3D). However, any shape region could be used for clipping [1]. This region is called a *clipping region*.

In every scene of the computer graphics animation, *scan-conversion* needs to take place. Scan conversion is the process of taking geometric shapes and converting them into an array of pixels stored in the frame buffer to be displayed on the screen. Animation is a simulation of movement created by rapidly displaying a series of pictures, or frames (for example, cartoons on television). Clipping allows only the visible parts of the scene to advance in the viewing pipeline [1] for scan conversion. Clipping accelerates the rendering process by eliminating unneeded scan-conversion. Clipping methods are used in many graphics applications. For example, whenever zooming of a particular scene is necessary, the portion of the scene outside the zoom window needs to be clipped and then the visible area needs to be displayed zoomed-in.

Figure 1.1: Images Made of Primitives [2].

Clipping of such portions of a given scene often involves considering many primitives. Removal of the non-visible parts of the scene results in substantial saving of scan-conversion computational time.

1.1 Clipping of Primitives

Clipping procedures are needed for all basic primitives, such as points, lines, circles and polygons. Many graphics scenes are made up of these basic primitives. For example, Figure 1.1 shows images of scenes made up of triangles. The hand image has about 16,000 triangles, the human has about 78,000 triangles, the cylindrical toy scene has about 11,000 triangles and the scene of a fairy and dragonfly has about 178,000 triangles [2].

This section summarizes some aspects of line, polygon and circle clipping algorithms. Line clipping, which is the process of culling those parts of lines which are outside of the clipping region and allowing drawing of the parts which are inside [1],

is especially important due to large numbers of graphics scenes containing lines. The various line clipping techniques are described in Chapter 2.

Polygon clipping is the process of eliminating those parts of the given polygon that are outside of the clipping region and accepting those parts which are inside [1]. One well known polygon clipping algorithm is the Weiler-Atherton algorithm [1]. The Sutherland-Hodgman algorithm can also be used for polygon clipping, but its output is only vertices from the given polygon that are inside the clipping region [1]. The Weiler-Atherton algorithm allows the clipping of the given polygon over an arbitrarily shaped clipping region [1].

Circle clipping is a procedure which detects and eliminates the portions of the given circle that are outside of the clipping region [1]. Fast circle clipping and Cohen-Sutherland line clipping are considered in this thesis.

1.2 Line and Circle Clipping

Figure 1.2 shows a collection of candidate circles arranged to form a teddy bear and a clipping region. The clipping region bounds are marked as dotted lines, and the candidate circles are shown as solid lines. Figure 1.3 shows the same collection of circles after being clipped. Figure 1.4 shows a collection of candidate lines arranged as a star and a clipping region using the same marking styles for candidates and bounds as in Figure 1.2 and Figure 1.3. Figure 1.5 shows the same collection of lines after being clipped against the clipping region.

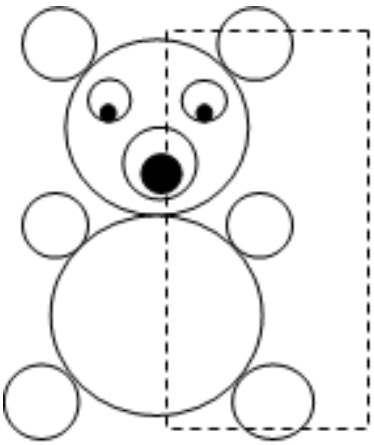


Figure 1.2: Candidate Circles

Figure 1.3: Clipped Circles

1.3 Our Goals

The primary goal of this thesis research was to explore ways to improve the execution time performance of the popular Cohen-Sutherland line clipping and fast circle clipping. The improvement studied was by parallelizing the algorithms for operation on GPUs and multi-processor (especially cluster) computers.

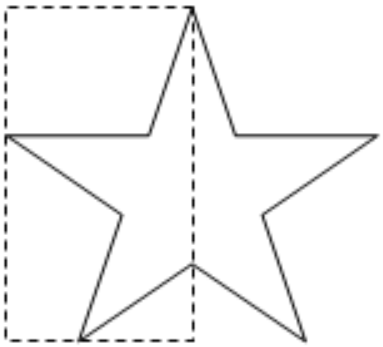


Figure 1.4: Candidate Lines

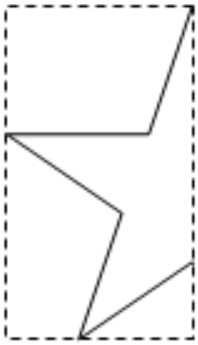


Figure 1.5: Clipped Lines

1.4 Organization

This thesis is organized as follows. This chapter has provided the background. Chapter 2 provides the literature survey and details parallel graphics technologies, especially GPUs and CUDA. Chapter 3 describes the Cohen Sutherland line clipping algorithm and our parallel approaches to it. Chapter 4 describes the fast circle clipping algorithm and our parallel approaches to it. Chapter 5 discusses experimental results. Chapter 6 provides the conclusion and the future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

Over the past ten years there has been a significant amount of research and development into high performance computing and parallel computing technologies. In this chapter, some of the advancements in these areas are discussed. We focus on developments applicable to the graphics and visualization field. Since our focus is clipping, we also describe clipping algorithms which includes line clipping, circle clipping, polygon clipping, etc., that have been described in the literature. There has been extensive research done on these types of clippings.

2.1 Background

First, we provide background details on line and circle clipping. In Section 2.1.1, a background of the various line clipping techniques is provided. Then, in section Section 2.1.2, a background of circle clipping techniques is provided.

2.1.1 Line Clipping

Since the purpose of line clipping is to determine which lines or portions of lines should be scan-converted, and typical scenes can contain a large number of lines,

having a fast line clipping algorithm is important for interactive graphics applications. Some line clipping algorithms are the Cohen-Sutherland, Liang-Barsky, Cyrus-Beck and Nicholl-Lee-Nicholl algorithms [1]. The Cohen-Sutherland algorithm makes its decisions based on where the two endpoints of the line are [1]. The clipping region of Cohen-Sutherland algorithm has to have a rectangle shape. The region also must be axis aligned. Many viewing regions in graphics have such shape. For example, many orthographic view volumes have this shape. The Cohen-Sutherland algorithm is described in detail in a later chapter. The Liang-Barsky algorithm uses the parametric equation of a line and inequalities related to the extents of the clipping region to determine the intersection between the line and the clipping region [1]. The clipping region of Liang-Barsky algorithm has to have rectangle shape. The Cyrus-Beck algorithm is faster than the Cohen-Sutherland line clipping algorithm [1]. The Cyrus-Beck algorithm is a parametric line clipping algorithm similar to the Liang-Barsky [1]. The clipping region of Cyrus-Beck algorithm has to have rectangle shape [1]. The Nicholl-Lee-Nicholl algorithm reduces the chances of clipping a single line segment multiple times, which can happen in other well-known algorithms [1]. The Nicholl-Lee-Nicholl algorithm also is restricted to clipping against rectangular clipping regions.

2.1.2 Circle Clipping

The purpose of circle clipping is to determine the portions of circles that are inside the clipping region. Like in line clipping, the goal is for circle clipping is to be fast so that circles in interactive graphics applications can be rendered quickly. In circle clipping, if the circle lays completely outside the clipping region, it is entirely

clipped. If it is only partially inside and partially outside of the clipping region, then only that part is clipped. Here, we focus on parallelizing one circle clipping algorithm, called the Fast Circle Clipping technique. It is described in detail in Chapter 4.

2.2 Graphical Processing Unit (GPU)

Next, some details about the four generations of GPUs are discussed. The discussion about the fourth generation includes details about the NVidia GeForce 8800 GPU (which is the basis for a new concept in GPUs, the Tesla Architecture) and a programming interface and architecture model for some fourth generation GPUs, the Compute Unified Device Architecture (CUDA) technology. The CUDA programming model and GPU based visualization are also discussed.

2.2.1 GPU Basics

A GPU (Graphics Processing Unit) is a dedicated microprocessor unit specifically designed to perform mathematical and geometric calculations which are necessary for graphics rendering [1]. In comparison to a CPU, a GPU has more ALUs whereas a CPU has more control and Input-Output circuitry than a GPU. A floor plan comparison of CPUs and GPUs is shown in Figure 2.2. The ALU, Input-Output Control and Memory Unit is denoted with green, yellow and orange color, respectively. Hence, a GPU is more devoted to data processing than to data caching and flow control [1]. The Arithmetic Logical Unit (ALU) is a digital circuit which performs arithmetic and logical operations. The typical arithmetic operations it performs include integer addition, subtraction, multiplication and division. It also performs

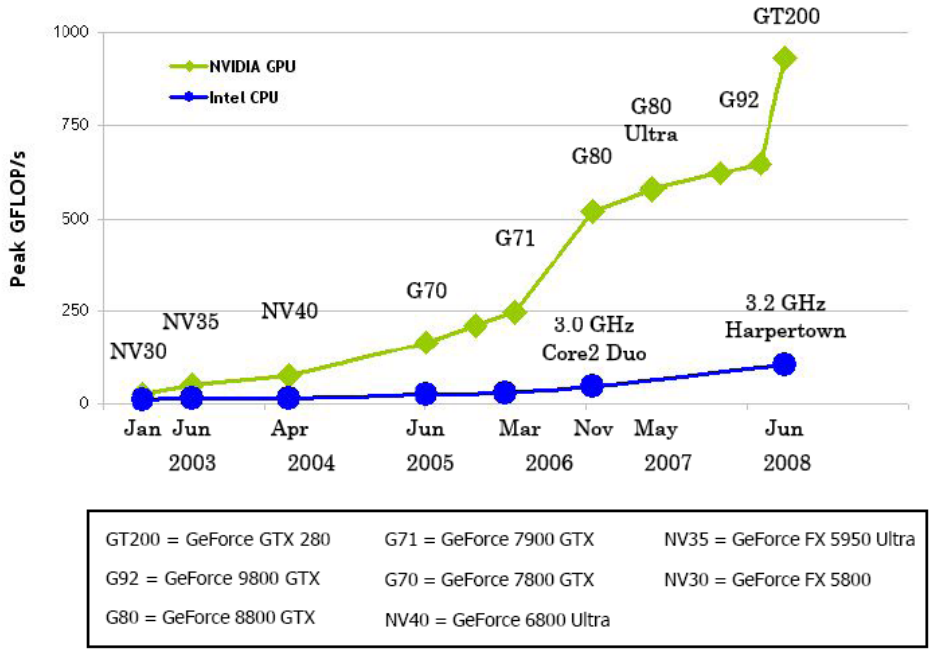


Figure 2.1: GPU vs CPU Computational Performance Comparison [3].

logical operations such as AND, OR and NOT. Figure 2.1 compares the relative peak computational performance of GPU versus CPU in last decade.

Most GPUs have performed their processing using a pipeline that has stages corresponding to major steps in rendering [1]. This rendering pipeline takes a 3D model and generates an image on the 2D screen [4]. This pipeline may include special circuits for certain graphics tasks. For example, since typical primitives in a 3D graphics model are lines and triangles [4], there are often special circuits in the GPU for line and triangle scan conversion [4].

Although modern GPUs have a larger number of transistors than the average modern CPU [4], as well as more ALU capabilities, to use these GPU capabilities, special programming has to be utilized. One example programming environment is

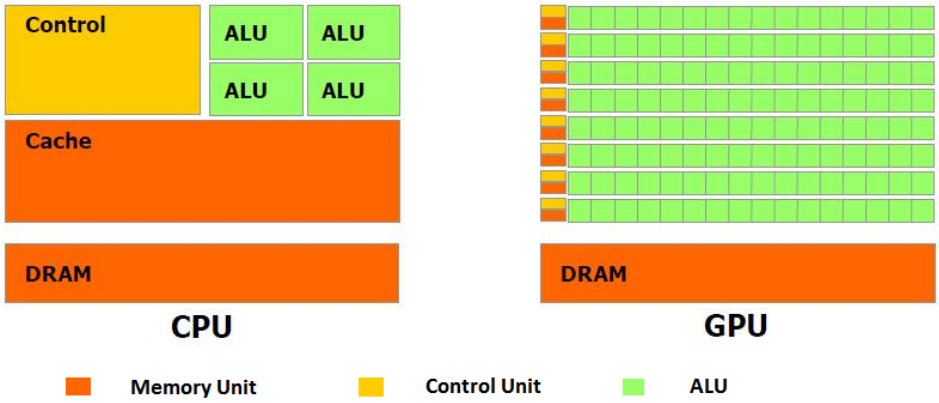


Figure 2.2: GPU and CPU Floor Plan Comparison [3].

the one the current NVidia GPUs use, Compute Unified Device Architecture (CUDA) [4]. CUDA is discussed in more detail later in this chapter.

2.2.1.1 First-Generation GPUs

The first generation of GPUs includes GPUs which were developed before 1998, such as NVIDIA′s TNT2, ATI′s Rage, and 3dFX′s Voodoo3 [5]. These GPUs have some special circuits that are capable of scan-converting triangles [5]. These GPUs completely relieve the CPU from updating individual pixels when running most 2D and 3D graphics applications [5]. First generation GPUs have two limitations: First, they cannot transform 3D objects, Second, they have a limited set of math operations [5].

2.2.1.2 Second-Generation GPUs

The second generation period is from 1999 to 2000 [5]. Second generation GPUs included NVIDIA′s GeForce 256 and GeForce 2, ATI′s Radeon 7500 and S3′s

Savage3D [5]. Both OpenGL and DirectX were supported by the second generation GPU hardware [5]. Also there was expanded support for signed math operations in second generation GPUs [5]. This generation is more configurable than the first generation but not really programmable [5].

2.2.1.3 Third Generation GPUs

The third generation period is the year 2001 [5]. Third generation GPUs include the GeForce and GeForce4 Ti, Microsoft′s Xbox and ATI′s Radeon 8500 [5]. Third generation GPUs have some programmable capabilities [5].

2.2.1.4 Fourth Generation (Modern) GPGPUs

The fourth generation period is the year 2002 and onwards [5]. The fourth generation GPUs include NVIDIA′s GeForce 8800, the GeForce FX family and ATI′s Radeon 9700 [5]. These GPUs have a high performance many-core processor [5]. This feature enables high performance computation in terms of the number of operations that can be done per second. Fourth generation GPUs also have a higher throughput (in terms of number of tasks completed or generated per unit of time) compared to uniprocessor desktop processor [5]. The fourth generation GPU has special units such as rendering pipelines and streaming processors which are aimed at graphics processing but which are general enough that they allow use for non-graphics processing [5].

Some fourth generation GPUs have a *unified* graphics and computing architecture [6]. The first such GPU was the NVIDIA GeForce 8800 GPU [6]. It was

introduced in 2006. It has drivers based on DirectX and OpenGL [4] for graphics applications. It is also programmable using the CUDA library, which is described later. Unified architectures allow a series of operations, called kernel functions, to be applied to each element in the processing *stream* [4]. In this stream processing, one kernel function is applied to all the elements in the stream [4]. For graphics applications, the unified architecture includes a *vertex shader* [6]. A vertex shader runs on each vertex of an object [6]. The 3D position of each vertex in virtual space is transformed to the 2D coordinate at which it appears on the screen by the vertex shader [6]. A vertex shader also adds traits such as position and color to object vertices [6]. Vertex shader cannot create new vertices, though [6]. Current unified architectures also include a *geometry shader* that can add and remove vertices from object primitives [6]. The geometry shader receives the primitives from the vertex shader [6]. The output of the geometry shader is fed to a *rasterizer* [6]. Pixel shader (also called a *fragment shader*) then receives output from the rasterizer [6]. It computes the color of each pixel [6]. It also used for scene lightning [6]. Pixel shader can allow there to be more than one color associated with a given screen pixel [6].

The unified processor cores of GeForce 8800 GPU allows executing vertex, geometry and pixel shader threads for graphics programs and computing threads for CUDA C programs [4]. The 8800 hardware is enabled to execute up to 12,288 threads in 128 cores concurrently and efficiently [4]. (A dispatchable unit of work with its own data and shared (or separate) address space is called a *thread* [7].) In scalar processing, every instruction does only a unary or binary operation [4]. On the other hand, in vector processing each instruction operates simultaneously on multiple

data items (more than two) [4]. Unified core GPUs, like the 8800, unify vertex and pixel/fragment processing and enable high performance [4].

Recently, a large number of general purpose computing tasks have been performed using fourth generation GPUs. Therefore, fourth generation GPUs have increased the capability for general purpose computing on GPU (called GPGPU processing) [4]. In GPGPU processing, general purpose application operations have typically been mapped on to graphics operations [4], although some current GPUs have streaming processors that directly support many common computation processings, such as basic arithmetic operations. These streaming processors allow a form of parallel processing [4] that involves executing a single program component on many data elements simultaneously [6]. This is a data-parallel mode of computing.

The first GPU developed for GPGPU style computing by NVidia is the Tesla architecture [6], introduced in 2007. It is a Fourth Generation GPU. Tesla is intended exclusively for high performance general computing and lacks the ability to render images to a display [6]. This separates Tesla from other GPU architectures where rendering images on the display is the primary intention [6].

The Teslas can have multiple GPUs. Some models are the Tesla C870, D870 and S870 GPUs. They have one, two and four GPUs, respectively [6]. The Teslas GPUs are based on the GeForce 8800 GPU [6]. Teslas are programmed using the same programming model as the 8800s [6]. More details about the Compute Unified Device Architecture is provided in Section 2.2.2.

Figure 2.3 shows the typical architecture of a first generation GPGPU, the NVIDIA GeForce 8800 GPU is shown [6]. In this architecture, the Streaming Pro-

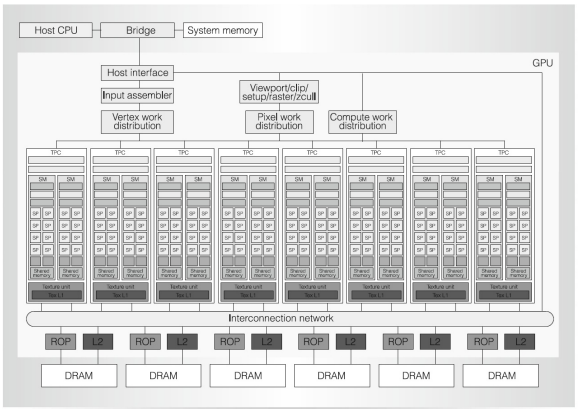


Figure 2.3: NVidia GeForce 8800 Architecture (Typical Tesla Architecture) [6].

cessor (SP) is made up of several Arithmetical and Logic Units (ALUs) and Floating Point Units (FPUs) [6]. The SP performs floating-point operations including add and multiply as well as many integer operations [6]. The Streaming Multiprocessor (SM) contains several SPs and Special Function Units (SFUs) [6]. The SFU can perform reciprocal, reciprocal of square root, $\log_2 x$, 2^x and sin/cos functions [6]. The Texture/Processing Cluster (TPC) has several SMs, a texture unit and some logic control [6]. The texture unit operates in one of four operation modes, namely, vertex shader, geometry shader, pixel shader or compute mode [6]. The NVidia GeForce 8800 also has drivers to support the Microsoft Direct3D DirectX10 graphics API [6]. Tesla computing applications are developed with the same CUDA other NVidia fourth generation GPUs use [6].

The second generation of GPGPUs includes the NVidia Tesla C1060 and S1070, introduced in 2008. These Teslas use the T10 GPU, which is based on the GeForce GTX 280 GPU [6]. The T10 has 240 processor cores [6]. It achieves at peak one teraFLOPs performance for single precision floating-point operations [6]. It

supports the IEEE 754 single precision floating-point and the IEEE 754-2008 double precision (64-bit) floating-point arithmetic formats [6]. It has 4 GB DRAM memory [6].

In March 2010, NVIDIA introduced the third generation of GPGPUs, called the Fermi GPU computing architecture [8]. This generation includes the Fermi GF100 GPU. This GF100 Fermi has 512 CUDA cores and 3 billion transistors [8]. It has 6 GB DRAM memory. That memory is Error Correcting Code (ECC) memory to correct single-bit errors [8]. It can also detect (but not correct) double bit errors [8]. The Fermi GPUs have a two level cache [8] and feature a significant increase in double-precision performance. Its drivers support the programming languages C, C++, and FORTRAN, plus libraries /environments of OpenCL and DirectCompute [8]. Up to 16 double-precision floating point operations can be performed per streaming multiprocessor in each clock cycle, which is an improvement over the Tesla architecture [8].

2.2.2 Compute Unified Device Architecture (CUDA) Technology

CUDA was developed by NVidia and introduced in November 2006 (in the GeForce 8800) [3]. In CUDA, the GPU is treated as a generic data parallel computing device [3]. Its programming environment, CUDA with C, allows developers to use C for programming the GPU [3]. Other languages and APIs supported by CUDA-enabled-GPUs are shown in Figure 2.4 [3].

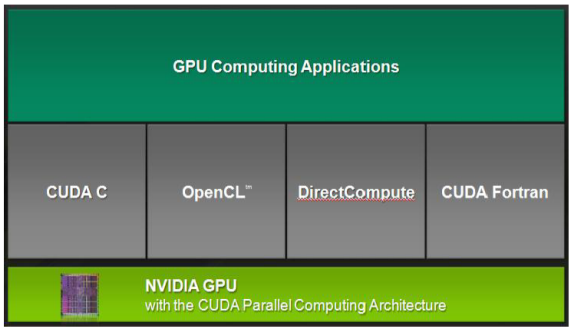


Figure 2.4: NVIDIA CUDA Application Programming Interface(API) [3].

2.2.2.1 CUDA Programming Model

A CUDA program has a kernel (the portion of the program source code executed in parallel) which is called by the host program [3]. This kernel is executed in parallel by each thread of a set of threads [3]. The CUDA thread model is depicted in Figure 2.5. Only one kernel can execute at a time in CUDA [3]. A set of concurrently executing threads is called a *thread block* [3]. In thread blocks, threads can cooperate among themselves through barrier synchronization and shared memory [3]. A group of threads that is executing the same kernel is called a *grid* [3]. One kernel is executed by one grid of blocks [3]. Grids are organized by compiler or the programmer [3]. Every thread block has a unique block ID within its grid [3]. The kernel program is instantiated by the GPU on the threads of the grid(s) [3]. Every one of these threads has an input, a thread ID within its thread block, a program counter, set of registers, per-thread private memory, and output results [3].

