REAL-TIME 3D INTEGRAL IMAGING SYSTEM USING A FASTER ELEMENTAL IMAGE GENERATION METHOD USING GPU PARALLEL PROCESSING

Md. Sifatul Islam	**13241001**
Mahfuze Subhani	**13201035**
Mohd. Zishan Tareque	**13201010**
M. Rashidur Rahman Rafi	**13201008**

Supervisor: Dr. M. Ashraful Alam

Department of Computer Science and Engineering

BRAC University

DECLARATION

We hereby declare that this report is our own work and effort and that is has not been submitted anywhere for any award. All the contents provided here is totally based on our own labor dedicated for the completion of the thesis. Where other sources of information have been used, they have been acknowledged and the sources of information have been provided in the reference section.

Signature of Supervisor:

____

Dr. M. Ashraful Alam
Assistant Professor
Department of Computer Science and Engineering
BRAC University

Signature of Authors:

____

Md.Sifatul Islam

____

Mahfuze Subhani

____

Mohd. Zishan Tareque

____

M. Rashidur Rahman Rafi

ACKNOWLEDGEMENT

First of all, we would like to express our deepest sense of gratitude to almighty Allah. Secondly, we would like to express our sincerest gratitude to our advisor Dr. M. Ashraful Alam for his continuous support, patience, motivation, and immense knowledge in our research. His guidance helped us in all parts of the progress.

Furthermore, we would like to personally thank Md. Shahinur Alam for his help in conducting the experiment from Optical Information Processing Lab, School of Information and Communication Engineering, Chungbuk National University, Korea.

Finally, we would like to express our sincere gratefulness to our beloved parents, brothers and sisters for their love and care. We are grateful to all of our friends who helped us directly or indirectly to complete our thesis.

Table of Contents

List of figures

List of tables

ABSTRACT

A novel method of faster computation of Elemental Image generation for real time integral imaging 3D display system, with the implementation of GPU parallel processing is proposed. Previous experiments were conducted to generate Real Time Integral Image and resulting frame rate was greater than 30fps. Our proposed system consists of the following steps: information acquisition of real objects in real time, calculation of lens property, generation of elemental image sets using pixel mapping algorithm and GPU parallel processing for faster generation. To implement this system, firstly the color (RGB) and depth information data of each object point is acquired from the depth camera (Kinect sensors). Using these information, we create the elemental image sets using pixel mapping algorithm. And finally using a wrapper/sdk of CUDA (CUDAfy) we implemented the pixel mapping algorithm in GPU and hence the overall computational speed of the real time integral display system increases. The proposed system provides elemental images generated at a rate of more than 65 fps. Furthermore, this opens up a new field of possibilities of improvement in integral imaging such as real time projection in multi direction and many others.

Keywords: real-time, integral imaging, depth and color information, multi-directional elemental image sets, GPU parallel processing, pixel mapping algorithm, acquisition, frame.