Grids read inputs from global memory, write results back to global memory, and synchronize between dependent kernels [3]. In the CUDA programming model,

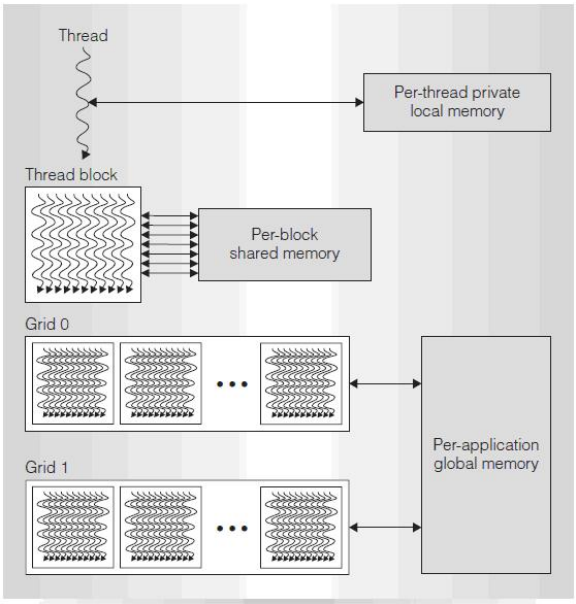


Figure 2.5: CUDA Thread Model [3].

every thread has private memory, which is used for function calls and C automatic variables [3]. Every CUDA thread block also has shared memory used for inter-thread communication and sharing of results among parallel threads [3]. Grids of thread blocks have a global memory space that is used for sharing results after kernel-wide global synchronization [3]. The CUDA memory model is illustrated in Figure 2.6.

CUDA program execution on GPU hardware takes place as follows. The hierarchy of threads in the program is mapped to a hierarchy of processing elements on the GPU [3]. The GPU executes the threads of each kernel grid [3]. GPUs that support CUDA currently follow a streaming multiprocessor (SM) model [3], which means the GPU executes one or more thread blocks on its processing elements, where each of these is an execution units in the SM model [3]. This is shown in Figure 2.7.

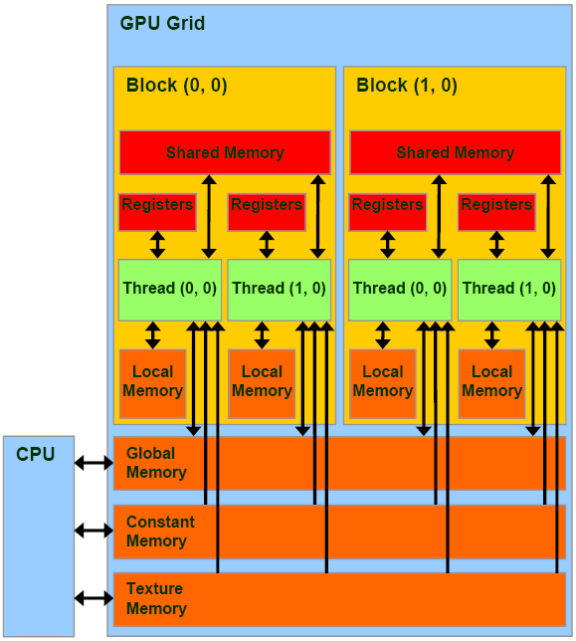


Figure 2.6: CUDA Memory Model [3].

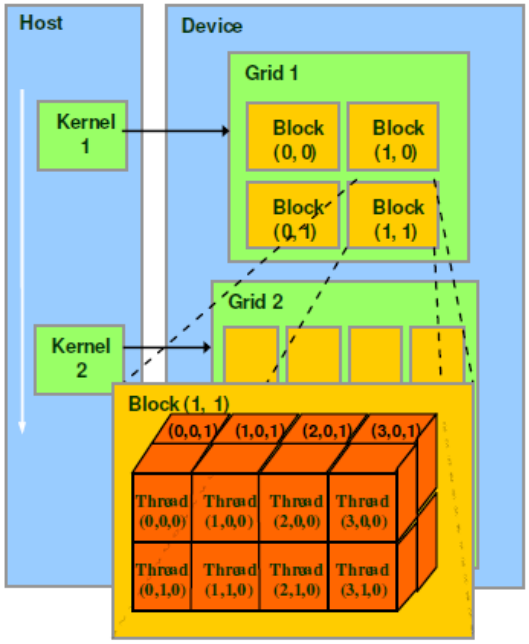


Figure 2.7: CUDA Kernel, Grids and Blocks Model [3].

2.2.3 Visualization (3D) based on GPUs

It has been stated [9] that "Visualization can assist understanding large and complex datasets involved in engineering, research, and business work-flows." The understanding comes from the user interactively exploring a visual representation created by the visualization. But real time visualization of large scale input data, which can be hundreds of megabytes to petabytes in size, is still an open challenge [9]. The usage of modern Graphics Processing Units (GPUs) in visualizing this large scale data could allow higher performance than the CPU [9], and a number of visualizations on GPUs have been reported. GPU execution of a task is often two to three times faster than the CPU [9].

There are several advantages of GPUs over CPUs. First, the GPU has more transistors than the CPU [9] which provides the potential for doing more computations each second [9]. Second, GPUs have wider data paths to memory than CPUs [9], resulting in faster availability of data for processing than on CPU [9].

The GPU creates the visual (image) representation for the display device from the input data using a three stage processing model, namely, filtering, mapping and rendering [9]. For volume visualization application, the filtering stage obtains input volume data that could be from a numerical simulation, measurements of physical data or a database [9]. This stage also may select the part of the data to be visualized [9]. The mapping stage maps filtering stage results to geometric primitives such as points or lines, and determines attributes of the primitives, such as color,

position, or size [9]. Finally, the rendering stage generates a rendering by performing operations such as view transformations, scene illumination and shading [9].

Implementing GPU based visualization methods usually involves programming of the vertex and fragment processing stages of the GPU [9]. The programmer programs the stages to operate on *streams* of which consist of data elements (vertices). The programs are kernels, which were discussed earlier (they are computational programming code) [9]. The vertex and fragment processing stages of the GPU act on the input stream elements (vertices or fragments). These elements are kept in the input registers of the GPU [9]. The results from vertex and fragment processing are written into the output registers of the GPU [9].

We next discuss one visualization task that has been performed on GPUs. The task is volume clipping, which allows cutting away selected parts of a volume dataset. Two volume clipping approaches have been described for GPUs. The first one is an image-space approach where the depth structure of the geometry of an object is kept in 2D texture′s and each texel has a one-to-one correspondence to a pixel [9]. The clipped area is represented as a tessellated boundary surface in the form of triangular meshes [9]. In the approach, the depth structure of the clipped volume is used by the fragment (vertices or triangles) processing stage of the GPU to remove the clipped part of the volume [9]. The second approach is an object-space approach in which a 3D object model is first converted onto 3D volume elements [9]. The clipping regions for the volume are specified by marking corresponding voxels on the volume [9]. The depth based volume clipping gives high quality of images because the volume clipping is performed with per pixel accuracy [9].

2.3 Parallel Environments and Supercomputing

Many scientific and engineering problems involve large quantities of repetitive calculations on large datasets [10]. Often, these calculations need to be done in a finite amount of computation time. This computation performance requirement from some scientific and engineering applications has motivated the development of parallel computing approaches [10]. Some of the applications done using parallel computers include weather prediction, astronomical simulations, etc.

A parallel computer can be a single computer with multiple internal processors or a collection of multiple interconnected computers that form a coherent high-performance computing platform [10]. The two fundamental types of parallel computer are shared memory multiprocessors and distributed memory multiprocessors [10]. Many multiprocessors do parallel computing using Message-Passing [10].

The shared memory model uses coordinated execution on distinct processors in the parallel environment [10]. It has a common memory which is connected to all the processors in a way allowing any processor to access any memory location [10]. The model uses a single address space, in which each memory location has a unique address [10].

There is also a distributed, shared memory model in which each processor is physically at a different distance from the shared memory [10]. In it, processors access physically nearby memory locations faster than distance locations [10]. Distributed, shared memory computers usually have different memory accesses times for the dif-

ferent memory locations. When this is the case, the computer is called Non-Uniform Memory Access (NUMA) [10].

The distributed memory model uses an interconnected group of processors, each with its own memory [10]. Many distributed memory machines have a local memory for each processor that is not accessible by other processors [10]. The interconnection network connects the processors [10] and is used to carry messages for communication between processors [10]. These messages carry data from one processor to another as directed by the program which resides on one of the processor [10]. The distributed memory computers are also called examples of computers using message passing for parallel computation [10].

The Flynns taxonomy is a way to classify the computer architectures. It was proposed by Michael J. Flynn in 1966 [10]. It has four categories that are based on the count of instructions executed at a time and the number of data streams instructions act on at a time [10]. A data stream is the set of scalar operands operated on by the architecture. The four categories of Flynn are defined by two axes, namely instruction axis and data axis. And each axis has two slots for either single or multiple instruction stream and single or multiple data stream.

The first category is Single Instruction stream, Single Data stream (SISD) computing. In SISD, there is one instruction executing at a time operating on simple data input stream and outputting one data item [10]. This category includes uniprocessor computers.

In Multiple Instruction stream, Multiple Data stream (MIMD) computers, there are more than one instruction stream executing at a time, and the streams to-

gether operate on more than one data stream at the same time [10]. MIMD computers generate more than one output data stream at a time [10].

In Single Instruction stream, Multiple Data stream (SIMD) computers, one instruction executes at a time and it operates upon multiple data items and generates more than one output [10]. SIMD instructions are often supported in devices used for multimedia and graphics applications [10].

In Multiple Instruction stream, Single Data stream (MISD) computers, there are more than one instruction executing at a time but all instruction streams operate on the same data stream at a time and output one data stream [10]. No commercial computers of this type have been built to date.

2.3.1 Parallel Environments

Next, three popular high-level parallel programming environments or libraries are discussed. These three environments have special parallel programming constructs and statements for declaring shared variables and parallel code sections.

2.3.1.1 Message Passing Interface (MPI) Computing API

The first of these is the Message-Passing Interface (MPI). It was developed by group of academic and industrial partners as a standard for parallel execution in message-passing systems [10]. It allows processes and processors to communicate with each other by sending and receiving messages through a set of library routines that support message passing and associated operations [10]. It is a standard for parallel programs running on computer clusters and for supercomputers having a distributed

memory system [10]. The MPI provides an advantage of portability across many computers [10]. MPI encourages memory locality (locality of reference) [10]. Locality of reference is the phenomenon where programs tend to reuse data and instructions they have used recently [10].

2.3.1.2 Open Multi-Processing (OpenMP) Computing API

The second environment is the OpenMP API. It was released in 1997. It supports a shared memory multiprocessing model [10]. The programming languages supported by OpenMP are C, C++ and Fortran [10]. The operating systems supported by OpenMP are Unix, Solaris, Mac OS X and Microsoft Windows [10]. OpenMP is said by [10] to be "based on a standard sequential programming language, but it includes additional directives that the compiler uses to produce parallel code." As stated in [10], "the OpenMP directives are a set of parallel programming constructs and environmental variables." OpenMP employs a thread based *fork-join* model of the processes [10]. Fork-join is a method of parallelization where a *master thread* forks a specified number of *slave threads*. The task is divided among them [10], with the slave threads run concurrently to complete the task [10]. The set of slave threads is created by a *parallel directive* [10], which causes a block of code to be executed by multiple threads in parallel on one or more processors" [10].

2.3.1.3 Portable Operating System Interface (POSIX) Computing API

The third environment is standardized threads. Many multi-threaded operating systems have been developed, including SUN Solaris, IBM AIX, MAC OS X,

GNU/Linux, SGI IRIX, and Windows XP [11]. Although these systems have facilities to execute multiple threads at once, every system is different in its design, specification and implementation [11]. A standard multi-threading library has been developed by a joint working group called the Austin Group [11]. This standard is called the Portable Operating System Interface (POSIX) threads. POSIX threads are supported in many operating systems [11]. As stated by [11], "POSIX threads include a command interpreter (or shell) for executing the commands to create, join, destroy and schedule threads [11] and utility programs that support applications portability at the source code level."

2.3.1.4 Programming Model

In the POSIX threads programming model, the main program itself is a thread [10]. The main program can explicitly create and terminate slave threads [10].

In shared memory systems, data is shared among the slave threads for processing [10]. Accessing this shared data needs to be carefully controlled if the data is to be altered, or erroneous output can occur [10]. In multi-threading, a *critical section* is a section of the code that accesses a particular resource (e.g., a data item) that only one thread should access at a time [10]. Whichever thread first reaches its critical section should be the only one allowed to modify the shared resource while all other threads are prevented from entering and modifying [10]. When the thread which entered the critical section first completes its operation on the shared data, then another thread is allowed to the enter critical section [10]. The control on shared resource usage is imposed by use of *mutual exclusion (mutex)* [10].

Pthreads have mutex lock and unlock functions. When a thread finds a mutex locked, it waits for the lock to open [10]. When a thread finds a locked critical section, that thread is blocked by the system [10]. It can proceed only once the thread that did the mutex lock unlocks the mutex [10].

2.3.2 Modern Supercomputers

Modern supercomputers have often been either single CPU machines with sophisticated architectures or a collection of many relatively simple processing elements interconnected together [10]. The problems which are too demanding for other computers are solved using supercomputers. A current supercomputer trend is towards the cluster of computers [10].

As of June 2011, the Japanese K Computer is number one on the Top 500 list of the worlds top supercomputer (http://www.top500.org). It is produced by Fujistu and located at the RIKEN Advanced Institute for Computational Science [12]. It has a peak computation performance level of 8.162 PetaFLOPs with a computing efficiency ratio of 93% [12]. The K computer currently has 68544 SPARCV64 VIIIfx CPUs running at 2 GHz clock speed [12]. Each CPU has eight core, resulting in a total of 548,352 cores. The CPUs are arranged in cabinets [12]. Each cabinet has 96 computing CPUs and 6 CPUs for input and output [12]. There are 16GB memory on each CPU [12]. A special 6-dimensional torus network interconnect called *Tofu* connects all the CPUs [12]. The computer uses a version of Message Passing Interface optimized for its network [12]. The computer uses the Fujistu Exabyte File System, which is based on Lustre (**L**inux Cl**uster**), a parallel distributed file system

used for large scale cluster computing [12]. This system scales to several hundred petabytes [12]. The K system has almost twice the number of cores of any other system in the Top 500 [12]. Its computational performance level is approximately equal to that of one million linked desktops [12]. That performance is larger than the combined levels of the next five computers on the list [12]. The average power consumption of the top 10 systems on the list is 4.3 MW. The K system consumes 9.89 MW of power [12].

2.3.2.1 Alabama Supercomputer Center (ASC) (www.asc.edu)

The supercomputers available for the thesis research were at the Alabama Supercomputer Center which has two supercomputers, the SGI Altix supercomputer and the Dense Memory Cluster (DMC) [13].

2.3.2.2 SGI Altix

There are two classes of SGI Altix servers at the ASC, the Altix 350 and Altix 450 servers [13]. The Altix 350 and 450 are scalable, mid range servers that run a 64-bit Linux operating system [13]. The Altix 450 server at ASC has three compute nodes, namely, Altix 7, Altix 8 and Altix 9. The Altix 7 and Altix 8 have 72 CPU cores and 432 GB of primary memory each [13]. The Altix 9 has 12 CPU cores and 464 GB of primary memory [13]. There is a total of 1340 GB shared memory [13]. The Altix7 and Altix 8 have 1.6 GHz dual core Intel Itanium2 9000 series processors [13]. The file system is a Pansasa storage array of data disks with a 15 terabytes file system for the permanent storage of data [13]. In total, the ASC Altix yields a peak floating

point performance of 1035 GFLOPs [13]. Its processors are connected through an interconnection network which includes NUMA link for sharing memory [13]. It has an Infiniband communication link for file system access and gigabit Ethernet internet connectivity [13]. It also has a secondary Ethernet connection as a redundant fail over for its internet [13].

2.3.2.3 Dense Memory Cluster (DMC)

The ASC's DMC is a commodity cluster which was put together using commodity components from Microway, Penguin, Cisco, Linksys, Cluster Resources, Novell, SGI, Spectrum, Avocent and Pansasa [13].

The DMC machine has 1512 cores in total, arranged in 60 nodes and a total of 8224 GB of distributed shared memory [13]. Each node has 64 GB of primary memory [13]. It has 8.2 TB of permanent storage [13]. It has a Panasas file server system [13].

The DMC also has 8 GPUs, in the form of two NVidia Tesla S1070 units [13]. Each Tesla S1070 has 4 NVidia Tesla 10-Series GPUs [13]. The conventional processing capability of the DMC is 13.25 TFLOPs [13]. The single precision and double precision GPU computational capability is 8.3 TFLOPs and 0.7 TFLOPs, respectively [13]. Software development for GPU has done using NVIDIA CUDA programming suite with standard C programming language extended for thread and memory management [13].

CHAPTER 3

LINE CLIPPING

This chapter focuses on the new parallel approaches for the Cohen-Sutherland line clipping algorithm. There are approaches described for using POSIX threads and for GPUs. First details of the serial Cohen-Sutherland are described.

3.1 Serial Approach of Cohen-Sutherland Line Clipping

The Cohen-Sutherland line clipping described here is for a 3D line clipped against a 3D clipping region shaped as an axis-aligned rectangular box. Cohen-Sutherland line clipping is an encoding based algorithm. Line segment endpoint properties are used for encoding a bit code for each line segment endpoint. We call this bit code an *outcode*. The location of the line endpoint with respect to the clipping region is described by this outcode. Fast bitwise operations are performed on these codes to get the visible portions of the line segment.

The encoding of a segment's endpoints into an outcode is in a 6-bit encoding scheme. Each bit encodes the relationship of the endpoint with one face of the clipping region. If the endpoint lies outside the clipping region face, that face's bit in the outcode is set to 1. Otherwise, that bit is set to 0. This 6-bit outcode representation

Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
F(ar)	N(ear)	T(op)	B(ottom)	R(ight)	L(eft)

Figure 3.1: Bit Representation of an Outcode

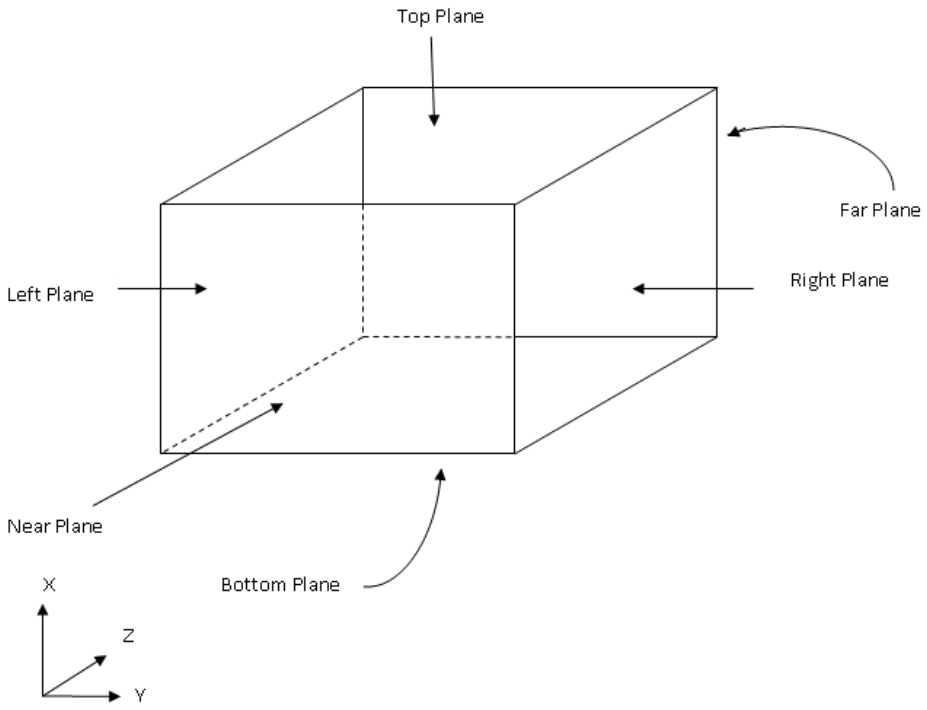


Figure 3.2: Clipping Region Faces

is shown in Figure 3.1. A line segment is *accepted* if the line segment starts inside the clipping region and ends inside the clipping region. If line segment′s both endpoints are are outside of the clipping region for the same plane and does not intersect the clipping region then, the line segment is *rejected*.

The Cohen-Sutherland outcode has the form FNTBRL where each letter indicates a particular face's bit. If a line segment endpoint lies beyond the far face of the clipping region then the F bit of the outcode is set to 1. Similarly, the N,T,B,R

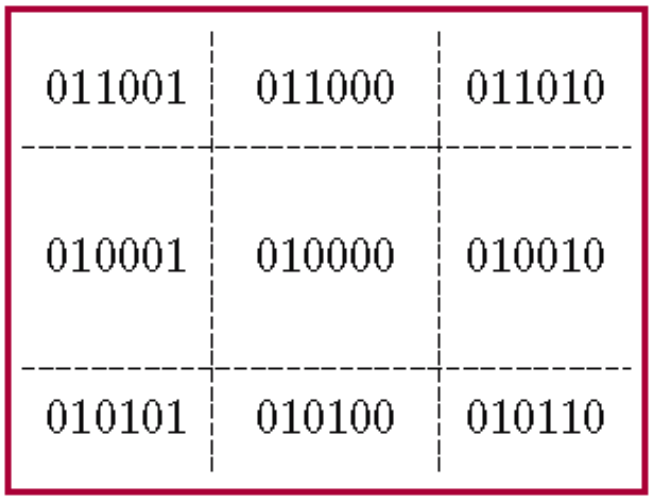


Figure 3.3: Outcodes for Areas in Front of the Plane [1].

and L bits are set to 1 if the line segment's endpoints lies beyond the plane containing clipping region's near, top, bottom, right, and left face, respectively. Figure 3.2 shows the planes. The F(far) plane contains the back face of the clipping region. The N(near) plane contains the front face of the clipping region. The T(top) face contains the upper face of the clipping region. The B(bottom) face contains the down face of the clipping region. The R(right) face contains the right side face of the clipping region. The L(left) face contains the left side face of the clipping region. Figure 3.3 shows outcodes in regions of space in front of the near plane. There are eight regions here. Each one is formed by extending the four boundaries of the near face of the clipping region. In total, there are 27 possible outcodes.