Chapter 1: Introduction

1.1 Introduction

Lippmann first introduced integral imaging (II) in 1908[1]. II is a promising autostereoscopic [2] and multiscopic [3] three-dimensional (3D) imaging technique. It was known as "integral photography" where all the small parts of an image are combined to form one complete image. It captures and reproduces a light field by using a two-dimensional (2D) array of microlenses, sometimes called a fly's-eye lens. Each microlens acquires an image of the subject viewed from the location of the lens during capture period. During projection period, each microlens permits the observer's eye to view only the area of the corresponding micro-image of the certain part of the object which is visible from position of observer's eye. In a stereoscopic 3D display we needed to use special glasses, which is being replaced by autostereoscopic technology. Autostereoscopic technology is the technology of displaying 3D view without the use of any special wearing devices or glasses [1-3]. II is one of the better alternatives to use instead of other modern 3D display technologies such as a hologram or volumetric 3D display [7]. The hologram is able to reconstruct an object without any interference between depth cues, because it can provide all four visual components such as binocular disparity, motion parallax, accommodation, and convergence. However, coherent light source is needed in order for it to work properly and accurately whereas no coherent light source is needed for integral imaging its current techniques enable the display of full natural color dynamic 3D images. Furthermore, hologram is calculation intensive and it requires to be in a well strict lighted environment (i.e. it cannot work in daylight) which is not necessarily required for II. Volumetric 3D displays provide both physiological and psychological cues to the human visual system to perceive 3D objects, and they are considered a powerful and desirable device for human/computer visual interface. But the 3D output result is oriented mechanically which will prove inefficient to be used in real time as computation time will be slower while II has no such limitations. This system is now becoming more popular among researchers as these are the pillars for future technological advancement. An integral imaging system consists of a pickup and display section. In the pickup section, an object is imaged through a lens array. Each elemental lens forms a corresponding image of the object, and the elemental images were initially captured by a charge-coupled device (CCD) camera. However, some disadvantages arose in obtaining elemental images from real objects in a conventional optical method due to the setup of a lens array and the correct placement of a CCD camera and prevention of unnecessary beams entering the lens array [1-6]. In order to overcome these drawbacks, a new pickup method to extract 3D information such as color and depth data from a real object using a depth camera to generate elemental images based on the depth and color data of object pixels was introduced [4]. The pickup method just needed a depth camera, PC, and LCD monitor. But this was not enough to produce a real-time II display as the computation of this method was not fast enough due to intensive calculation but it would be possible if there were any ways to speed up the computation. Then it became possible when a method was proposed to enable real time based on graphics parallel processing using a depth camera [8]. In that system, elemental image arrays (EIAs) were generated using a quad-core central processing unit (CPU) and graphics processing unit (GPU) with 256 parallel processor cores. Therefore, this method proved to be much faster in generating EIAs at a frame rate enough to be real time (around 30 frame per second). For this method, the computation process is carried out by the host program for setting the environment and the OpenCL kernel for pickup processing which is processor dependent. Despite this progress, the speed of the computation was still not fast enough for real time applications. Furthermore, there were still some problems left unresolved such as narrow viewing angle, poor resolution and shallow image depth that are restricted by the specifications of the lens array used such as lens pitch, focal length, etc. Due to these problems, commercial use of this 3D display system was on halt. The viewing angle, the expressible depth range, and the resolution of the 3D images are closely related to one another, and one of them should be sacrificed to improve the others [9-10]. Real-time capture and visualization of 3D information of real world object is one of the most promising issues in 3D display and imaging fields. However, real-time 3D information technology has been suffering from intensive calculation complexity and the cost of special optical devices. It is very important to extract 3D information from 3D objects and to display it in real-time for the 3D broadcasting and 3D interaction technology. Many systems have been developed to alleviate the narrow viewing zone problem such as the lens switching method[11], the curved lens array technique[12], arrays with low fill factor[13]. Micro-convex-mirror arrays [14], multiple axis telecentric relay system [15] and two display devices and a lens array [16]. But still they were not enough for commercial implementation. Some other effective methods have been introduced to tackle this problem. Choi et al. showed a multiple-viewing zone II display in which multiple viewing zones were produced by guiding the light rays coming out of the elemental images with the usage of a dynamic barrier array [17]. Unfortunately, this system was not feasible for real-time operation because of the dependency of mechanical movement and low speed. Later Baasantseren et al. proposed a wide viewing angle II display using two EI masks [18]. Shin et al. also demonstrated an II display with viewing direction control using 4-f optical relay and a dynamic aperture to control diverging ray directions [19]. However, due to the inefficient and unsuitable optical setup this method was not optimum for practical implementation. So then M. A. Alam et al. proposed an II display system capable of controlling the viewing zone with directional projection of EIs at a predefined projection angle at a time[20] which did not require any barrier array to control the orientation of the ray coming out from each EI. Shortly after M. A. Alam et al. further enhanced this system by implementing a time-multiplexed two-directional sequential projection scheme by implementing Directional Elemental Image Generation and Resizing (DEIGR) algorithm[21] which allows up to two predefined projection angles at a time by switching between the screens of two projectors. D. Fattal et al. also proposed a multi-viewing 3D display that can project the correct perspectives of a 3D image in many spatial directions simultaneously [1-4]. They do not require special glasses or eye tracking and provide a 3D stereoscopic experience to many viewers at the same time with full motion parallax and it is passive i.e. it does not require special illumination of the scene and so it can operate with regular incoherent daylight. They introduced a multi-directional diffractive backlight technology that permits the rendering of high-resolution, full-parallax 3D images in a very wide view zone. In both of these methods, the elemental image needs to be generated at a fast speed in order for them to be implemented on a commercial scale. Therefore even faster computation of EIs are required.

Hence, we propose a faster computation of EI generation in 3D Integral imaging systems using NVIDIA GPU parallel processing by implementing with a CUDA wrapper called CUDAfy. According to the experiments conducted, optical and hardware setup and theoretical formulas implemented using the data sets we generated, it is expected to achieve a frame rate of approximately 118 frames per second at an average for generating EIs. So in overall taking in account the acquisition time, the frame rate will be at an average of 17 frames per second which may aid in getting closer for commercial usage of other novel methods of II stated.

Chapter 2: Integral Imaging

2.1 Integral Image Pickup Method

When Integral Image(II) was first proposed by Lippmann in 1908[1] the Elemental Images (EI) were acquired or captured and recorded on a 2D image sensor such as charge coupled device by tracing the light rays emerging from a 3D object, using lens array consisting of number of convex lenses termed as elemental lens. Basically, this is an acquisition process of the optical information of a 3D object, in which the directional and intensity information of the 3D object are spatially sampled by an array of refractive optical elements such as lens array.

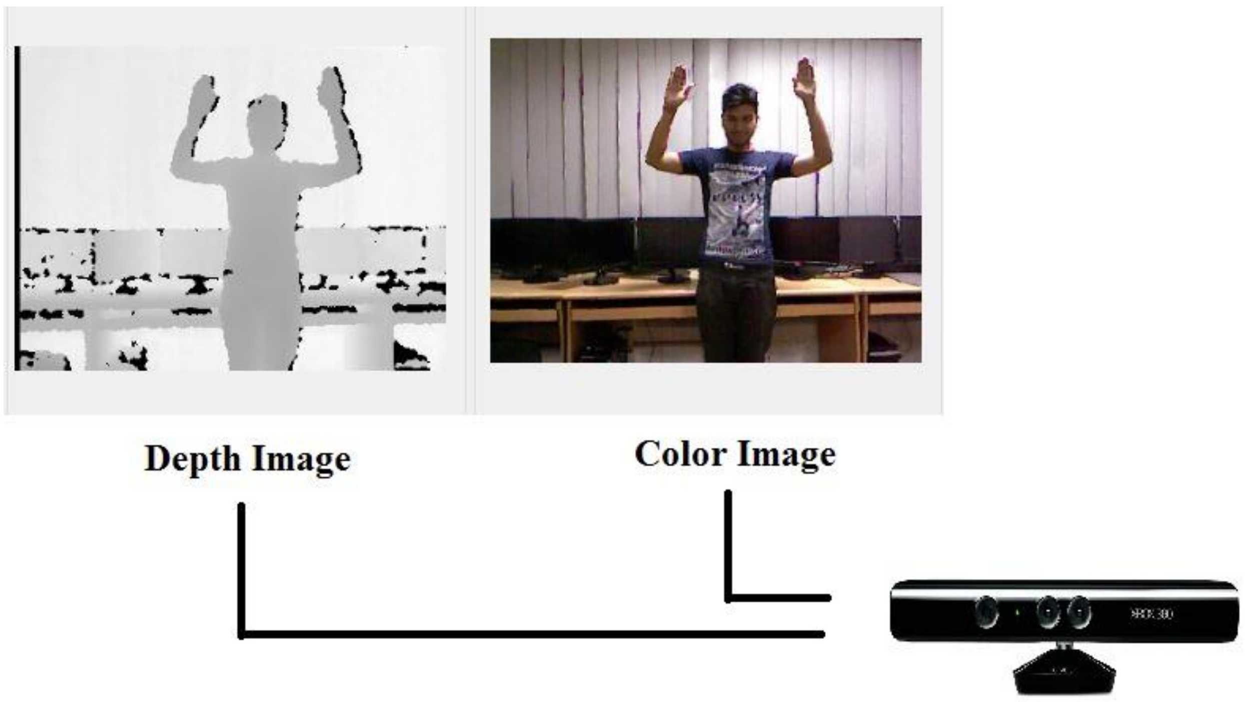


Figure 1: Pickup method using a depth camera

In our proposed method, we are going to acquire the optical information of the 3D object i.e. the depth values and the color (RGBA) information using a depth camera such as Kinect sensor or any other type of depth camera. Using these information, multiple sets of EI are generated through further calculations. First of all, the parameters of the lens array and display panel should be stored in the program. The depth camera (i.e. Kinect Sensor) will acquire the depth data of every pixel on the 3D object surface as shown in the figure below.