The determination of trivial acceptance or rejection is described next. After computing the outcodes of both the endpoints of line segment, it performs bitwise AND and bitwise OR operations. If the result of the bitwise AND operation is non-

zero (which means common bits are set in both endpoint outcode), then both the endpoints are beyond one face and the line segment is rejected. That is the trivial reject case. If the result of the bitwise OR operation results in zero (which means no bit is set in any of the outcodes of the endpoints of the line segment), then both endpoints are inside the clipping region and the line segment is accepted. That is the trivial accept case.

When the given line segment is neither trivially rejected nor trivially accepted, then it is iteratively clipped with respect to appropriate faces of the clipping region until there is a sub-segment of the given line that is either trivially accepted or trivially rejected. Line segments which are neither trivially accepted nor trivially rejected have at least one non-zero outcode. For these lines, an endpoint with a non-zero outcode values is considered. The end of the line containing that endpoint is clipped versus the face of the clipping region that is associated with the first non-zero bit in the endpoint's outcode. (The algorithm scans the non-zero outcode of the selected endpoint from left to right from most significant bit to least significant bit until a non-zero bit is found.) For example, in the FNTBRL notation, if F is the first bit set to 1 then the line segment is clipped against the far face of the clipping region. After the point of intersection of the line with the plane containing that face is found, outcodes are completed. The iterative process continues until the clipped line segment is either trivially accepted or trivially rejected. The approach is shown graphically in Figure 3.4.

The pseudo-code for outcode computation is shown as,

Procedure Outcode_Compute() in Figure 3.5. The pseudo-code for Cohen-Sutherland

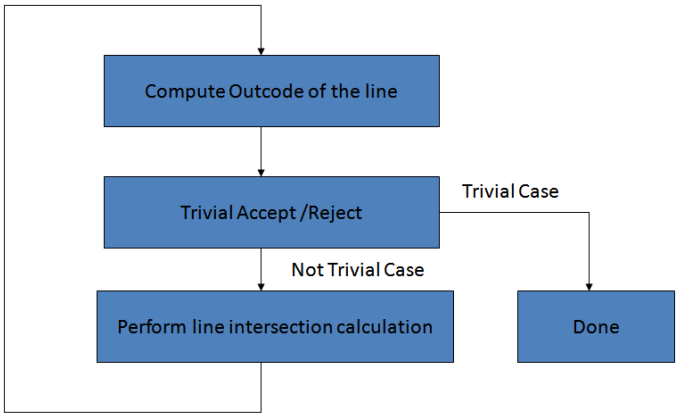


Figure 3.4: Cohen-Sutherland Line Clipping Algorithm Serial Version

Clipping, is shown as Procedure LineClip in Figure 3.6. In the pseudocodes, Point is a data structure which holds x,y and z coordinates of a line segment endpoint and outcode is a data structure which holds endpoint outcodes. We use the same ordering as described earlier, FNTBRL, where the letters represent the far, near, top, bottom, right and left bits, respectively. Here the most significant bit is F and least significant bit is L.

3.2 Parallel Approaches

We have designed two data parallel approaches for Cohen-Sutherland line clipping algorithm. These two approaches are named Approach A and Approach B and are discussed in detail in this section.

3.2.1 Parallel Approach A for Line Clipping

In Approach A, each line (or group of lines) is processed independently (in parallel). In it, each line (or group) is processed completely by one thread. That

```
Procedure Outcode_Compute(Point P)
    Set Outcode E to zero
    If P is beyond the left face
        Set the bit L in E
    Else if P is beyond the right face
        Set the bit R in E
    Else if P is beyond the top face
        Set the bit T in E
    Else if P is beyond the bottom face
        Set the bit B in E
    Else if P is beyond the near face
        Set the bit N in E
    Else if P is beyond the far face
        Set the bit F in E
    End If
        Return E
```

Figure 3.5: Cohen-Sutherland Line Clipping Outcode Computation

```
Procedure Line_Clip(Point p, Point q)
    Outcode E1 ← Outcode_Compute(p)
    Outcode E2 ← Outcode_Compute(q)

  Repeat

    If E1 and E2 both zero (trivial acceptance)
        Return as accept
    Else if E1 and E2 are non-zero (trivial reject)
        Return as reject
    End If

    If E1 is zero
        Set Outcode E to E2
    Else
        Set Outcode E to E1
    End If

    If L bit is set in E
        Assign Point S the intersection point with the left face
    End If

    If R bit is set in E
        Assign Point S the intersection point with the right face
    End If

    If T bit is set in E
        Assign Point S the intersection point with the top face
    End If

    If B bit is set in E
        Assign Point S the intersection point with the bottom face
    End If

    If N bit is set in E
        Assign Point S the intersection point with the near face
    End If

    If F bit is set in E
        Assign Point S the intersection point with the far face
    End If

    If E is equal to E1
        p ← S
        E1 ← Outcode_Compute(p)
    Else
        q ← S
        E1 ← Outcode_Compute(q)
    End If

Until E1 and E2  are non-zero
Return the line  from p to q
```

Figure 3.6: Cohen-Sutherland Line Clipping Routine Pseudocode

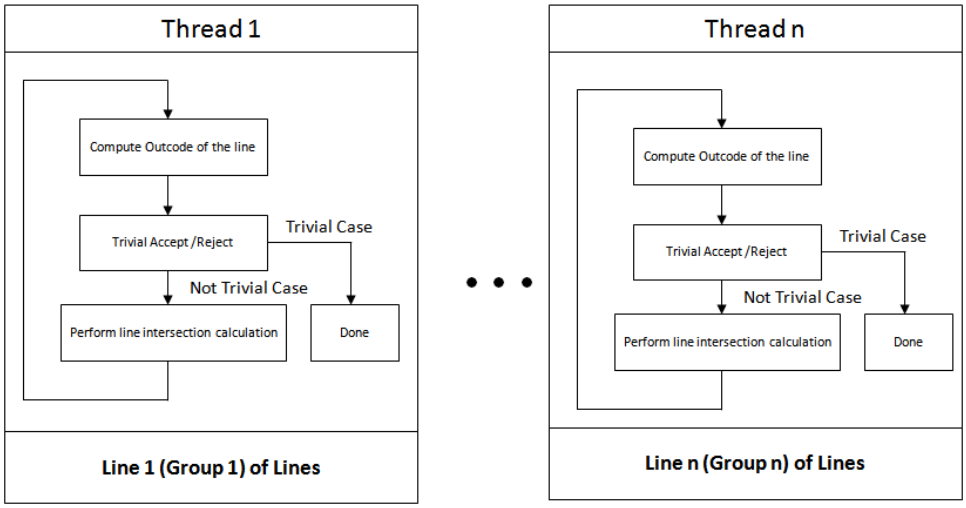


Figure 3.7: Cohen-Sutherland Line Clipping Parallel Approach A

thread does outcode computation, trivial accept, trivial reject, and clipping region intersection calculations. This Approach A is shown graphically in Figure 3.7. The parallel execution time is approximately the time to process whichever line which takes the longest time to finish.

3.2.2 Parallel Approach B for Line Clipping

The second approach, we developed, Approach B, performs the outcode computation independently (in parallel) for each line (or group of lines). First, each thread of a set of k threads performs the outcode computation for one line (or a group of lines). We call this its Step 1. After Step 1 is done, another set of k threads performs the task of trivial reject, trivial accept and, if necessary, the clipping region intersection calculations. Each thread of the set of threads does the computation for one line (or a group of lines). We call this the Step 2 of the Approach B. Approach B is shown graphically in Figure 3.8 and Figure 3.9. The parallel execution time

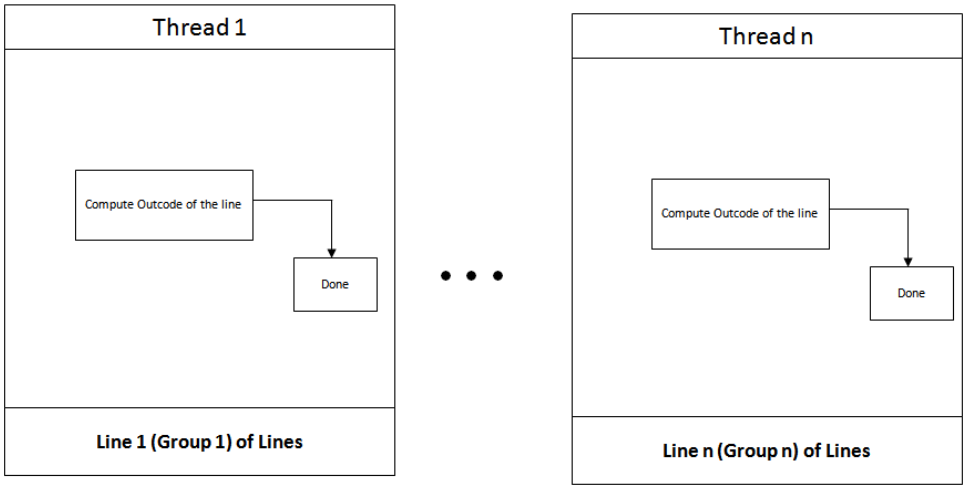


Figure 3.8: Cohen-Sutherland Line Clipping Parallel Approach B, Step 1

is calculated by adding the time for Step 1 to the longest execution of any line for Step 2.

After completion of clipping region intersection calculations, the outcome from the thread was recorded. The outcome of the thread is trivial accept or trivial reject or the clipping region intersection point values. After this the thread is destroyed. Once all the threads are destroyed the task of Cohen-Sutherland line clipping has finished.

3.3 Supercomputer Environment Implementation

The implementation on the supercomputer environment is developed with POSIX Threads. In this environment there is a limit on number of threads created. This limit is presently because the single CPU executes the multiple threads (at least more than 2) in a time shared manner. This time sharing results in increased wait time for a thread to get a free resource or until the resource becomes free. There-

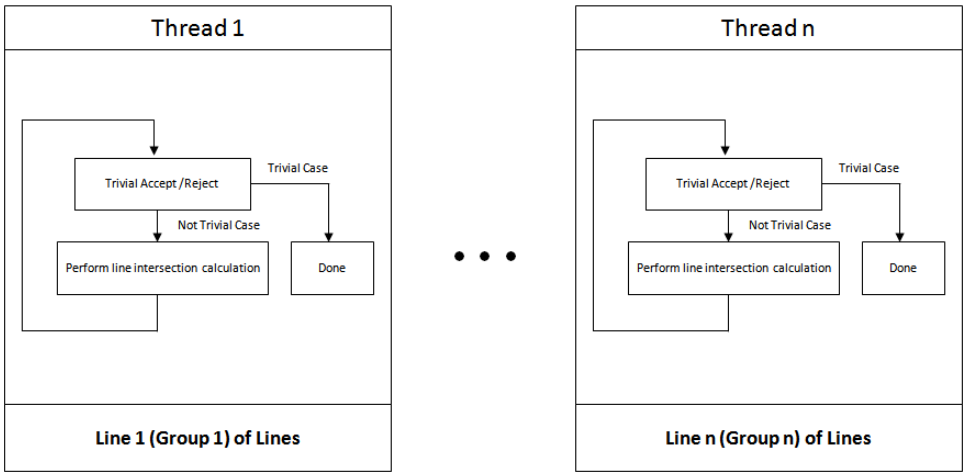


Figure 3.9: Cohen-Sutherland Line Clipping Parallel Approach B, Step 2

fore, a cluster of CPUs (32 or 64 CPUs) executes a large number of threads in a time shared manner. Also in this environment this task is not the only task executing or waiting to execute (there are other tasks from other sources competing for the same resources). In addition to this, creation of large number of threads has an overhead of thread-creation and destruction time. These reasons in all limit the number of threads being created.

For the Approach A, a set of k threads processes the lines. Each thread computes the outcode for one line and does the testing for trivial accept, trivial reject, and, if necessary, the intersection of the line with some clipping boundary. The thread repeats the process if necessary (until line is either trivially accepted or trivially rejected). For Approach B, a set of k threads first does the outcodes computations. Then, another set of k threads computes the Step 2 of trivial accept, trivial reject, and, if necessary, the intersection of the line with a clipping boundary. If intersection is found, the thread computes the outcode again and then repeats the

Step 2 again. That process continues until the line is either trivially accepted or trivially rejected.

3.4 GPU Environment Implementation

The GPU implementation of the parallel Cohen-Sutherland line clipping was developed with NVidia CUDA. In CUDA the maximum number of blocks that can be created are 65535 X 65535. And within every block, 512 threads can be created. Therefore, a total 65535 X 65535 X 512 threads can be created. In Approach A, one CUDA thread was created per line. Therefore, all lines could be clipped simultaneously. Each thread did all the work for one line, including computing the outcodes and testing for the trivial reject, trivial accept or intersection of the line with clipping boundary. For Approach B, there is one CUDA thread per outcode and one thread per line to detect trivial accept, reject and if necessary, computation of line intersection calculations with clipping region.

CHAPTER 4

CIRCLE CLIPPING

This chapter explains the new parallel approaches for the fast circle clipping. The approaches are described for POSIX threads and for GPUs. First, details of serial circle clipping is discussed, though.

4.1 Serial Fast Circle Clipping

This section describes the serial Fast Circle Clipping approach that was parallelized. There are two objects involved in the approach. One object is the circle to be clipped. The other object is a clipping region. The Fast Circle Clipping's clipping region is 2D rectangular shape. It uses parametric equation of a circle. The approach uses steps similar to the Cohen-Sutherland line clipping. In it, a circle is said to be *trivially accepted* if it lies entirely within the clipping region. The circle is *trivially rejected* if it lies entirely outside one boundary of the given clipping region. Part of the circle must be clipped when it lies only partially within the clipping region and partially outside the clipping region. The fast circle clipping algorithm finds and accepts only the part of circle which lies inside the clipping region. Figure 4.1 shows a circle centered at (x_c, y_c) over a clipping region.

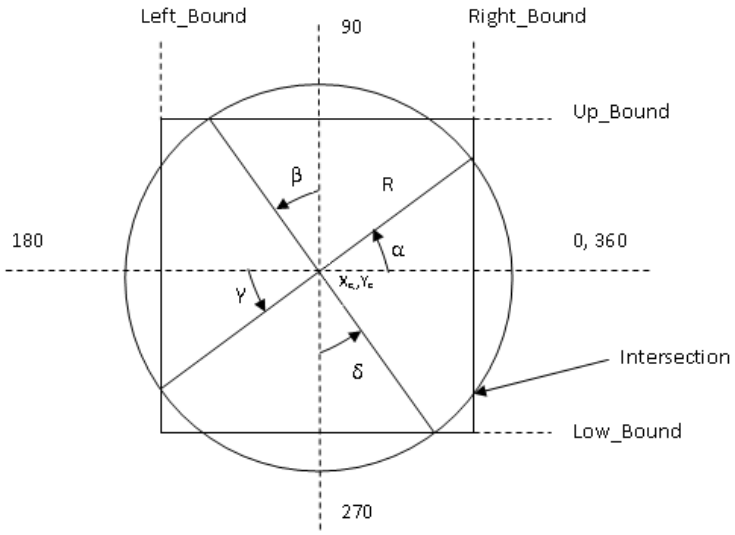


Figure 4.1: Fast Circle Clipping with the Clipping Boundaries and Circle

The approach has a clipping region defined by an upper, lower, left and right bound. These boundaries are labeled in Figure 4.1 as Right_Bound, Up_Bound, Left_Bound and Low_Bound. The angles labeled α, β, γ, δ in Figure 4.1 are half-intersection angles (i.e., the angular measure of the part of the circular arc outside the region). The angle α is computed from Right_Bound x_c and radius, R. The β is computed from Up_Bound y_c and radius, R. The γ is calculated from Left_Bound, x_c and the R. The δ is calculated from Low_Bound, y_c and R. The clipping of a circle against a rectangular boundary results in one to four circular arc segments depending upon where the circle lays over the clipping region boundary. The Fast Circle Clipping defines the arc segments using a parametric representation. The parametric representation for a circle is

$$x = x_c + R\ cos\ (\phi),$$

$$y = y_c + R\ sin\ (\phi),$$

where *(x, y)* is a point on the circle, *(x_c,y_c)* is the center, R is the radius, and ϕ is the angle measured counterclockwise from the vertical (parallel to x) axis to the radius vector at *(x, y)* [8]. In a complete circle ϕ always goes from 0^0 to 360^0. For the clipped circle arcs, the part of the circle that is accepted is defined by the values of ϕ marking the start and the end of the circular arc segments.

Next, the use of trivial accept and reject in the algorithm is explained in more detail. As stated before, the *trivial accept* case for the given circle is that the given circle coordinates lie completely within the clipping region boundary [8]. The *trivial reject* case for the given circle is that the given circle lies completely above, below, left or right of the clipping region [8].

If the circle is neither trivially rejected nor accepted then additional processing is done to find the part (or parts) of the circle that is within the clipping region boundary. The resulting arc (or arcs) is defined by starting and ending angle parameters, center and radius. The starting and ending angles are the locations where the given circle intersects a clipping region boundary. The clipped arcs are then displayed on the screen [8]. For example, if the circle is clipped only against the right boundary of the clipping region, the clipped portion is stored as two arcs, one from 0^0 to α^0 and the other from $(360 - \alpha)^0$ to 360^0, where α is one half the angle subtended by the clipped portion at the center of the circle [8].

The pseudocode for procedure FastCircle_Accept() which tests for the trivial accept of a given circle is given in Figure 4.2. The pseudocode for procedure FastCircle_Reject() which tests for the trivial reject of a given circle is given in Figure 4.3. The pseudocode for procedure FastCircle_Clipping() which handles the clipping of a

```
Procedure  FastCircle_Accept (Center Xc, Yc Radius R)
    If (Xc + R) is not beyond the right boundary
        If (Xc - R) is not beyond the left boundary
            If (Yc + R) is not beyond the lower boundary
                If (Yc - R) is not beyond the upper boundary
                    Trivially accept Circle
                End If
            End If
        End If
    End If
Return as accept
```

Figure 4.2: Pseudocode for Fast Circle Trivial Accept

given circle is shown in Figure 4.4. In these listings, (x_c,y_c) is the center of the circle and R is the radius of the circle. The serial version of the Fast Circle Clip is shown graphically in Figure 4.5.

4.2 Parallel Approaches

We have designed two data parallel approaches for circle clipping. We call these Approach Circle-A and Approach Circle-B. These approaches are discussed in detail next. We also discuss a third parallel approach, Approach Circle-C, that is only usable in a POSIX threads environment.

4.2.1 Approach Circle-A

In Approach Circle-A, each circle (or group of circles) is processed independently (in parallel). In it, each circle (or group) is processed completely by one thread. That thread performs trivial accept, trivial reject and intersection calculations. This

```
Procedure FastCircle_Reject (Center Xc, Yc, Radius R)
    If (Xc + R) is beyond the left boundary
            Trivially reject Circle
    End If

    If (Xc - R) is beyond the right boundary
        Trivially reject Circle
    End If

    If (Yc + R) is beyond the lower boundary
        Trivially reject Circle
    End If

    If (Yc - R) is beyond the upper boundary
        Trivially reject Circle
    End If

Return as reject
```

Figure 4.3: Pseudocode for Fast Circle Trivial Reject

approach is shown graphically in Figure 4.6. The parallel execution time is the time to process whichever circle takes the longest to finish.

4.2.2 Parallel Approach Circle-B

In the Approach Circle-B, each thread of a set of k threads first performs the trivial accept and trivial reject for all the circles in parallel. This step is shown graphically in Figure 4.7. We call it Step 1. Once this step is completed, another set of k threads performs the intersection calculations for the circles. Each thread of the

```
Procedure  FastCircle_Clipping (Center Xc, Center Yc, Radius R)
    If FastCircle_Reject (Center Xc, Center Yc, Radius R) is Reject
        Exit
    End If

    If FastCircle_Accept (Center Xc, Center Yc, Radius R) is A
        Draw Circle (Xc, Yc, R)
    End If

    Initialize n to zero
    Initialize I1, I2, I3, I4 to null

    If (Xc + R) is beyond the right boundary
        Increment n by 1
        Assign I1 the intersection angle with the right boundar
    End If

    If (Xc - R) is beyond the left boundary
        Increment n by 1
        Assign I2 the intersection angle with the left boundary
    End If

    If (Yc + R) is beyond the upper boundary
        Increment n by 1
        Assign I3 the intersection angle with the upper boundary
    End If

    If (Yc - R) is beyond the lower boundary
        Increment n by 1
        Assign I4 the intersection angle with the lower boundary
    End If

    If n is null
        There are no intersections
    Else
        Draw clipped circle of arcs of I1, I2, I3, I4
    End If
```

Figure 4.4: Pseudocode for Circle Clipping

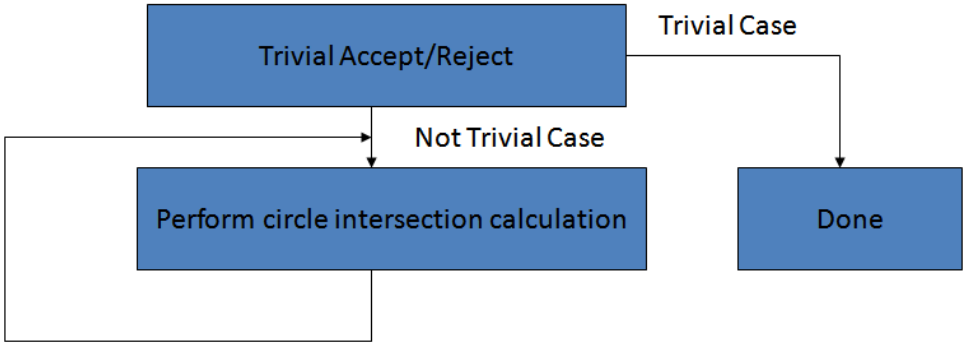


Figure 4.5: Fast Circle Clip Serial Version

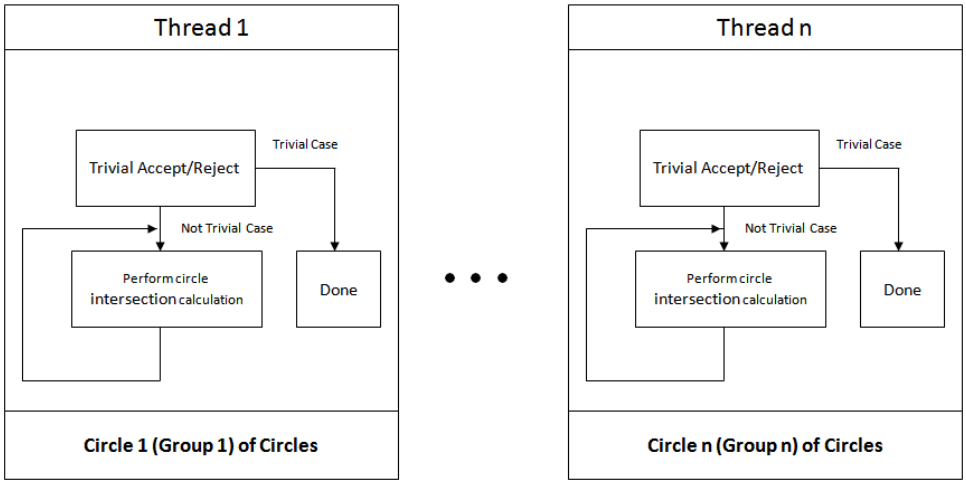


Figure 4.6: Circle Clipping, Parallel Approach Circle-A

set of threads does the computation for one circle (or group of circles). This step is illustrated in Figure 4.8. We call it Step 2. The parallel execution time is calculated by adding the time of processing whichever circle took the longest for Step 1 to the time of processing whichever circle took the longest for Step 2 to finish.

The steps are thus:

Step 1: Do trivial accept or reject for all circles in parallel.

Step 2: Do circle intersection calculations in parallel.

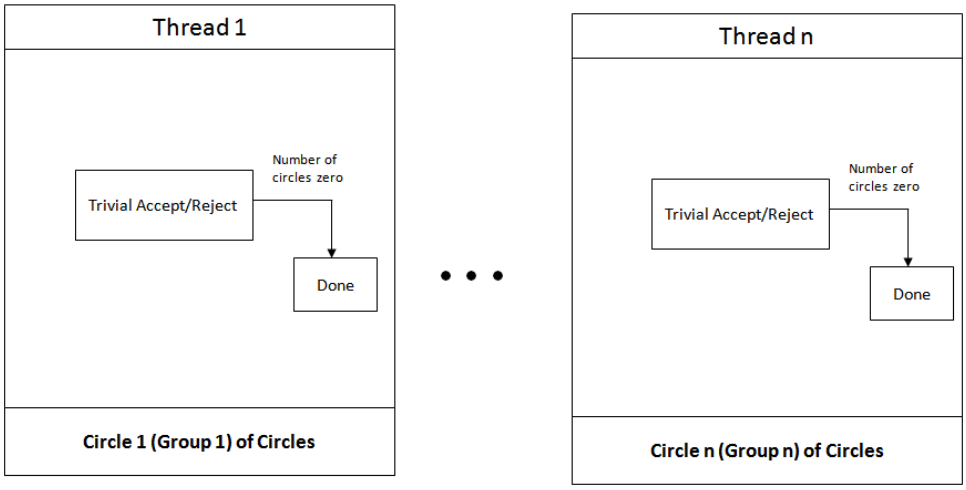


Figure 4.7: Circle Clipping, Parallel Approach Circle-B, Step 1

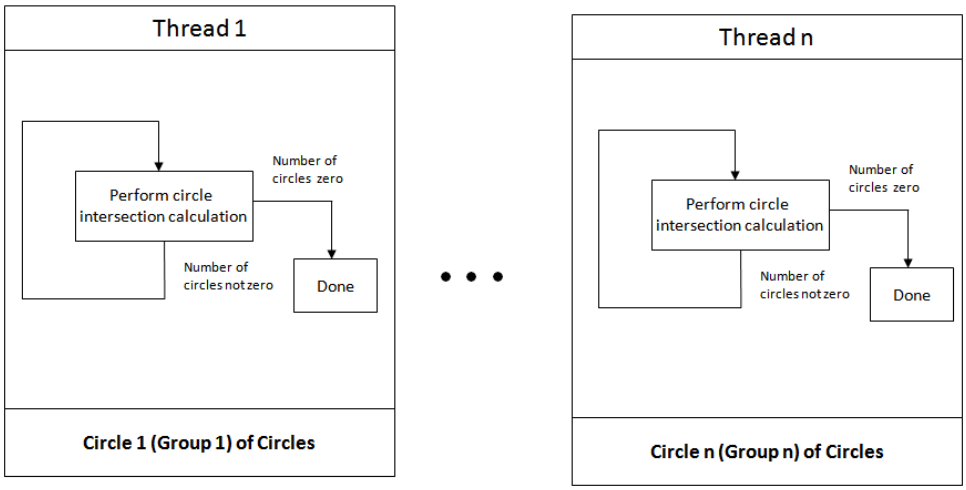


Figure 4.8: Circle Clipping, Parallel Approach Circle-B, Step 2

4.2.3 Parallel Approach Circle-C for Circle Clipping for POSIX Threads

The Approach Circle-C uses m number of threads per side of the clipping region. For the approach, four blocks of threads are used. Each block computes trivial accept, trivial reject and intersection clipping of circle for one of the four sides of the clipping region. These blocks are executed simultaneously. This approach

only works in POSIX threads. In GPU it is not possible to create and execute simultaneously more than one kernel (parallel) function at a time. Approach Circle-C still has significant amount of speedup improvement over its serial counterpart.

4.3 Supercomputer Environment Implementation

POSIX Threads are used for the implementations of the approaches on the supercomputer environment. In each step of an approach, set of k POSIX threads is created. Each thread processes a group of circles in parallel. For Approach Circle-A, each thread performs trivial accept and trivial reject and computation of intersection calculations and completes the clipping for one (or group) of circles. For Approach Circle-B, each thread tests for trivial accept and reject. Then, another set of threads is created and computes the intersection calculations and completes the clipping for one (or group) of circles.

4.4 GPU Environment Implementation

The GPU implementation of the parallel circle clipping is developed with NVidia CUDA technology. For Approach Circle-A, one CUDA thread is created per circle. All circles are clipped at same time. Every thread tests for trivial accept, trivial reject of a circle and computes intersection calculations. For Approach Circle-B, there is first one thread per circle. Each thread tests for the trivial accept and trivial reject. Once this step is over, more threads are created. Each thread computes the intersection(s) calculations and completes the clipping for one circle.

CHAPTER 5

EXPERIMENTS, RESULTS, ANALYSIS

In this chapter, the performance experiments to evaluate the performance of the new parallel line and circle clipping methods are presented. The experiments consider performance against serial, standalone CPU execution results of the line and circle clipping.

The 3D line data set used in the tests has one million randomly generated lines. Every line in it is defined by coordinates. The clipping region is axis-aligned and 3D rectangular shape. It is defined by its right, left, top, bottom, near and far planes. The clipping region is cuboid like shape with coordinates of (0,0,0) as the bottom-left-near and (100,100,100) as the top-right-far. Every x, y and z coordinate value is generated within the range of -100 to 200. This dataset has 50% of its lines lines lying only partially within the clipping region. The remaining lines are a mix of trivial accept and reject cases.

The 2D circle data set used in the tests has one million randomly generated circles. Every circle in it is defined by a center and radius. The clipping region is axis-aligned and rectangular. It is defined by its left, right, up and low bounds. The left bound is x=10. The bottom bound is y=10. The right bound is x=87. The top

bound is y=87. Every x and y coordinate value of the center of the circle is generated within the range of -100 to 200 and the radius of the circle is generated within the range of 1 to 100. This data set has 50% of its circles lying only partially within the clipping region. The remaining circles are a mix of trivial accept and reject cases.

5.1 Multi-threading Tests (Altix Cluster Computer)

The two environments for the experiments are the ones described in Chapter 2. The tests on the Altix cluster were for POSIX threads implementation. The tests on the GPUs were for CUDA implementations of the approaches.

5.2 Multi-threading Results

Here, the experiments of the performance of our two parallel approaches to Cohen-Sutherland line clipping and Fast Circle Clipping for multi-threaded cluster computers using the POSIX threads are reported.

5.2.1 Cohen-Sutherland Line Clipping Results

First, results for line clipping using multi-threading are reported.

5.2.1.1 Multi-threading on 1 CPU

Table 5.1 shows the serial execution time on one CPU compared to the multi-threading time with two threads on one CPU for Approach A for cases of 1000 to one million lines. Cases of less than one million lines used only a subset of the test dataset described at the start of this chapter. The speedup for less than 10000 lines

Table 5.1: Result Summary, Line Clipping Approach A, 1 CPU

# Lines	Serial Time	Time (2)	Speedup	Time (4)	Speedup
1000	0.0002	0.0004	0.43	0.0006	0.34
5000	0.0010	0.0010	0.92	0.0013	0.77
10000	0.0019	0.0018	1.09	0.0018	1.10
50000	0.0099	0.0083	1.20	0.0084	1.18
100000	0.0215	0.0159	1.35	0.0160	1.34
500000	0.1243	0.1055	1.18	0.0976	1.27
1000000	0.2578	0.1969	1.31	0.2019	1.28

Table 5.2: Result Summary, Line Clipping Approach B, 1 CPU

# Lines	Serial Time	Time (2)	Speedup	Time (4)	Speedup
1000	0.0002	0.0008	0.25	0.0012	0.16
5000	0.0010	0.0029	0.33	0.0018	0.55
10000	0.0019	0.0023	0.85	0.0028	0.69
50000	0.0099	0.0100	0.99	0.0102	0.98
100000	0.0215	0.0219	0.98	0.0217	0.99
500000	0.1243	0.1222	1.02	0.1201	1.04
1000000	0.2578	0.2455	1.05	0.2455	1.05

is under 1. As the number of lines increases, actual speedups are observed. For example, speedup for 100,000 lines and 1,000,000 lines is close to 1.3. Parallelization gives little improvement here because only one CPU is used.

Results for Approach B are shown in Table 5.2. Speedups are lower for Approach B than for Approach A. The speedup is less because the Approach B has to complete the trivial accept or reject tests for all lines before computing intersections for lines that are not trivial cases. Therefore, any line neither trivially rejected or

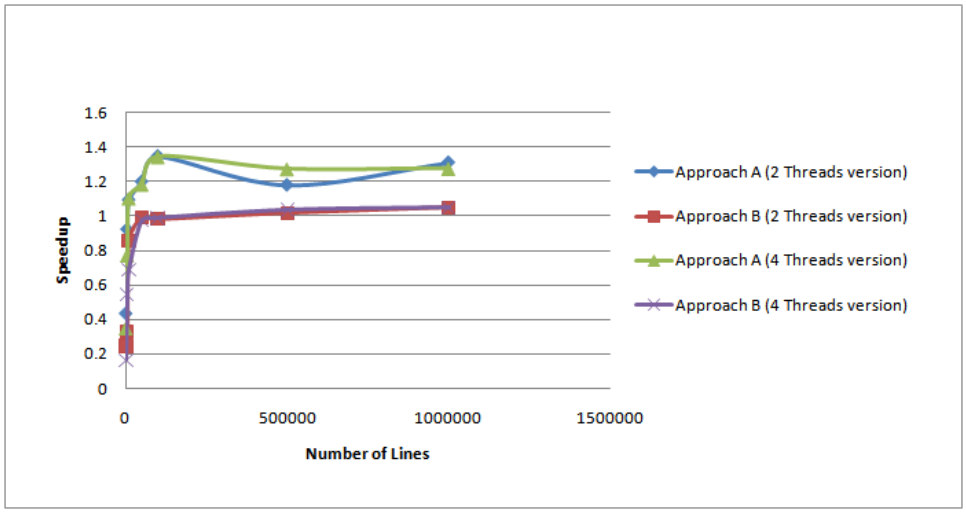


Figure 5.1: Speedups on 1 CPU for Approaches A and B of Cohen-Sutherland Line Clipping

accepted has to wait to finish its processing until all the trivial reject or accept tests of all the lines are done. Also since the number of threads created in Approach B are twice that of in Approach A, thread creation and destruction time is about doubled from Approach A.

We also tested times for having four threads on one CPU for the approaches. Parallel execution time results for four threads are shown in Table 5.1 and Table 5.2. Times using four threads are not substantially better than using two threads.

Figure 5.1 shows the speedup plots.

Table 5.3: Result Summary I, Line Clipping Approach A, 2 CPU

# Lines	Serial Time	Time (2)	Speedup	Time (4)	Speedup
1000	0.0002	0.0006	0.34	0.0008	0.25
5000	0.0010	0.0010	0.96	0.0013	0.76
10000	0.0019	0.0019	1.03	0.0020	0.95
50000	0.0099	0.0078	1.28	0.0083	1.19
100000	0.0215	0.0132	1.62	0.0122	1.76
500000	0.1243	0.0532	2.34	0.0529	2.35
1000000	0.2578	0.1046	2.46	0.1160	2.22

Table 5.4: Result Summary II, Line Clipping Approach A, 2 CPU

# Lines	Serial Time	Time (8)	Speedup	Time (16)	Speedup
1000	0.0002	0.0012	0.16	0.0017	0.12
5000	0.0010	0.0018	0.53	0.0028	0.35
10000	0.0019	0.0041	0.48	0.0030	0.65
50000	0.0099	0.0078	1.27	0.0098	1.02
100000	0.0215	0.0141	1.52	0.0120	1.79
500000	0.1243	0.0464	2.68	0.0478	2.60
1000000	0.2578	0.0931	2.77	0.0957	2.69

5.2.1.2 Multi-Threading on Multiple CPUs

Next, the results of various parallel techniques of Cohen-Sutherland Line Clipping on multiple CPUs are reported.

Tables 5.3 and 5.4 compare the times and speedups for 2, 4, 8 and 16 threads on 2 CPUs versus serial processing time for Approach A for cases of 1000 to one million lines.

Table 5.5: Result Summary I, Line Clipping Approach B, 2 CPU

# Lines	Serial Time	Time (2)	Speedup	Time (4)	Speedup
1000	0.0002	0.0009	0.22	0.0013	0.15
5000	0.0010	0.0014	0.68	0.0017	0.55
10000	0.0019	0.0023	0.83	0.0059	0.33
50000	0.0099	0.0114	0.87	0.0097	1.03
100000	0.0215	0.0208	1.03	0.0170	1.26
500000	0.1243	0.0645	1.93	0.0680	1.83
1000000	0.2578	0.2351	1.10	0.1400	1.84

Table 5.6: Result Summary II, Line Clipping Approach B, 2 CPU

# Lines	Serial Time	Time (8)	Speedup	Time (16)	Speedup
1000	0.0002	0.0026	0.07	0.0042	0.05
5000	0.0010	0.0027	0.36	0.0044	0.22
10000	0.0019	0.0042	0.46	0.0056	0.34
50000	0.0099	0.0106	0.93	0.0132	0.75
100000	0.0215	0.0180	1.19	0.0200	1.07
500000	0.1243	0.0682	1.82	0.0708	1.76
1000000	0.2578	0.1281	2.01	0.1291	2.00

The maximum speedup is achieved with Approach A on 2 CPUs with 8 threads. Tables 5.5 and 5.6 compare the times for 2, 4, 8 and 16 threads on 2 CPUs versus serial processing time for Approach B for cases of 1000 to one million lines. Figure 5.2 shows results for 2, 4, 8 and 16 threads on 2 CPUs for both approaches as speedup plots. Here, Approach A has proven to be more efficient than Approach B. The highest level of parallelism is achieved when there is sufficiently large number of lines and enough threads.

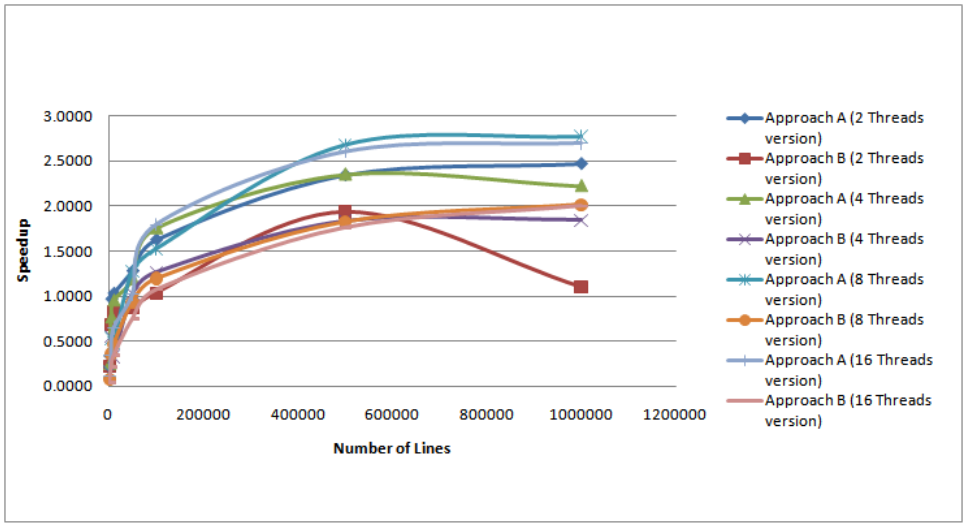


Figure 5.2: Speedups on 2 CPUs for Approaches A and B of Line Clipping

Table 5.7: Result Summary, Line Clipping Approach A, 4 CPU

# Lines	Serial Time	Time (4)	S_4	Time (8)	S_8	Time (16)	S_{16}
1000	0.0002	0.0006	0.33	0.0010	0.18	0.0019	0.10
5000	0.0010	0.0012	0.82	0.0016	0.60	0.0024	0.41
10000	0.0019	0.0012	1.61	0.0021	0.94	0.0032	0.61
50000	0.0099	0.0046	2.14	0.0055	1.81	0.0078	1.27
100000	0.0215	0.0080	2.67	0.0079	2.73	0.0110	1.95
500000	0.1243	0.0511	2.43	0.0485	2.56	0.0330	3.76
1000000	0.2578	0.0909	2.84	0.0698	3.69	0.0627	4.11

Table 5.8: Result Summary, Line Clipping Approach B, 4 CPU

# Lines	Serial Time	Time (4)	S_4	Time (8)	S_8	Time (16)	S_{16}
1000	0.0002	0.0034	0.06	0.0027	0.07	0.0044	0.05
5000	0.0010	0.0032	0.30	0.0028	0.34	0.0038	0.25
10000	0.0019	0.0028	0.70	0.0040	0.49	0.0056	0.16
50000	0.0099	0.0094	1.05	0.0108	0.92	0.0121	0.82
100000	0.0215	0.0221	0.97	0.0161	1.33	0.0229	0.47
500000	0.1243	0.0661	1.88	0.1018	1.22	0.0458	2.72
1000000	0.2578	0.1190	2.17	0.1121	2.30	0.0908	2.84

Table 5.7 compares the times and speedups for 4, 8 and 16 threads on 4 CPUs versus serial processing time for Approach A for cases of 1000 to one million lines. The speedups columns are labeled as S. The maximum speedup is achieved with Approach A on 4 CPUs with 16 threads. Table 5.8 compares the times and speedups for 4, 8 and 16 threads on 4 CPUs versus serial processing time for Approach B for cases of 1000 to one million lines.

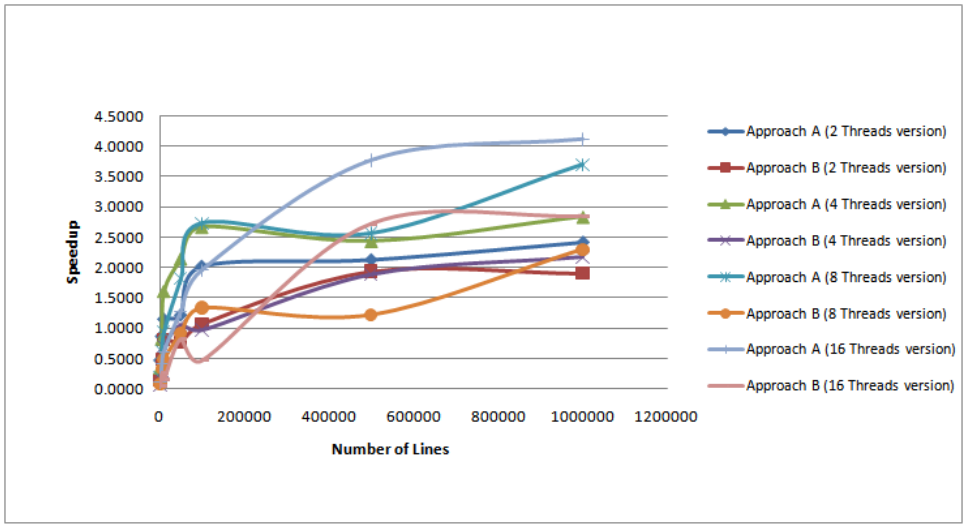


Figure 5.3: Speedups on 4 CPUs for Approaches A and B of Line Clipping

Figure 5.3 shows speedup plots for various numbers of threads on 4 CPUs. Here, Approach A has proven to be more efficient than Approach B. The highest degree of parallelism is achieved when there is sufficiently large number of lines and enough threads.