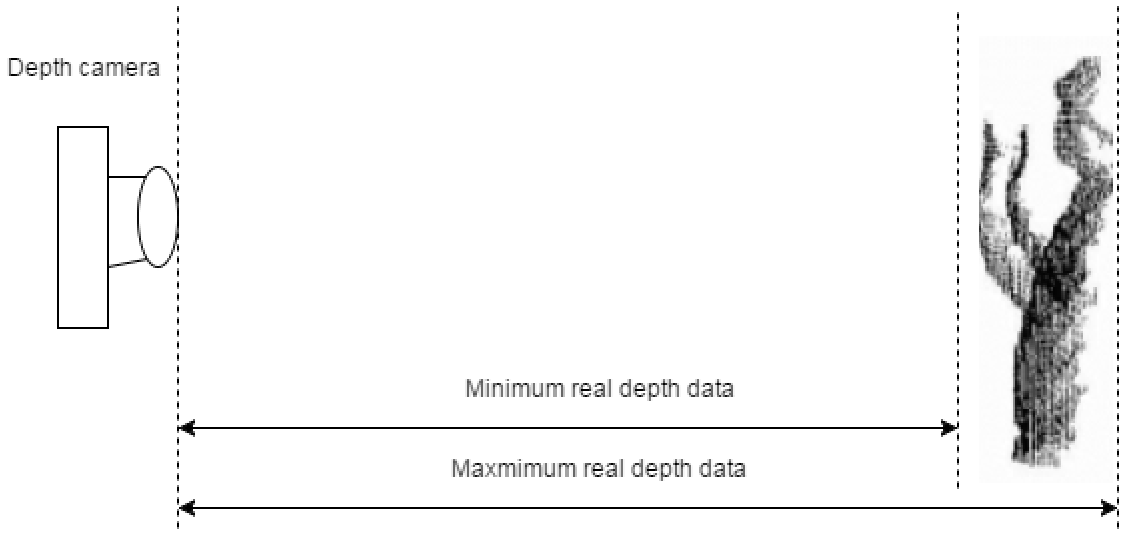
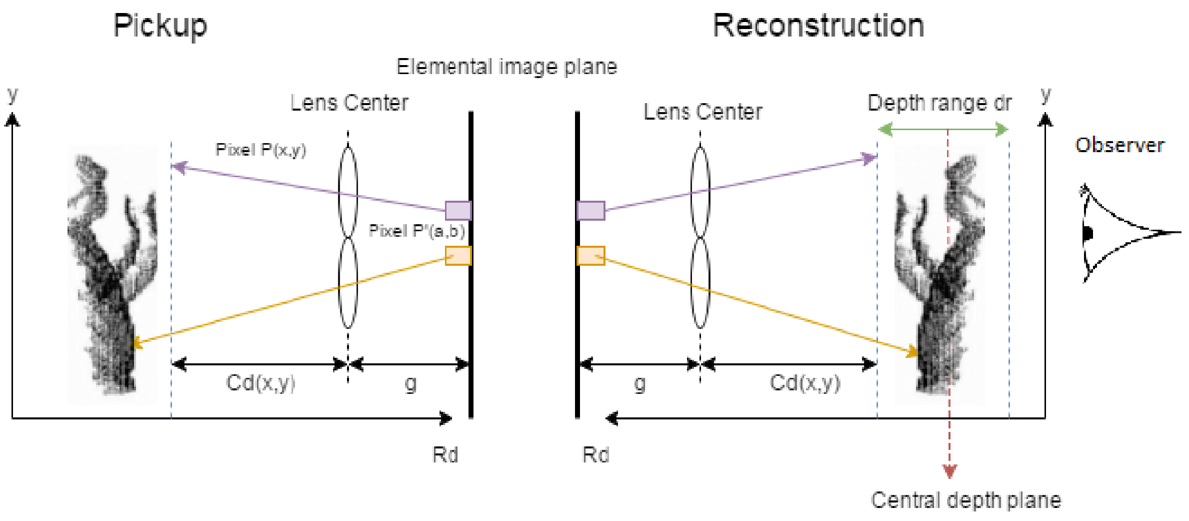
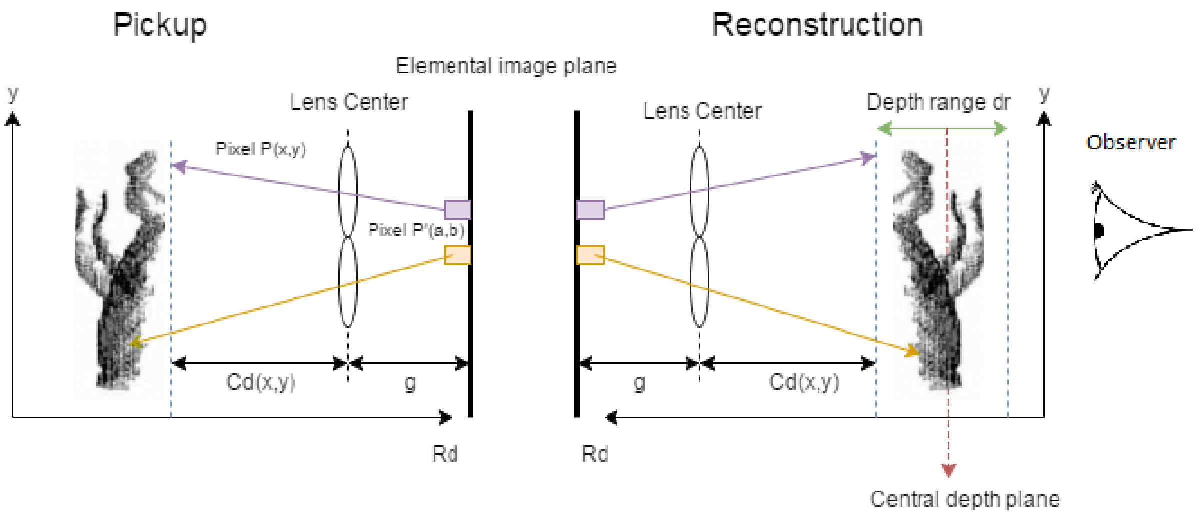