Table 5.9: Result Summary I, Line Clipping Approach A, 8 CPU

# Lines	Serial Time	Time (16)	S_{16}	Time (32)	S_{32}
1000	0.0002	0.0021	0.09	0.0044	0.04
5000	0.0010	0.0041	0.23	0.0063	0.15
10000	0.0019	0.0033	0.59	0.0051	0.38
50000	0.0099	0.0079	1.26	0.0099	1.00
100000	0.0215	0.0095	2.26	0.0133	1.61
500000	0.1243	0.0224	5.56	0.0236	5.28
1000000	0.2578	0.0438	5.89	0.0392	6.57

Table 5.10: Result Summary II, Line Clipping Approach A, 8 CPU

# Lines	Serial Time	Time (48)	S_{48}	Time (64)	S_{64}
1000	0.0002	0.0059	0.03	0.0095	0.02
5000	0.0010	0.0075	0.13	0.0099	0.10
10000	0.0019	0.0072	0.27	0.0087	0.22
50000	0.0099	0.0133	0.75	0.0132	0.75
100000	0.0215	0.0171	1.20	0.0183	1.17
500000	0.1243	0.0367	3.39	0.0364	3.41
1000000	0.2578	0.0466	5.53	0.0537	4.8

Tables 5.9 and 5.10 compare the times for 16, 32, 48 and 64 threads on 8 CPUs versus serial processing time for Approach A for cases of 1000 to one million lines. The maximum speedup is achieved with Approach A on 8 CPUs with 32 threads. Tables 5.11 and 5.12 compare the times for 16, 32, 48, and 64 threads on 8 CPUs versus serial processing time for Approach B for cases of 1000 to one million lines. Figure 5.4 shows speedup plots for various numbers of threads on 8 CPUs. Here, Approach A has proven to be more efficient than Approach B. The highest level of

Table 5.11: Result Summary I, Line Clipping Approach B, 8 CPU

# Lines	Serial Time	Time (16)	S_{16}	Time (32)	S_{32}
1000	0.0002	0.0059	0.03	0.0105	0.02
5000	0.0010	0.0054	0.18	0.0123	0.08
10000	0.0019	0.0086	0.23	0.0435	0.04
50000	0.0099	0.0151	0.66	0.0179	0.55
100000	0.0215	0.0171	1.26	0.0329	0.65
500000	0.1243	0.0444	2.8	0.0659	1.89
1000000	0.2578	0.1195	2.16	0.1003	2.57

Table 5.12: Result Summary II, Line Clipping Approach B, 8 CPU

# Lines	Serial Time	Time (48)	S_{48}	Time (64)	S_{64}
1000	0.0002	0.0155	0.01	0.0189	0.01
5000	0.0010	0.0161	0.06	0.0203	0.05
10000	0.0019	0.0149	0.13	0.0199	0.10
50000	0.0099	0.0222	0.45	0.0318	0.31
100000	0.0215	0.0360	0.60	0.0358	0.60
500000	0.1243	0.0712	1.75	0.0904	1.37
1000000	0.2578	0.1497	1.72	0.1336	1.93

parallelism is achieved when there is a sufficiently large number of lines and enough threads.

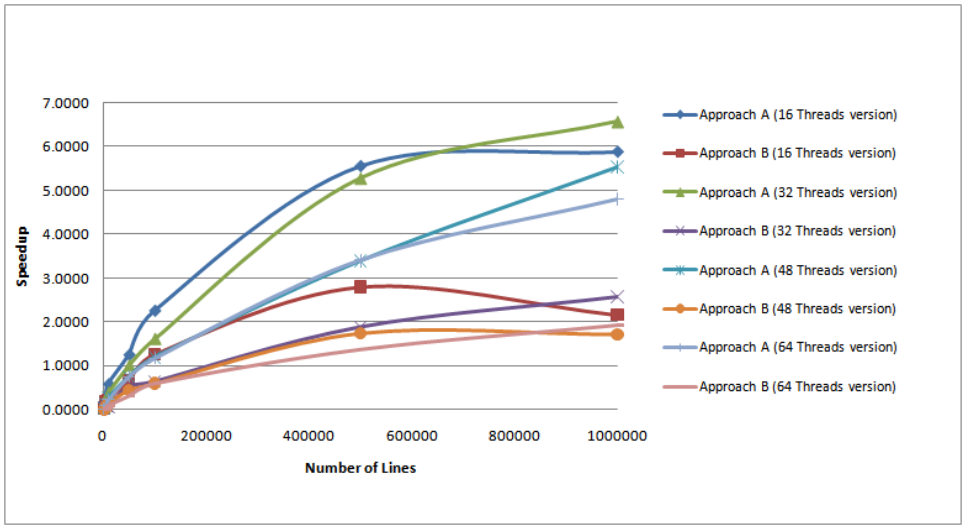


Figure 5.4: Speedups on 8 CPUs for Approaches A and B of Cohen-Sutherland Line Clipping

Table 5.13: Result Summary I, Line Clipping Approach A, 16 CPU

# Lines	Serial Time	Time (16)	S_{16}	Time (32)	S_{32}
1000	0.0002	0.0043	0.04	0.0048	0.04
5000	0.0010	0.0027	0.35	0.0053	0.18
10000	0.0019	0.0039	0.50	0.0058	0.33
50000	0.0099	0.0050	1.97	0.0083	1.19
100000	0.0215	0.0115	1.86	0.0105	2.04
500000	0.1243	0.0213	5.84	0.0184	6.76
1000000	0.2578	0.0338	7.64	0.0434	5.94

Table 5.14: Result Summary II, Line Clipping Approach A, 16 CPU

# Lines	Serial Time	Time (64)	S_{64}	Time (128)	S_{128}
1000	0.0002	0.0048	0.04	0.0198	0.01
5000	0.0010	0.0053	0.18	0.0186	0.05
10000	0.0019	0.0058	0.33	0.0037	0.52
50000	0.0099	0.0083	1.19	0.0212	0.47
100000	0.0215	0.0105	2.04	0.0284	0.76
500000	0.1243	0.0184	6.76	0.0486	2.56
1000000	0.2578	0.0434	5.94	0.0905	2.85

Tables 5.13 and 5.14 compare the times and speedups for 16, 32, 64 and 128 threads on 16 CPUs versus serial processing time for Approach A for cases of 1000 to one million lines. The maximum speedup is achieved with Approach A on 16 CPUs with 16 threads. Tables 5.15 and 5.16 compare the times and speedups for 16, 32, 64 and 128 threads on 16 CPUs versus serial processing time for Approach B for cases of 1000 to one million lines. Figure 5.5 shows speedup plots for various numbers of threads on 16 CPUs for both approaches. Here, Approach A has proven

Table 5.15: Result Summary I, Line Clipping Approach B, 16 CPU

# Lines	Serial Time	Time (16)	S_{16}	Time (32)	S_{32}
1000	0.0002	0.0063	0.03	0.0109	0.02
5000	0.0010	0.0059	0.16	0.0119	0.08
10000	0.0019	0.0082	0.24	0.0104	0.19
50000	0.0099	0.0081	1.22	0.0155	0.64
100000	0.0215	0.0142	1.51	0.0229	0.94
500000	0.1243	0.0426	2.92	0.0388	3.20
1000000	0.2578	0.0795	3.24	0.0646	3.99

Table 5.16: Result Summary II, Line Clipping Approach B, 16 CPU

# Lines	Serial Time	Time (64)	S_{64}	Time (128)	S_{128}
1000	0.0002R	0.0109	0.02	0.0186	0.01
5000	0.0010	0.0119	0.08	0.0345	0.03
10000	0.0019	0.0104	0.19	0.0450	0.04
50000	0.0099	0.0155	0.64	0.0438	0.23
100000	0.0215	0.0229	0.94	0.0577	0.37
500000	0.1243	0.0388	3.20	0.0957	1.30
1000000	0.2578	0.0646	3.99	0.0957	1.78

to be more efficient than Approach B. The highest degree of parallelism is achieved when there is a sufficiently large number of lines and enough threads.

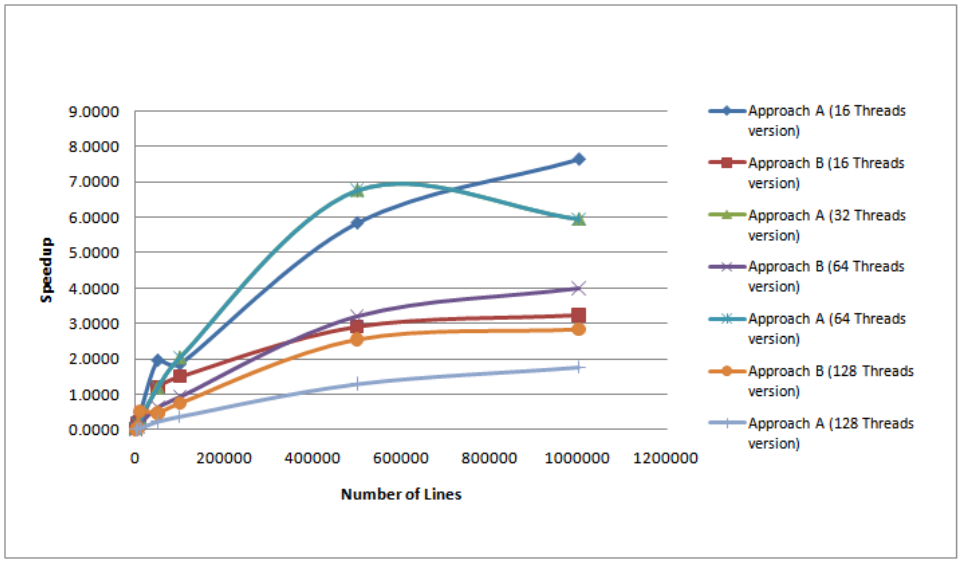


Figure 5.5: Speedups on 16 CPUs for Approaches A and B of Cohen-Sutherland Line Clipping

Table 5.17: Result Summary I, Line Clipping Approach A, 24 CPU

# Lines	Serial Time	Time (24)	S_{24}	Time (48)	S_{48}
1000	0.0002	0.0039	0.05	0.0077	0.02
5000	0.0010	0.0041	0.23	0.0077	0.13
10000	0.0019	0.0037	0.52	0.0081	0.24
50000	0.0099	0.0061	1.64	0.0099	1.00
100000	0.0215	0.0105	2.04	0.0164	1.31
500000	0.1243	0.0177	7.00	0.0255	4.88
1000000	0.2578	0.0241	10.69	0.0394	6.55

Table 5.18: Result Summary II, Line Clipping Approach A, 24 CPU

# Lines	Serial Time	Time (72)	S_{72}	Time (96)	S_{96}
1000	0.0002	0.0114	0.02	0.0154	0.01
5000	0.0010	0.0111	0.09	0.0137	0.07
10000	0.0019	0.0156	0.12	0.0159	0.12
50000	0.0099	0.0139	0.71	0.0166	0.60
100000	0.0215	0.0160	1.34	0.0227	0.94
500000	0.1243	0.0310	4.02	0.0411	3.03
1000000	0.2578	0.0426	6.05	0.0450	5.73

Tables 5.17 and 5.18 compare the times and speedups for 24, 48, 72 and 96 threads on 24 CPUs versus serial processing time for Approach A for cases of 1000 to one million lines. The maximum speedup is achieved with Approach A on 24 CPUs with 24 threads. Tables 5.19 and 5.20 compare the times and speedups for 24, 48, 72 and 96 threads on 24 CPUs versus serial processing time for Approach B for cases of 1000 to one million lines. Figure 5.6 shows speedup plots for 24, 48, 72 and 96 threads on 24 CPUs for both approaches. Here, Approach A has proven to be

Table 5.19: Result Summary I, Line Clipping Approach B, 24 CPU

# Lines	Serial Time	Time (24)	S_{24}	Time (48)	S_{48}
1000	0.0002	0.0067	0.03	0.0129	0.01
5000	0.0010	0.0074	0.13	0.0163	0.06
10000	0.0019	0.0075	0.26	0.0146	0.13
50000	0.0099	0.0103	0.96	0.0258	0.38
100000	0.0215	0.0163	1.32	0.0309	0.69
500000	0.1243	0.0365	3.41	0.0471	2.64
1000000	0.2578	0.0619	4.16	0.0757	3.40

Table 5.20: Result Summary II, Line Clipping Approach B, 24 CPU

# Lines	Serial Time	Time (72)	S_{72}	Time (96)	S_{96}
1000	0.0002	0.0209	0.01	0.0282	0.01
5000	0.0010	0.0201	0.05	0.0270	0.04
10000	0.0019	0.0215	0.09	0.0295	0.07
50000	0.0099	0.0306	0.32	0.0303	0.33
100000	0.0215	0.0415	0.52	0.0454	0.47
500000	0.1243	0.0647	1.92	0.0741	1.68
1000000	0.2578	0.0862	2.99	0.1099	2.35

more efficient than Approach B. The highest degree of parallelism is achieved when there is a sufficiently large number of lines and enough threads.

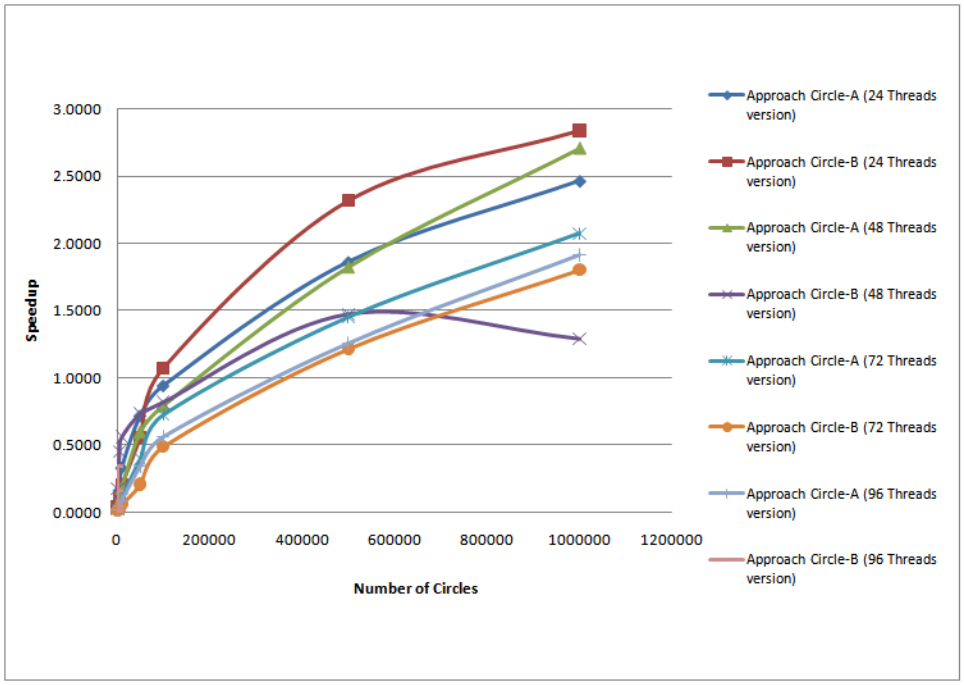


Figure 5.6: Speedups on 24 CPUs for Approaches A and B of Cohen-Sutherland Line Clipping

Table 5.21: Result Summary I, Line Clipping Approach A, 32 CPU

# Lines	Serial Time	Time (32)	S_{32}	Time (64)	S_{64}
1000	0.0002	0.0049	0.04	0.0104	0.02
5000	0.0010	0.0050	0.19	0.0102	0.09
10000	0.0019	0.0057	0.34	0.0092	0.21
50000	0.0099	0.0076	1.31	0.0108	0.92
100000	0.0215	0.0097	2.22	0.0141	1.52
500000	0.1243	0.0217	5.73	0.0280	4.45
1000000	0.2578	0.0216	11.91	0.0392	6.57

Table 5.22: Result Summary II, Line Clipping Approach A, 32 CPU

# Lines	Serial Time	Time (192)	S_{192}
1000	0.0002	0.0314	0.01
5000	0.0010	0.0276	0.03
10000	0.0019	0.0229	0.08
50000	0.0099	0.0289	0.34
100000	0.0215	0.0366	0.59
500000	0.1243	0.0412	3.02
1000000	0.2578	0.0671	3.84

Tables 5.21 and 5.22 compare the times and speedups for 32, 64, and 192 threads on 32 CPUs versus serial processing time for Approach A for cases of 1000 to one million lines. The maximum speedup is achieved with Approach A on 32 CPUs with 32 threads. Tables 5.23 and 5.24 compare the times and speedups for 32, 64 and 192 threads on 32 CPUs versus serial processing time for Approach B for cases of 1000 to one million lines. Figure 5.7 shows speedup plots for 32 and 64 threads on 32 CPUs for both approaches. Here, Approach A has proven to be more efficient than

Table 5.23: Result Summary I, Line Clipping Approach B, 32 CPU

# Lines	Serial Time	Time (32)	S_{32}	Time (64)	S_{64}
1000	0.0002	0.0094	0.02	0.0187	0.01
5000	0.0010	0.0092	0.10	0.0184	0.05
10000	0.0019	0.0099	0.20	0.0193	0.10
50000	0.0099	0.0176	0.56	0.0218	0.45
100000	0.0215	0.0195	1.22	0.0334	0.64
500000	0.1243	0.0664	1.87	0.0496	2.51
1000000	0.2578	0.0695	3.88	0.0642	4.02

Table 5.24: Result Summary II, Line Clipping Approach B, 32 CPU

# Lines	Serial Time	Time (192)	S_{192}
1000	0.0002	0.0576	0.003
5000	0.0010	0.0541	0.02
10000	0.0019	0.0554	0.03
50000	0.0099	0.0741	0.13
100000	0.0215	0.0756	0.28
500000	0.1243	0.1122	1.11
1000000	0.2578	0.1763	1.46

Approach B. The highest degree of parallelism is achieved when there is a sufficiently large number of lines and enough threads.

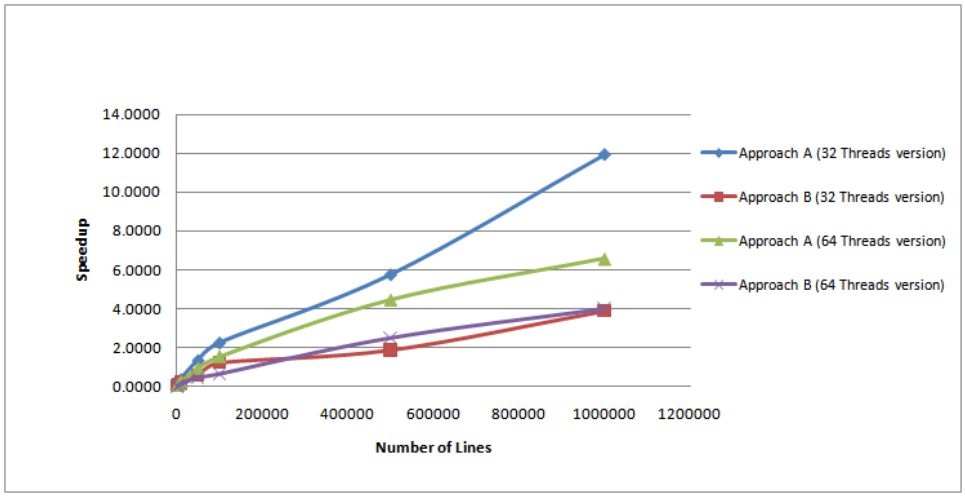


Figure 5.7: Speedups on 32 CPUs for Approaches A and B of Cohen-Sutherland Line Clipping

Table 5.25: Results Summary, Circle-A, 1-CPU

# Circles	Serial Time	Time (2)	S_2	Time (4)	S_4
1000	0.0002	0.0008	0.30	0.0007	0.35
5000	0.0007	0.0014	0.48	0.0013	0.50
10000	0.0013	0.0023	0.56	0.0022	0.60
50000	0.0051	0.0101	0.51	0.0096	0.53
100000	0.0128	0.0202	0.63	0.0189	0.68
500000	0.0634	0.1045	0.61	0.0963	0.66
1000000	0.1266	0.2092	0.61	0.1923	0.66

5.2.2 Fast Circle Clipping Results

Next, results for circle clipping are reported.

5.2.2.1 Multi-threading on 1 CPU

Table 5.25 shows the execution times on one CPU compared to multi-threading with up to four threads on one CPU for Approach Circle-A for cases of 1000 to one million circles. Speedups are also shown. They are always less than 1. As the number of circles increases, speedups speedups do increase, however. Parallelization gives little improvement here because only one CPU is used. Hence, there is limited parallelism.

Table 5.26: Results Summary, Circle-B, 1-CPU

# Circles	Serial Time	Time (2)	S_2	Time (4)	S_4
1000	0.0002	0.0009	0.27	0.0016	0.15
5000	0.0007	0.0013	0.51	0.0021	0.32
10000	0.0013	0.0020	0.66	0.0028	0.48
50000	0.0051	0.0074	0.69	0.0082	0.63
100000	0.0128	0.0143	0.89	0.0153	0.84
500000	0.0634	0.0795	0.80	0.0793	0.80
1000000	0.1266	0.1567	0.81	0.1590	0.80

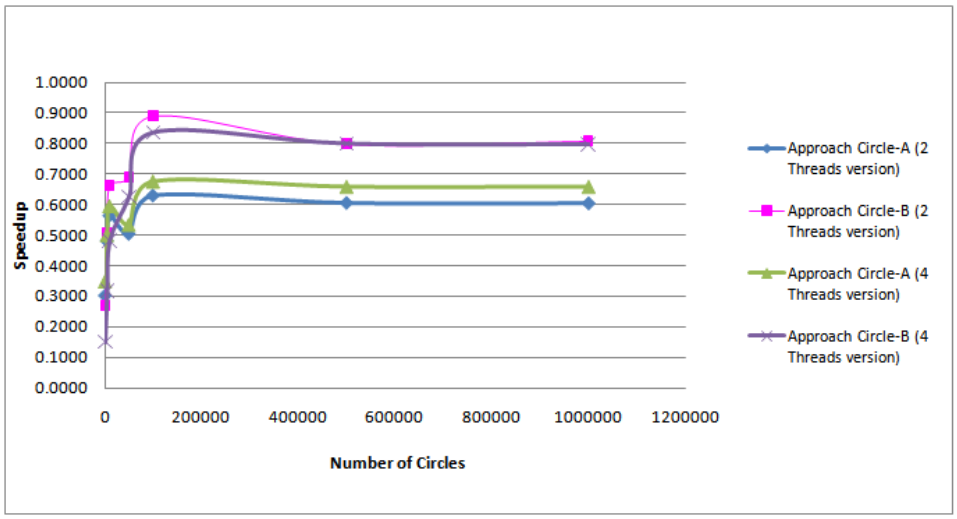


Figure 5.8: Speedups on 1 CPU for Approaches Circle-A and Circle-B

Results for Approach Circle-B are shown in Table 5.26. Speedups are similar for Approach Circle-B and Approach Circle-A. Figure 5.8 shows the result as speedup plots. Times using four threads are not substantially better than using two threads.

5.2.2.2 Multi-threading on multiple CPUs

Next, the results of various parallel techniques of Fast Circle Clipping on multiple CPUs are reported.

Tables 5.27 and 5.28 present the times and speedups for 2, 4, 8 and 16 threads on 2 CPUs versus serial processing time for Approach Circle-A for cases of 1000 to one million circles. The maximum speedup is achieved here with 8 threads.

Tables 5.29 and 5.30 compare the times and speedup for 2, 4, 8 and 16 threads on 2 CPUs versus serial processing time for Approach Circle-B for cases of 1000 to one million circles.

Table 5.27: Results Summary I, Circle-A, 2-CPU

# Circles	Serial Time	Time (2)	S_2	Time (4)	S_4
1000	0.0002	0.0006	0.39	0.0006	0.39
5000	0.0007	0.0012	0.57	0.0012	0.57
10000	0.0013	0.0021	0.64	0.0021	0.64
50000	0.0051	0.0119	0.43	0.0119	0.43
100000	0.0128	0.0358	0.36	0.0358	0.36
500000	0.0634	0.2907	0.22	0.2907	0.22
1000000	0.1266	0.5792	0.22	0.5792	0.22

Table 5.28: Results Summary II, Circle-A, 2-CPU

# Circles	Serial Time	Time (8)	S_8	Time (16)	S_{16}
1000	0.0002	0.0001	2.20	0.0020	0.13
5000	0.0007	0.0018	0.36	0.0026	0.26
10000	0.0013	0.0019	0.71	0.0035	0.37
50000	0.0051	0.0128	0.40	0.0084	0.61
100000	0.0128	0.0175	0.73	0.0223	0.57
500000	0.0634	0.0515	1.23	0.0571	1.11
1000000	0.1266	0.1006	1.26	0.1375	0.92

We note the Approach Circle-C was not investigated here because it is not applicable for 2 CPUs, because four sets of threads, each for one side of clipping region, need to be run simultaneously in it. Figure 5.9 shows a speedup plot for various number of threads on 2 CPUs. Here, Approach Circle-A has proven to be more efficient than Approach Circle-B. The highest level of parallelism is achieved when there is a sufficiently large number of circles and enough threads.

Table 5.29: Results Summary I, Circle-B, 2-CPU

# Circles	Serial Time	Time (2)	S_2	Time (4)	S_4
1000	0.0002	0.0009	0.29	0.0012	0.20
5000	0.0007	0.0013	0.53	0.0017	0.39
10000	0.0013	0.0023	0.57	0.0022	0.60
50000	0.0051	0.0116	0.44	0.0074	0.69
100000	0.0128	0.0229	0.56	0.0186	0.69
500000	0.0634	0.1111	0.57	0.0661	0.96
1000000	0.1266	0.2125	0.60	0.1198	1.06

Table 5.30: Results Summary II, Circle-B, 2-CPU

# Circles	Serial Time	Time (8)	S_8	Time (16)	S_{16}
1000	0.0002	0.0021	0.12	0.0042	0.06
5000	0.0007	0.0028	0.24	0.0049	0.14
10000	0.0013	0.0032	0.41	0.0054	0.25
50000	0.0051	0.0065	0.79	0.0097	0.52
100000	0.0128	0.0112	1.13	0.0175	0.73
500000	0.0634	0.0394	1.61	0.0674	0.94
1000000	0.1266	0.1020	1.24	0.0920	1.38

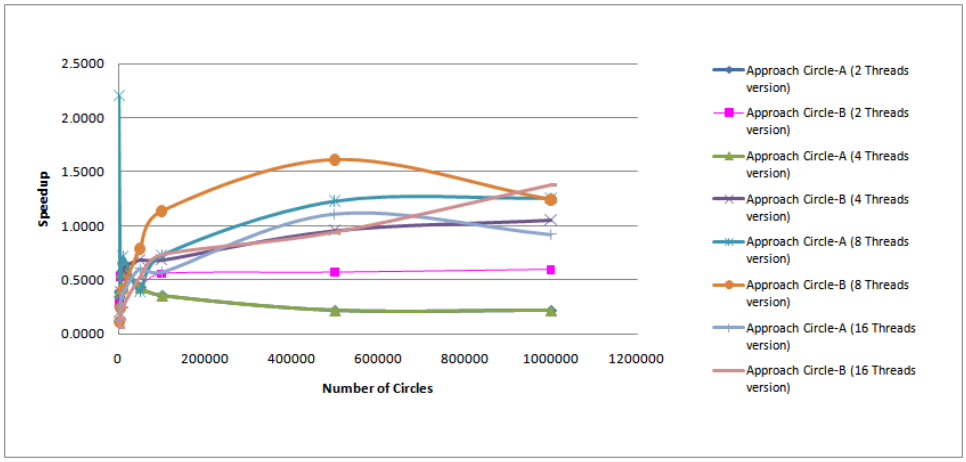


Figure 5.9: Speedup Plots for 2 CPUs for Circle-A and Circle-B

Table 5.31: Results Summary I, Circle-A, 4-CPU

# Circles	Serial Time	Time (4)	S_4	Time (8)	S_8
1000	0.0002	0.0007	0.35	0.0012	0.21
5000	0.0007	0.0014	0.47	0.0018	0.36
10000	0.0013	0.0023	0.57	0.0030	0.44
50000	0.0051	0.0191	0.27	0.0096	0.53
100000	0.0128	0.0340	0.38	0.0193	0.66
500000	0.0634	0.1558	0.41	0.0571	1.11
1000000	0.1266	0.3089	0.41	0.1384	0.92

Table 5.32: Results Summary II, 16 Threads All Approaches, 4-CPU

# Circles	Serial Time	T (A)	S (A)	T (B)	S (B)	T (C)	S (C)
1000	0.0002	0.0021	0.12	0.0045	0.06	0.0043	0.06
5000	0.0007	0.0028	0.24	0.0048	0.14	0.0044	0.15
10000	0.0013	0.0035	0.38	0.0050	0.26	0.0059	0.22
50000	0.0051	0.0103	0.49	0.0105	0.49	0.0092	0.56
100000	0.0128	0.0147	0.87	0.0150	0.85	0.0139	0.92
500000	0.0634	0.0639	0.99	0.0538	1.18	0.0463	1.37
1000000	0.1266	0.0924	1.37	0.0941	1.35	0.0990	1.28

Table 5.31 compares the times and speedups for 4 and 8 threads on 4 CPUs versus serial processing time for Approach Circle-A for cases of 1000 to one million circles. For 8 threads, only on 500,000 circles was there a speedup.

Table 5.33 compares the times and speedups for 4 and 8 threads on 4 CPUs versus serial processing time for Approach Circle-B for cases of 1000 to one million circles.

Table 5.33: Results Summary I, Circle-B, 4-CPU

# Circles	Serial Time	Time (4)	S_4	Time (8)	S_8
1000	0.0002	0.0013	0.19	0.0023	0.11
5000	0.0007	0.0018	0.38	0.0027	0.25
10000	0.0013	0.0024	0.56	0.0033	0.35
50000	0.0051	0.0076	0.68	0.0069	0.74
100000	0.0128	0.0164	0.78	0.0172	0.74
500000	0.0634	0.0660	0.96	0.0448	1.42
1000000	0.1266	0.1323	0.96	0.1369	0.92

Table 5.34: Results Summary I, Circle-C, 4-CPU

# Circles	Serial Time	Time (4)	S_4	Time (8)	S_8
1000	0.0002	0.0011	0.24	0.0017	0.14
5000	0.0007	0.0014	0.49	0.0020	0.33
10000	0.0013	0.0021	0.65	0.0030	0.45
50000	0.0051	0.0053	0.96	0.0062	0.82
100000	0.0128	0.0113	1.13	0.0117	1.09
500000	0.0634	0.0508	1.25	0.0413	1.54
1000000	0.1266	0.1034	1.23	0.0921	1.38

Table 5.34 compares the times and speedups for 4 and 8 threads on 4 CPUs versus serial processing time for Approach Circle-C for cases of 1000 to one million circles. The maximum speedup is achieved with 8 threads, but only for 100,000 circles and up is speedup more than 1.

Table 5.32 presents times and speedups using 16 threads on 4 CPUs for all approaches. These 16 thread results are usually best if there are 100,000 circles or more. Figure 5.10 shows speedup plots for multi-threadings on 4 CPUS for all

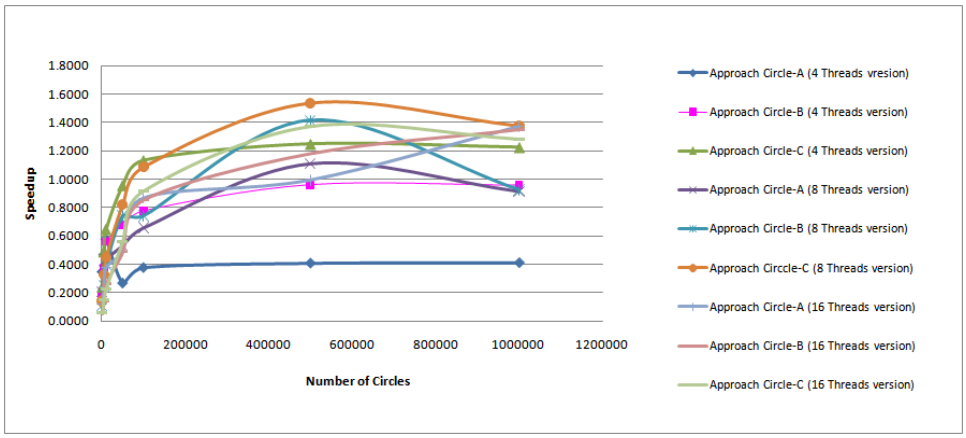


Figure 5.10: Speedup Plots for 4 CPUs for Circle-A, Circle-B and Circle-C.

three approaches. Here, Approach Circle-A has proven to be more efficient than Approaches Circle-B and Circle-C. The highest level of parallelism is achieved when there is sufficiently large number of circles and enough threads.

Approach Circle-C uses 16 threads in total here, (each side of the clipping region has 4 threads). Fast Circle Clipping algorithm is running in parallel for all sides simultaneously, which allows Circle-C to have more simultaneous work available than in Approaches Circle-A and Circle-B, which have the each side processed in serial manner.

Table 5.35: Results Summary I, Circle-A, 8-CPU

# Circles	Serial Time	Time (8)	S_8	Time (16)	S_{16}
1000	0.0002	0.0025	0.10	0.0046	0.05
5000	0.0007	0.0028	0.24	0.0049	0.14
10000	0.0013	0.0035	0.38	0.0070	0.19
50000	0.0051	0.0073	0.70	0.0104	0.49
100000	0.0128	0.0100	1.28	0.0139	0.92
500000	0.0634	0.0437	1.45	0.0402	1.58
1000000	0.1266	0.0938	1.35	0.0585	2.16

Table 5.36: Results Summary II, Circle-A, 8-CPU

# Circles	Serial Time	Time (32)	S_{32}	Time (48)	S_{48}
1000	0.0002	0.0014	0.18	0.0014	0.18
5000	0.0007	0.0019	0.34	0.0018	0.37
10000	0.0013	0.0023	0.59	0.0023	0.58
50000	0.0051	0.0082	0.62	0.0055	0.93
100000	0.0128	0.0149	0.86	0.0106	1.20
500000	0.0634	0.0441	1.44	0.0470	1.35
1000000	0.1266	0.0715	1.77	0.0676	1.87

Tables 5.35 and 5.36 and part of Table 5.41 present the times and speedups for 8, 16, 32, 48 and 64 threads on 8 CPUs versus serial processing time for Approach Circle-A for cases of 1000 to one million circles. Tables 5.37 and 5.38 and part of Table 5.41 present the same experimental results for Approach Circle-B for cases of 1000 to one million circles. The results for Circle-C are presented in Tables 5.39 and 5.40 and part of Table 5.41. The maximum speedup is achieved with the Approach Circle-A on 8 CPUs with 16 threads.

Table 5.37: Results Summary I, Circle-B, 8-CPU

# Circles	Serial Time	Time (8)	S_8	Time (16)	S_{16}
1000	0.0002	0.0304	0.00	0.0022	0.11
5000	0.0007	0.0299	0.00	0.0034	0.19
10000	0.0013	0.0282	0.00	0.0030	0.45
50000	0.0051	0.0381	0.01	0.0093	0.55
100000	0.0128	0.0387	0.01	0.0160	0.80
500000	0.0634	0.0730	0.07	0.0641	0.99
1000000	0.1266	0.1127	0.14	0.1494	1.48

Table 5.38: Results Summary II, Circle-B, 8-CPU

# Circles	Serial Time	Time (32)	S_{32}	Time (48)	S_{48}
1000	0.0002	0.0043	0.06	0.0090	0.03
5000	0.0007	0.0054	0.12	0.0218	0.03
10000	0.0013	0.0052	0.25	0.0100	0.13
50000	0.0051	0.0091	0.56	0.0138	0.37
100000	0.0128	0.0194	0.66	0.0262	0.49
500000	0.0634	0.0711	0.89	0.0557	1.14
1000000	0.1266	0.0994	1.27	0.0731	1.73

Figure 5.11 shows speedup plots for multi-threadings on 8 CPUs. Here, Approach Circle-A has proven to be more efficient than Approach Circle-B and Approach Circle-C. The highest degree of parallelism was achieved when there is a sufficiently large number of circles and enough threads.

Table 5.39: Results Summary I, Circle-C, 8-CPU

# Circles	Serial Time	Time (8)	S_8	Time (16)	S_{16}
1000	0.0002	0.0016	0.15	0.0046	0.05
5000	0.0007	0.0020	0.33	0.0034	0.20
10000	0.0013	0.0035	0.38	0.0049	0.27
50000	0.0051	0.0071	0.72	0.0089	0.58
100000	0.0128	0.0114	1.12	0.0128	1.00
500000	0.0634	0.0469	1.35	0.0434	1.46
1000000	0.1266	0.0794	1.59	0.0809	1.57

Table 5.40: Results Summary II, Circle-C, 8-CPU

# Circles	Serial Time	Time (32)	S_{32}
1000	0.0002	0.0060	0.04
5000	0.0007	0.0061	0.11
10000	0.0013	0.0083	0.16
50000	0.0051	0.0108	0.47
100000	0.0128	0.0164	0.78
500000	0.0634	0.0456	1.39
1000000	0.1266	0.0850	1.49

Table 5.41: Results Summary III, Circle-A, Circle-B, Circle-C, 64 Threads 8-CPU

# Circles	Serial Time	Time A	S_A	Time B	S_B	Time C	S_C
1000	0.0002	0.0088	0.03	0.0182	0.01	0.0135	0.02
5000	0.0007	0.0094	0.07	0.0169	0.04	0.0131	0.05
10000	0.0013	0.0092	0.14	0.0165	0.08	0.0134	0.10
50000	0.0051	0.0173	0.29	0.0246	0.21	0.0433	0.12
100000	0.0128	0.0213	0.60	0.0291	0.44	0.0212	0.60
500000	0.0634	0.0646	0.98	0.0620	1.02	0.0777	0.82
1000000	0.1266	0.0924	1.37	0.1050	1.21	0.0904	1.40

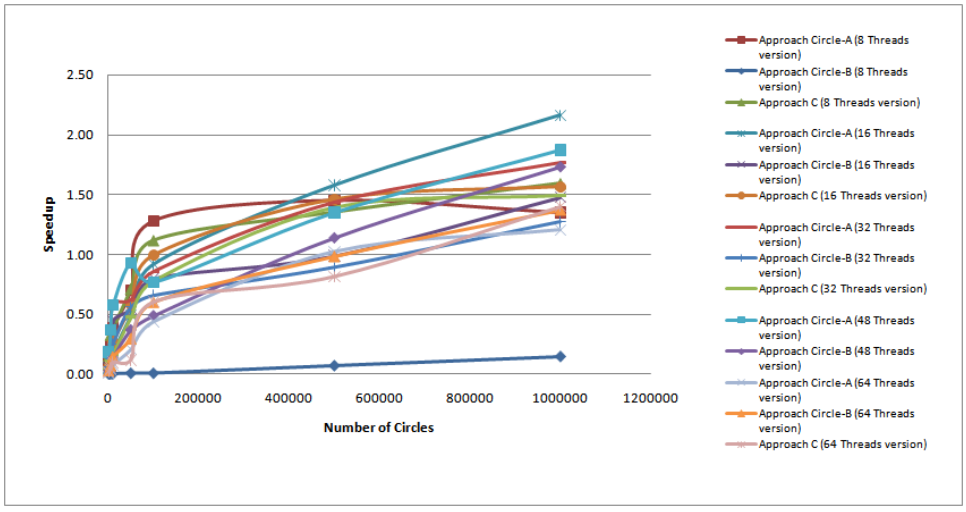


Figure 5.11: Speedups on 8 CPUs for Approaches Circle-A, Circle-B and Circle-C of Fast Circle Clipping

Table 5.42: Results Summary I, Circle-A, 16-CPU

# Circles	Serial Time	Time (16)	S_{16}	Time (32)	S_{32}
1000	0.0002	0.0031	0.08	0.0049	0.05
5000	0.0007	0.0028	0.23	0.0056	0.12
10000	0.0013	0.0034	0.39	0.0059	0.23
50000	0.0051	0.0067	0.76	0.0098	0.52
100000	0.0128	0.0117	1.10	0.0111	1.15
500000	0.0634	0.0306	2.07	0.0299	2.12
1000000	0.1266	0.0944	1.34	0.0637	1.99

Table 5.43: Results Summary II, Circle-A, 16-CPU

# Circles	Serial Time	Time (64)	S_{64}	Time (128)	S_{128}
1000	0.0002	0.0107	0.02	0.0205	0.01
5000	0.0007	0.0090	0.07	0.0188	0.04
10000	0.0013	0.0109	0.12	0.0218	0.06
50000	0.0051	0.0129	0.40	0.0266	0.19
100000	0.0128	0.0220	0.58	0.0307	0.42
500000	0.0634	0.0464	1.37	0.0648	0.98
1000000	0.1266	0.0666	1.90	0.0921	1.38

Tables 5.42 and 5.43 present the times and speedups for 16, 32, 64 and 128 threads on 16 CPUs versus serial processing time for Approach Circle-A for cases of 1000 to one million circles. Tables 5.44 and 5.45 compare the times and speedups for 16, 32, 64 and 128 threads on 16 CPUs versus serial processing time for Approach Circle-B for cases of 1000 to one million circles. Tables 5.46 and 5.47 compare the times and speedups for 16, 32, 64 and 128 threads on 16 CPUs versus serial processing time for Approach Circle-C for cases of 1000 to one million circles. Speedups

Table 5.44: Results Summary I, Circle-B, 16-CPU

# Circles	Serial Time	Time (16)	S_{16}	Time (32)	S_{32}
1000	0.0002	0.0050	0.05	0.0096	0.03
5000	0.0007	0.0051	0.13	0.0088	0.08
10000	0.0013	0.0072	0.18	0.0098	0.14
50000	0.0051	0.0084	0.61	0.0130	0.39
100000	0.0128	0.0123	1.04	0.0165	0.77
500000	0.0634	0.0291	2.18	0.0404	1.57
1000000	0.1266	0.0534	2.37	0.0592	2.14

Table 5.45: Results Summary II, Circle-B, 16-CPU

# Circles	Serial Time	Time (64)	S_{64}	Time (128)	S_{128}
1000	0.0002	0.0175	0.01	0.0452	0.01
5000	0.0007	0.0177	0.04	0.0350	0.02
10000	0.0013	0.0169	0.08	0.0404	0.03
50000	0.0051	0.0257	0.20	0.0381	0.13
100000	0.0128	0.0289	0.44	0.0448	0.28
500000	0.0634	0.0570	1.11	0.0764	0.83
1000000	0.1266	0.0802	1.58	0.1051	1.21

are more than 1.00 only for the cases with many (e.g., at least 500,000) circles. The single highest speedup is achieved with the Approach Circle-B on 16 CPUs with 16 threads.

Figure 5.12 shows speedup plots for the 16 CPU experiments. Here, Approach Circle-A usually had the highest speedup. The highest degree of parallelism was achieved when there is a sufficiently large number of circles and enough threads.