Figure 2: Depth data distribution from 3d object using depth camera

The actual distance from object surface point to the depth camera is the depth data of the object. But the acquired object depth information must be converted and there are two reasons for this. First of all, we need to convert from a pseudoscopic image to an orthoscopic image [22-23]. The recorded elemental images are back-projected through a lens array. So, if we use the original depth data to reconstruct integral images directly, the output image will be reversed. The second reason is due to the inherent limitations of the depth range of an integral imaging technique [24-25].



(a)



(b)

Figure 3: Geometry of integral image reconstruction using (a) original and (b) converted depth data.

Usually, the real depth data acquired by the depth camera will exceed the expressible depth range of the II method. The depth range dr of the integral imaging method is expressed as

$$dr = \frac{2.d.Ip}{Lp} \tag{1}$$

Where Lp is the pitch of the elemental lens, Ip is the pixel size of the object image, and d is the central depth of the integral imaging system. The variables d and Ip can be expressed by

$$d = \frac{f.g}{f+g} \tag{2}$$

$$Ip = \frac{d.Dp}{g} \tag{3}$$

where f is the focal length of the elemental lens, g is the gap between the LCD display and the lens array, and Dp is the pixel pitch of the LCD monitor. In order to display all the pixels accurately, the reconstructed pixels must be in the range from
d-dr/2 to d+dr/2. This implies that all of the converted depth data should be located in this range. Thus, the converted distance Cd(x,y) is expressed as

$$Cd(x,y) = \frac{d.(\max(Rd)+\min(Rd))}{Rd(x,y).2} \tag{4}$$

Where Rd(x,y) is the real depth of (x,y)-th pixel.

2.2 Processing

For the calculation stage of elemental images (EI) and generation, three buffers need to be created. The first buffer stores the information for every object pixel. The center coordinates of the elemental lenses are reserved for the second buffer. Finally, the third buffer is used to store the calculated EI pixel set. The coordinates of the elemental lenses center are computed based on the elemental lens indices and the lens pitch.