Table 5.46: Results Summary I, Circle-C, 16-CPU

# Circles	Serial Time	Time (16)	S_{16}	Time (32)	S_{32}
1000	0.0002	0.0033	0.08	0.0063	0.04
5000	0.0007	0.0036	0.18	0.0066	0.10
10000	0.0013	0.0049	0.27	0.0087	0.15
50000	0.0051	0.0070	0.73	0.0102	0.50
100000	0.0128	0.0111	1.15	0.0139	0.92
500000	0.0634	0.0444	1.43	0.0452	1.40
1000000	0.1266	0.0689	1.84	0.0801	1.58

Table 5.47: Results Summary II, Circle-C, 16-CPU

# Circles	Serial Time	Time (64)	S_{64}	Time (128)	S_{128}
1000	0.0002	0.0118	0.02	0.0283	0.01
5000	0.0007	0.0126	0.05	0.0260	0.03
10000	0.0013	0.0117	0.11	0.0259	0.05
50000	0.0051	0.0153	0.33	0.0281	0.18
100000	0.0128	0.0239	0.53	0.0305	0.42
500000	0.0634	0.0518	1.22	0.0717	0.88
1000000	0.1266	0.0871	1.45	0.0972	1.30

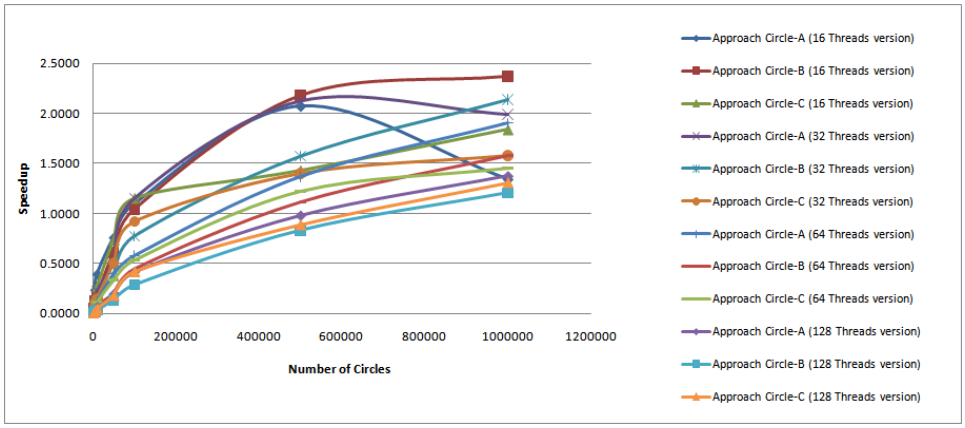


Figure 5.12: Speedup Plots on 16 CPUs for Circle-A, Circle-B and Circle-C

Table 5.48: Results Summary I, Circle-A, 24-CPU

# Circles	Serial Time	Time (24)	S_{24}	Time (48)	S_{48}
1000	0.0002	0.0039	0.06	0.0075	0.03
5000	0.0007	0.0041	0.16	0.0080	0.08
10000	0.0013	0.0040	0.33	0.0074	0.18
50000	0.0051	0.0071	0.72	0.0086	0.59
100000	0.0128	0.0136	0.94	0.0162	0.79
500000	0.0634	0.0341	1.86	0.0348	1.82
1000000	0.1266	0.0514	2.46	0.0468	2.71

Table 5.49: Results Summary II, Circle-A, 24-CPU

# Circles	Serial Time	Time (72)	S_{72}	Time (96)	S_{96}
1000	0.0002	0.0117	0.02	0.0154	0.02
5000	0.0007	0.0102	0.06	0.0142	0.05
10000	0.0013	0.0119	0.11	0.0156	0.09
50000	0.0051	0.0128	0.40	0.0150	0.34
100000	0.0128	0.0176	0.73	0.0228	0.56
500000	0.0634	0.0437	1.45	0.0505	1.26
1000000	0.1266	0.0611	2.07	0.0662	1.91

Tables 5.48 and 5.49 present the times and speedups for 24, 48, 72 and 96 threads on 24 CPUs versus serial processing time for Approach Circle-A for cases of 1000 to one million circles. Tables 5.50 and 5.51 compare the times and speedups for 24, 48, 72 and 96 threads on 24 CPUs versus serial processing time for Approach Circle-B for cases of 1000 to one million circles. The single highest speedup was achieved by Approach Circle-B on 24 CPUs with 24 threads.

Table 5.50: Results Summary I, Circle-B, 24-CPU

# Circles	Serial Time	Time (24)	S_{24}	Time (48)	S_{48}
1000	0.0002	0.0068	0.04	0.0014	0.17
5000	0.0007	0.0071	0.09	0.0015	0.45
10000	0.0013	0.0067	0.20	0.0024	0.56
50000	0.0051	0.0091	0.56	0.0070	0.73
100000	0.0128	0.0119	1.07	0.0156	0.82
500000	0.0634	0.0274	2.32	0.0431	1.47
1000000	0.1266	0.0446	2.84	0.0983	1.29

Table 5.51: Results Summary II, Circle-B, 24-CPU

# Circles	Serial Time	Time (72)	S_{72}	Time (96)	S_{96}
1000	0.0002	0.0199	0.01	0.0273	0.01
5000	0.0007	0.0266	0.02	0.0288	0.02
10000	0.0013	0.0221	0.06	0.0262	0.05
50000	0.0051	0.0246	0.21	0.0324	0.16
100000	0.0128	0.0260	0.49	0.0373	0.34
500000	0.0634	0.0521	1.21	0.0573	1.11
1000000	0.1266	0.0702	1.80	0.0903	1.40

Figure 5.13 shows speedup plots for the 24 CPU experiments. Here, Approach Circle-A was better than Circle-B for cases of 48 threads or more. The highest degree of parallelism was achieved when there was a sufficiently large number of circles and enough threads.

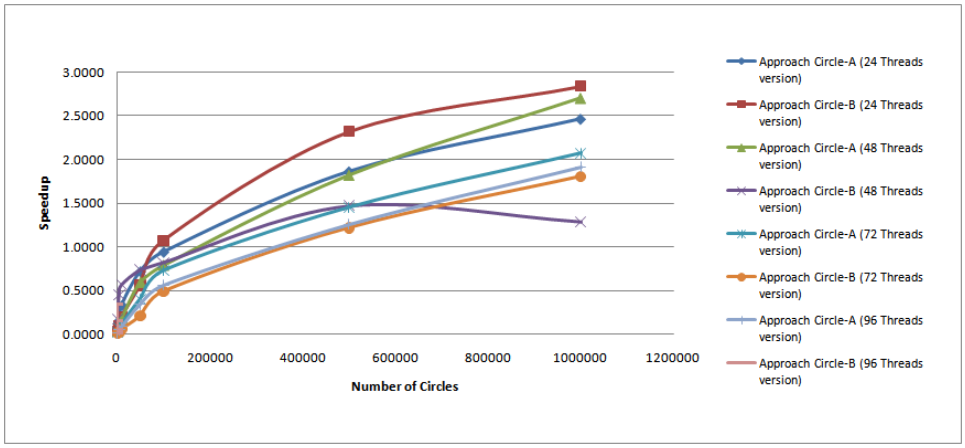


Figure 5.13: Speedup Plots for 24 CPUs for Approaches Circle-A and Circle-B

Table 5.52: Results Summary I, Circle-A, 32-CPU

# Circles	Serial Time	Time (32)	S_{32}	Time (64)	S_{64}
1000	0.0002	0.0049	0.05	0.0098	0.03
5000	0.0007	0.0045	0.15	0.0093	0.07
10000	0.0013	0.0050	0.26	0.0098	0.14
50000	0.0051	0.0076	0.67	0.0125	0.41
100000	0.0128	0.0146	0.88	0.0176	0.73
500000	0.0634	0.0337	1.88	0.0414	1.53
1000000	0.1266	0.0478	2.65	0.0428	2.96

Table 5.53: Results Summary II, Circle-A, 32-CPU

#Circles	Serial Time	Time (128)	S_{128}	Time (192)	S_{192}
1000	0.0002	0.0195	0.01	0.0304	0.01
5000	0.0007	0.0193	0.03	0.0299	0.02
10000	0.0013	0.0191	0.07	0.0282	0.05
50000	0.0051	0.0210	0.24	0.0381	0.13
100000	0.0128	0.0239	0.53	0.0387	0.33
500000	0.0634	0.0619	1.02	0.0730	0.87
1000000	0.1266	0.0650	1.95	0.1127	1.21

Tables 5.52 and 5.53 present the times and speedups for 32 CPUs for various number of threads for Approach Circle-A for cases of 1000 to one million circles. Tables 5.54 and 5.55 present the results for 32 CPUs for the same cases for Approach Circle-B. For these cases, increasing beyond 32 threads did not seen to improve results. Approach Circle-A was also usually faster. For 32 CPUs, the maximum speedup is achieved with Approach Circle-A with 64 threads. Figure 5.14 shows speedup plots of the results on 32 CPUs. Here, Approach Circle-A produces better

Table 5.54: Results Summary I, Circle-B, 32-CPU

# Circles	Serial Time	Time (32)	S_{32}	Time (64)	S_{64}
1000	0.0002	0.0089	0.03	0.0173	0.01
5000	0.0007	0.0092	0.07	0.0176	0.04
10000	0.0013	0.0084	0.16	0.0177	0.07
50000	0.0051	0.0103	0.50	0.0242	0.21
100000	0.0128	0.0138	0.92	0.0230	0.55
500000	0.0634	0.0283	2.24	0.0524	1.21
1000000	0.1266	0.0428	2.96	0.0722	1.75

Table 5.55: Results Summary II, Circle-B, 32-CPU

# Circles	Serial Time	Time (192)	S_{192}
1000	0.0002	0.0591	0.00
5000	0.0007	0.0548	0.01
10000	0.0013	0.0528	0.03
50000	0.0051	0.0676	0.08
100000	0.0128	0.0598	0.21
500000	0.0634	0.0943	0.67
1000000	0.1266	0.1214	1.04

speedups. The highest degree of parallelism is achieved when there is a sufficiently large number of circles and enough threads.

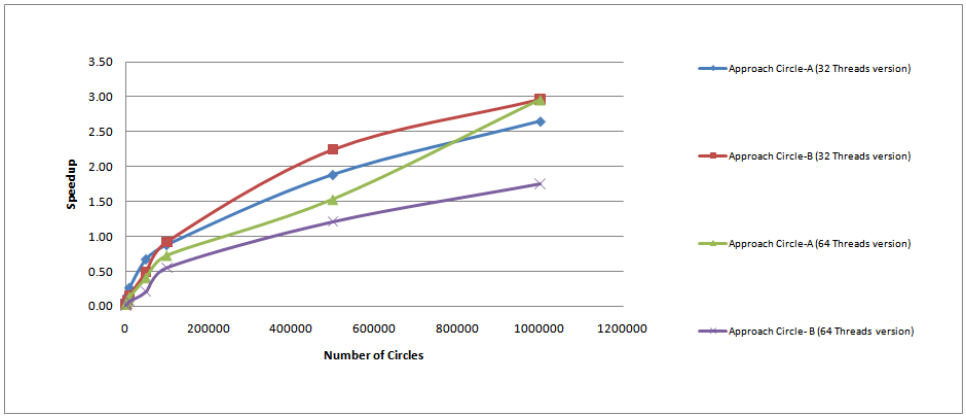


Figure 5.14: Speedup Plots on 32 CPUs for Circle-A and Circle-B

Table 5.56: Result Summary, Line Clipping on GPU

# Lines	Serial Time	T(A)	S(A)	T(B)	S(B)
1000	0.0002	0.0001	2.57	0.0001	1.92
5000	0.0010	0.0001	12.66	0.0001	10.23
10000	0.0019	0.0001	15.13	0.0002	11.20
50000	0.0099	0.0004	22.25	0.0005	18.69
100000	0.0215	0.0008	25.60	0.0010	20.99
500000	0.1243	0.0039	31.77	0.0047	26.27
1000000	0.2578	0.0076	34.06	0.0092	28.06

5.3 GPU Environment

Performance experiments of the GPU-based approaches are described next. The test environment was the NVidia Tesla System on the Alabama Supercomputer Center′s Dense Memory Cluster.

5.3.1 Cohen-Sutherland Line Clipping GPU Environment Results

First, the results for line clipping are shown. Table 5.56 shows the serial CPU and GPU execution times for both line clipping approaches. Speedups from use of the GPU are also shown. Figure 5.15 shows plots of the serial versus parallel (on GPU) execution time characteristic of the Cohen-Sutherland line clippings as the number of lines are increased from 1000 to 1,000,000.

Figure 5.16 shows speedup plots for Approaches A and B as the number of lines are increased from 1000 to 1,000,000. Approach A has larger speedup than Approach B for each number of lines. The speedup enhanced for Approach A from

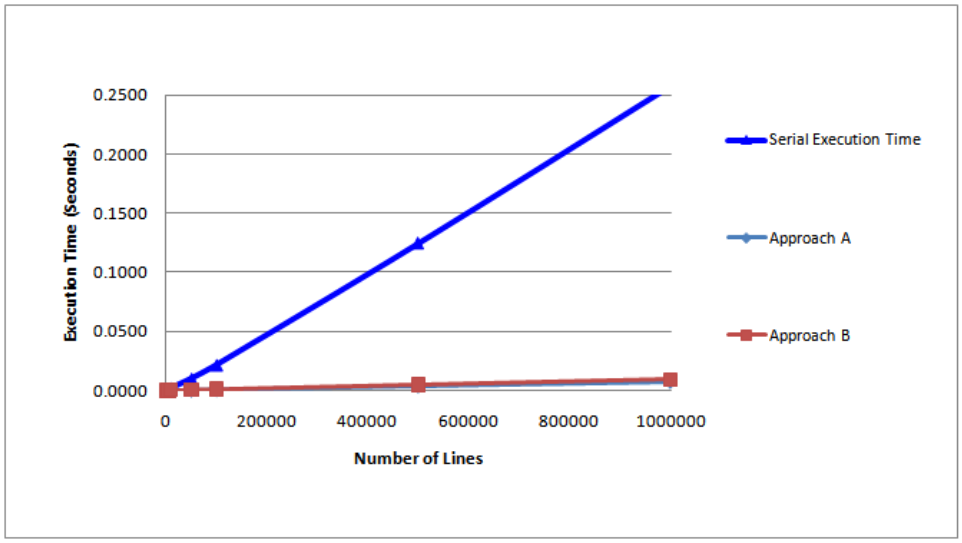


Figure 5.15: Execution Time on GPU for Line Clipping Approaches A and B

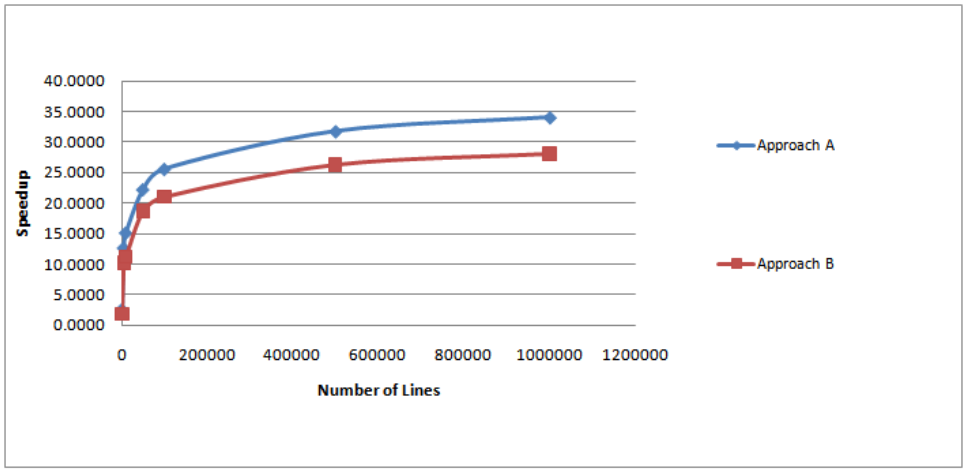


Figure 5.16: Speedup Plots for GPU for Approach A and B of Cohen-Sutherland Line Clipping

500,000 lines to 1,000,000 lines is 1.096. Using Amdahl′s law, this is equivalent to a fraction enhanced of 0.24. This limit in speedup is probably because almost twice the numbers of threads are created as in Approach B and, the thread create-destroy time may be a large component of the total time per task since the tasks are short. The memory bandwidth bottleneck between host CPU and GPU is also limiting the performance.

Table 5.57: Result Summary, Circle Clipping on GPU

# Circles	Serial Time	Circle-A	Speedup	Circle-B	Speedup
1000	0.0002	0.0001	3.94	0.0000	5.39
5000	0.0007	0.0001	10.03	0.0000	13.79
10000	0.0013	0.0001	17.21	0.0001	22.84
50000	0.0051	0.0001	38.44	0.0001	42.25
100000	0.0128	0.0002	69.36	0.0002	68.98
500000	0.0634	0.0007	87.23	0.0008	80.58
1000000	0.1266	0.0013	94.72	0.0015	83.70

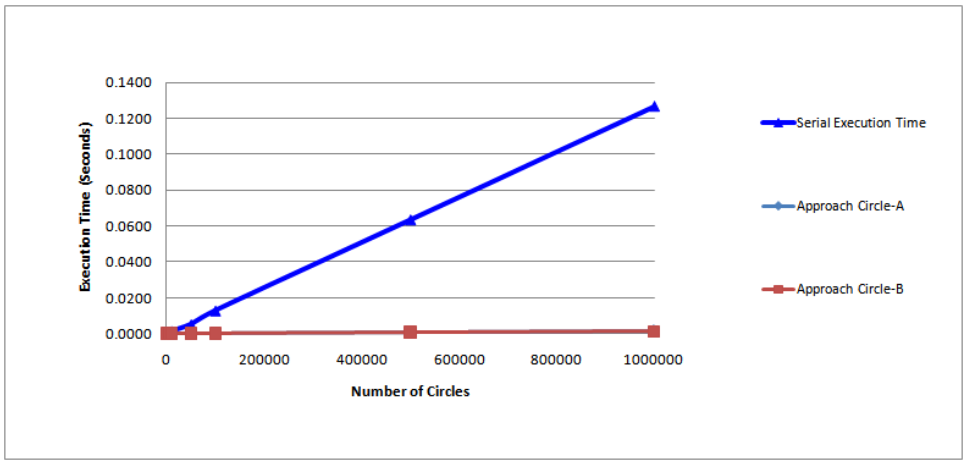


Figure 5.17: Execution Time on GPU for Fast Circle Clipping Approaches Circle-A and Circle-B

5.3.2 Fast Circle Clipping GPU Environment Results

This section describes and analyzes the results for circle clipping on the GPU.

Table 5.57 shows the serial CPU and parallel GPU execution times for both Fast Circle Clipping Approach Circle-A and Circle-B. Speedups from use of the GPU are also shown. Figure 5.17 shows plots of the serial versus parallel (on GPU) execution time characteristic of the Fast Circle Clipping approaches as the number of

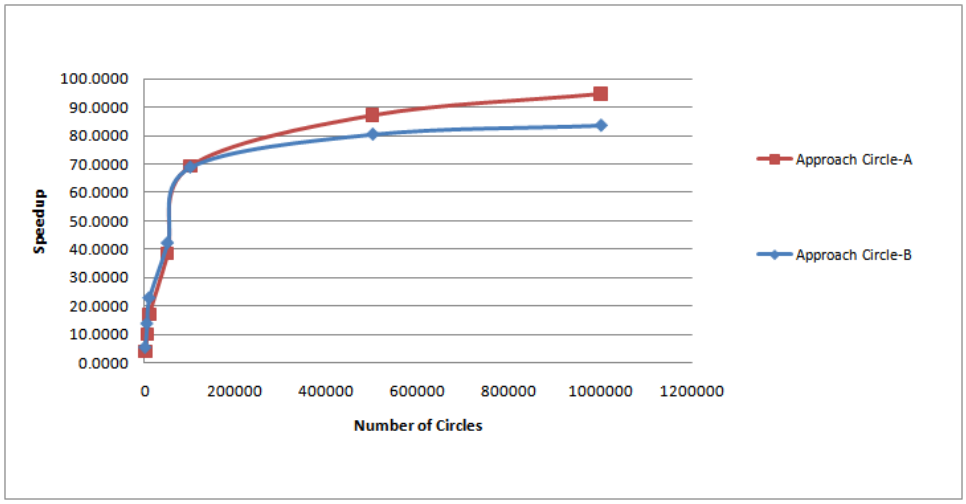


Figure 5.18: Speedup Plots for GPU, Circle-A and Circle-B

circles increases from 1,000 to 1,000,000. One interesting feature of the performance is that as circles increases by factor of 1000 (from 1000 to 1,000,000), time increases from close to 0.02 seconds to close to 0.13 seconds, which is less than 10 times.

Figure 5.18 shows speedup plots for the approaches as the number of circles are increased from 1000 to 1,000,000. As number of circles increases, speedups also increase. This difference in speedup increase for the same data set is probably because there is much thread creation-destruction overhead for both approaches. A′s speedup is higher, probably because Approach B creates and destroys almost twice the number of threads as that of Approach A, so B has even more thread creation overhead. Approach B creates almost twice the number of threads as that of Approach A.

5.3.2.1 Experiments on Group Size in Circle Clipping

We also considered the effect of the group size on performance. The group size is the number of circles being processed in one thread. Table 5.58 to Table 5.65 show the serial processing time on the CPU versus the parallel execution times on the GPU for Approaches Circle-A and Circle-B of the Fast Circle Clipping for 5000 to 1,000,000 circles, as the group size increases from 1 to 40 circles per group. 50 circles per group was also tested for the 500,000 circles cases. However, for Approach Circle-A, the 500,000 and 1,000,000 circles cases show improved performance as the group size increases. The improvement is probably because the percent of total processing time spent on thread(s) creation-destruction is relatively high in Approach Circle-B due to twice the number of threads created in Circle-B than in Circle-A. The memory bandwidth between the host memory and GPU memory is limited. This limits the rate of data transfer from host memory to GPU memory. This is the bottleneck for the performance of the GPU which makes it behave more serially. It limits the speedup to be achieved from GPU over CPU. The GPU also is not able to execute more than a few hundred threads simultaneously, so beyond a certain point, the creation of additional threads also doesn't expose more work simultaneously, that can actually be done parallel. Hence, Approach Circle-B does not seem to benefit from increasing group size as much as Approach Circle-A does.

Table 5.58: Result Summary I (msec), Group Size, Circle-A on GPU (1000 and 5000 circles)

# Circles	Serial Time	Group Size	# Threads	T (A)	Speedup
1000	0.2480	1	1000	0.0590	4.20
1000	0.2480	2	500	0.0660	3.76
1000	0.2480	5	200	0.0560	4.43
1000	0.2480	10	100	0.1040	2.38
1000	0.2480	20	50	0.0750	3.31
1000	0.2480	40	25	0.0850	2.92
5000	0.6620	1	5000	0.0670	9.88
5000	0.6620	2	2500	0.0700	9.46
5000	0.6620	5	1000	0.0880	7.52
5000	0.6620	10	500	0.1020	6.49
5000	0.6620	20	250	0.1190	5.56
5000	0.6620	40	125	0.1470	4.50

Table 5.59: Result Summary I (msec), Group Size, Circle-B on GPU (1000 and 5000 circles)

# Circles	Serial Time	Group Size	# Threads	Circle-B	Speedup
1000	0.2480	1	1000	0.0750	3.31
1000	0.2480	2	500	0.0850	2.92
1000	0.2480	5	200	0.0104	23.85
1000	0.2480	10	100	0.0126	19.68
1000	0.2480	20	50	0.0145	17.10
1000	0.2480	40	25	0.0137	18.10
5000	0.6620	1	5000	0.0710	9.32
5000	0.6620	2	2500	0.0810	8.17
5000	0.6620	5	1000	0.1090	6.07
5000	0.6620	10	500	0.1400	4.73
5000	0.6620	20	250	0.1950	3.39
5000	0.6620	40	125	0.1210	5.47

Table 5.60: Result Summary I (msec), Group Size, Circle-A on GPU (10,000 and 50,000 circles)

# Circles	Serial Time	Group Size	# Threads	Circle-A	Speedup
10000	1.3250	1	10000	0.0630	21.03
10000	1.3250	2	5000	0.0670	19.78
10000	1.3250	5	2000	0.0850	15.59
10000	1.3250	10	1000	0.1020	12.99
10000	1.3250	20	500	0.1130	11.73
10000	1.3250	40	250	0.1410	9.40
50000	5.1120	1	50000	0.1240	41.23
50000	5.1120	2	25000	0.1200	42.60
50000	5.1120	5	10000	0.1210	42.25
50000	5.1120	10	5000	0.1310	39.02
50000	5.1120	20	2500	0.1410	36.26
50000	5.1120	40	1250	0.1770	28.88

Table 5.61: Result Summary I (msec), Group Size, Circle-B on GPU (10,000 and 50,000 circles)

# Circles	Serial Time	Group Size	# Threads	Circle-B	Speedup
10000	1.3250	1	10000	0.0630	21.03
10000	1.3250	2	5000	0.0760	17.43
10000	1.3250	5	2000	0.0340	38.97
10000	1.3250	10	1000	0.1290	10.27
10000	1.3250	20	500	0.1920	6.90
10000	1.3250	40	250	0.3300	4.02
50000	5.1120	1	50000	0.1220	41.90
50000	5.1120	2	25000	0.1130	45.24
50000	5.1120	5	10000	0.1270	40.25
50000	5.1120	10	5000	0.1490	34.31
50000	5.1120	20	2500	0.2190	23.34
50000	5.1120	40	1250	0.3510	14.56

Table 5.62: Result Summary I (msec), Group Size, Circle-A on GPU (100,000 and 500,000 circles)

# Circles	Serial Time	Group Size	# Threads	Circle-A	Speedup
100000	12.7620	1	100000	0.1990	64.13
100000	12.7620	2	50000	0.1930	66.12
100000	12.7620	5	20000	0.1890	67.52
100000	12.7620	10	10000	0.1530	83.41
100000	12.7620	20	5000	0.1490	85.65
100000	12.7620	40	2500	0.1750	72.93
500000	63.4140	1	500000	0.7870	80.58
500000	63.4140	2	250000	0.6990	90.72
500000	63.4140	5	100000	0.5620	112.84
500000	63.4140	10	50000	0.4260	148.86
500000	63.4140	20	25000	0.3560	178.13
500000	63.4140	40	12500	0.2940	215.69
500000	63.4140	50	10000	0.2850	222.51

Table 5.63: Result Summary I (msec), Group Size, Circle-B on GPU (100,000 and 500,000 circles)

# Circles	Serial Time	Group Size	# Threads	Circle-B	Speedup
100000	12.7620	1	100000	0.1750	72.93
100000	12.7620	2	50000	0.1950	65.45
100000	12.7620	5	20000	0.1800	70.90
100000	12.7620	10	10000	0.1970	64.78
100000	12.7620	20	5000	0.2360	54.08
100000	12.7620	40	2500	0.3830	33.32
500000	63.4140	1	500000	0.6470	98.01
500000	63.4140	2	250000	0.6740	94.09
500000	63.4140	5	100000	0.6320	100.34
500000	63.4140	10	50000	0.6660	95.22
500000	63.4140	20	25000	0.6500	97.56
500000	63.4140	40	12500	0.6770	93.67
500000	63.4140	50	10000	0.7000	90.59

Table 5.64: Result Summary I (msec), Group Size, Circle-A on GPU (1,000,000 circles)

# Circles	Serial Time	Group Size	# Threads	Circle-A	Speedup
1000000	126.6430	1	1000000	1.5130	83.70
1000000	126.6430	2	500000	1.3050	97.04
1000000	126.6430	5	200000	1.0240	123.67
1000000	126.6430	10	100000	0.7660	165.33
1000000	126.6430	20	50000	0.6270	201.98
1000000	126.6430	40	25000	0.5340	237.16

Table 5.65: Result Summary I (msec), Group Size, Circle-B on GPU (1,000,000 circles)

# Circles	Serial Time	Group Size	# Threads	Circle-B	Speedup
1000000	126.6430	1	1000000	1.2010	105.45
1000000	126.6430	2	500000	1.2640	100.19
1000000	126.6430	5	200000	1.1770	107.60
1000000	126.6430	10	100000	1.1740	107.87
1000000	126.6430	20	50000	1.2640	100.19
1000000	126.6430	40	25000	1.2280	103.13

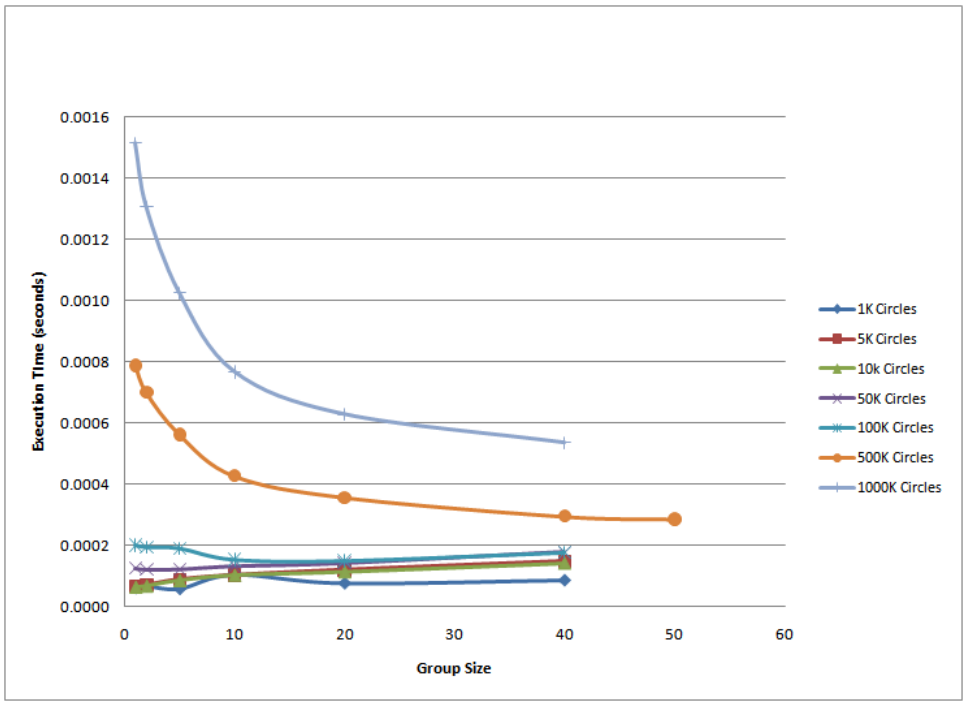


Figure 5.19: Execution Time on GPU for Circle-A by Group Size

For 1,000,000 circles for Circle-A, the speedup increases from 83 for group size 1 to 237 for group size 40. Similarly, for the case of 500,000 circles for Circle-A, the speedup increases from 90 for group size 2 to 222 for group size 40. Figure 5.19 shows a plot of execution time for Approach Circle-A. Figure 5.20 shows a speedup plot for it. The speedup remains almost uniform for group sizes 20, 30 and 40, for cases of 1000, 5000, 10000, 50000, and 100,000 circles for Approach Circle-A.

Figure 5.21 shows a plot of execution times for approach Circle-B. The increase in the parallel execution time as group size increases is obvious here. Specifically, the speedup has decreased from 45 for group size 2 to 14 for group size 40 for case 50,000 circles for Approach Circle-B.

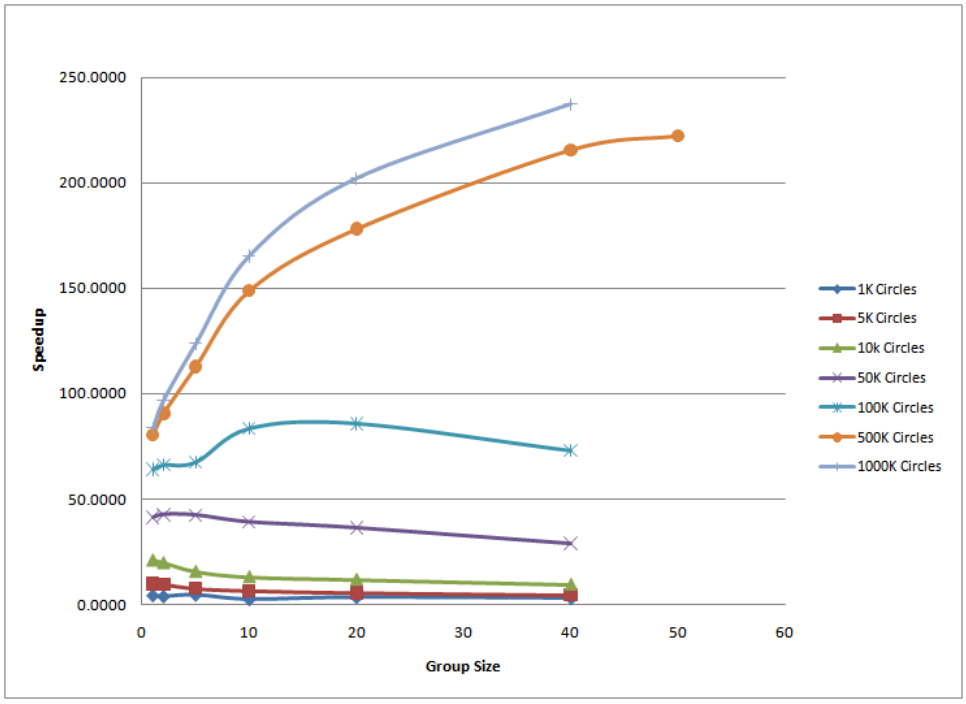


Figure 5.20: Speedup Plots for GPU, for Circle-A by Group Size

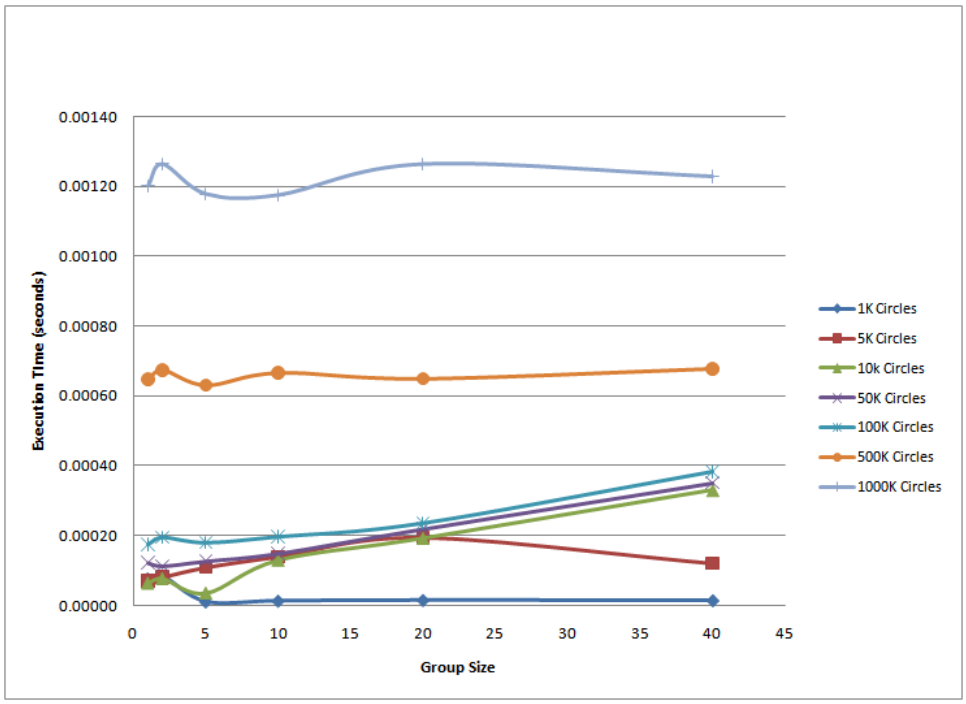


Figure 5.21: Execution Time on GPU for Circle-B by Group Size

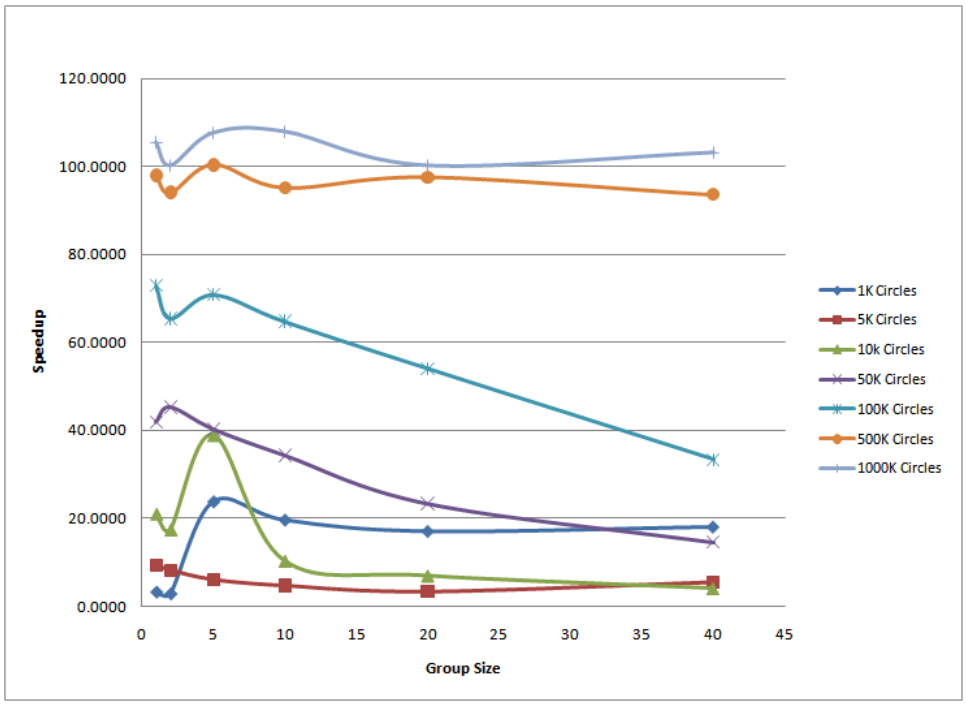


Figure 5.22: Speedup Plots for GPU, for Circle-B by Group Size

For the POSIX version, the range of number of CPUs tested for Cohen-Sutherland line clipping and Fast Circle Clipping are from 1, 2, 4, 8, 16, and 32 CPU. The range of number of threads tested are 4, 8, 16, 24, 32, 48, 64, 72, 96, 128, and 192 threads. Approach A of line clipping and Approach Circle-A of circle clipping are the best approaches in terms of speedup.

For the GPU version Approach A of line clipping and Approach Circle-A of circle clipping with no group clipping are best approaches in terms of speedup. But for circle clipping, Approach Circle-A with group clipping is the best in terms of speedup.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

Clipping is a graphical process to find portions of scene inside a specified region of space. Line clipping discards those parts of the lines which are outside of the clipping region and allowing rendering of the parts which are inside. The two parallel approaches are developed for Cohen-Sutherland line clipping. Circle clipping is a procedure to accept only those parts of the given circle that are inside of the clipping region. The Fast Circle Clipping algorithm also has four parallel approaches. These approaches are implemented on multi-threaded cluster computer and GPU environments.

The parallel Approach A of the Cohen Sutherland line clipping processes each line independently in parallel. Every line (or group a group of lines) is completely processed by one thread, performing outcode computation, trivial accept, trivial reject and if necessary, the intersection calculations with the clipping region. The highest level of speedup achieved with Approach A on the multi-threaded cluster computer is 11.91 using 32 CPU with 32 threads for 1,000,000 lines. The highest degree of speedup achieved on GPU is 34.06 for 1,000,000 lines. Approach B, a set of k threads

performs the outcode computation for one line (or a group of lines). After this step, another set of k threads performs the task of trivial reject, trivial accept, and if necessary, the intersection calculations with the clipping region. The highest level of speedup achieved from Approach B on multi-threaded cluster computer on 24 CPU with 24 threads is 4.16 for 1,000,000 lines. The highest degree of speedup achieved by Approach B on GPU is 28.06 for 1,000,000 lines.

Approach Circle-A processes each circle (or group of circles) independently in parallel. One thread processes one circle (or a group of circles) completely, performing trivial accept, trivial reject and if necessary, intersection calculations with clipping region. The highest level of speedup achieved with Approach Circle-A on the multi-threaded cluster computer is 2.96 using 32 CPU with 64 threads for 1,000,000 circles. The highest degree of speedup achieved on GPU is 94.72 for 1,000,000 circles. Approach Circle-B processes each circle (or group of circles) independently in parallel. A set of k threads performs the task of trivial reject, trivial accept, and if necessary, the intersection calculations with the clipping region. The highest level of speedup achieved with Approach Circle-B on the multi-threaded cluster computer is 2.96 using 32 CPU with 32 threads for 1,000,000 circles. The highest degree of speedup achieved on GPU is 83.70 for 1,000,000 circles. Approach Circle-A, group clipping has highest degree of speedup of 237.16 on GPU for 1,000,000 circles. Approach Circle-B, group clipping has highest degree of speedup of 103.13 on GPU for 1,000,000 circles.

Hence, Approach A of line clipping and Approach Circle-A of circle clipping are more efficient than other approaches.

6.2 Future Work

For future work, the new parallel approaches of Cohen-Sutherland line and Circle clipping presented in this thesis can be tested on consumer GPUs. They could also be tested on real scenes. The same techniques of the approaches of line and circle clipping can also likely be applied to other popular line and circle clipping techniques. The new parallel approaches can also be developed for other line and circle clipping.

REFERENCES

[1] Donald Hearn and M. Pauline Baker. *Computer Graphics with OpenGL*. Pearson Education, Inc., Upper Saddle River, NJ, 3rd edition, 2004.

[2] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll and Steven Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. on Graphics.*, 25(3):485–493, July 2006.

[3] NVidia. NVidia CUDA Programming Guide. Developer's Guide / Manual Version 3.0, NVidia Inc., NVIDIA, Santa Clara, CA, February 2010.

[4] John D. Owens, Mike Houston, David Luebke, Simon Green, John Stone and James Phillips. Graphics processing units - powerful, programmable, and highly parallel - are increasingly targeting general-purpose computing applications. In *Proceedings of IEEE*, volume 96(5), pages 879–899, April 2008.

[5] Randima Fernando and Mark Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Pearson Education, Inc., Upper Saddle River, NJ, 1st edition, 2003.

[6] E. Lindholm, J. Nickolls, S. Oberman and J. Montrym. NVidia Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, May 2008.

[7] William Stalling. *Operating Systems: Internals and Design Principles*. Pearson Education, Inc., Upper Saddle River, NJ, 6th edition, 2008.

[8] NVidia. NVidia's next generation CUDA compute architecture fermi. Whitepaper Version 1.0, NVidia Inc., NVidia, Santa Clara, CA, September 2009.

[9] Daniel Weiskopf. *GPU-Based Interactive Visualization Techniques*. Springer, Inc., NY, 1st edition, 2006.

[10] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Pearson Education, Inc., Upper Saddle River, NJ, 2nd edition, 2004.

[11] IEEE Computer Society. Standard for information technologyportable operating system interface (posix), application program interface api), input/output (i/o), x/open system interface (xsi). Technical Report IEEE Std 1003.1-2008, Portable applications standards committee and the open group, IEEE, NY, December 2008.

[12] Erich Strohmaier. Japan reclaims top ranking on latest top500 list of world's supercomputer. top500 supercomputer sites (www.top500.org), July 2011.

[13] Alabama Supercomputer Authority. Alabama supercomputer. Technical Report 9th Edition, Alabama supercomputer center, Huntsville, AL, December 2010.

[14] John Nickolls, William J. Dally. The GPU computing era. *IEEE Micro*, 30(2):56–69, April 2010.

[15] Raman Srinivasan. *A Fast Circle Clipping Algorithm in Graphics Gems III.* ed. by David Kirk, Morgan Kaufman, Inc., NY, 1992.