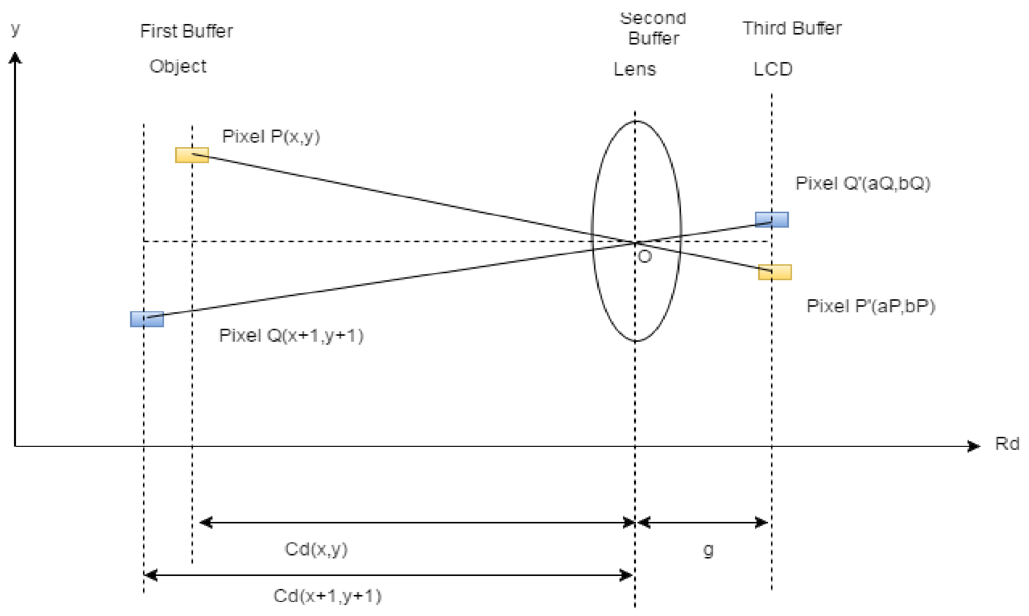


Figure 4: Geometry of pixels mapping from object image plane to elemental image plane for generating elemental image pixels

The above figure shows how the geometry of pixel mapping from object pixels to the elemental image plane through an elemental lens. The ray of the object pixel A (i,j) will be located at the positions of elemental image plane A'(a,b) as it passes through the center of the lens. The pixel coordinate (a,b) is given by

$$a = Lp.xL - (x.Ip - Lp.xL).\frac{g}{Cd(x,y)} \qquad (5)$$

$$b = Lp.yL - (y.Ip - Lp.yL).\frac{g}{Cd(x,y)} \qquad (6)$$

Where i and j are the object pixel indices in the horizontal and vertical axes, respectively and xL and yL are the indices of the lenses on both axes. Therefore by combining equation (5) and (6) we can show that every depth data Cd(x,y) calculates its own elemental image in every pixel position.

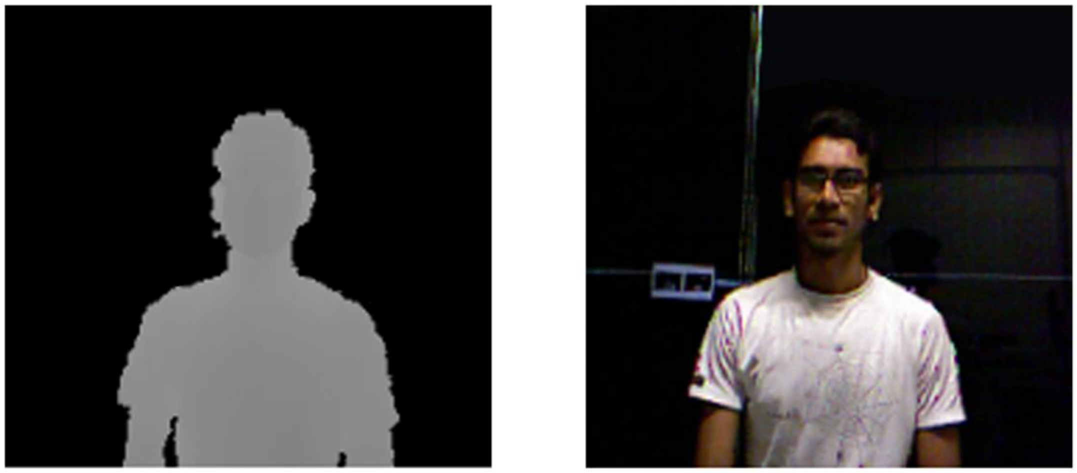


Figure 5: Object color image and its corresponding depth image

Figure 5 shows the color image and corresponding depth data of the object during the acquisition process. Figure 6 shows the elemental images generated using this method.

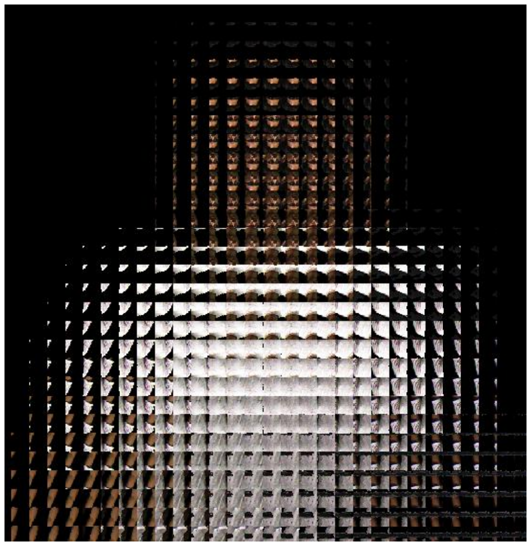
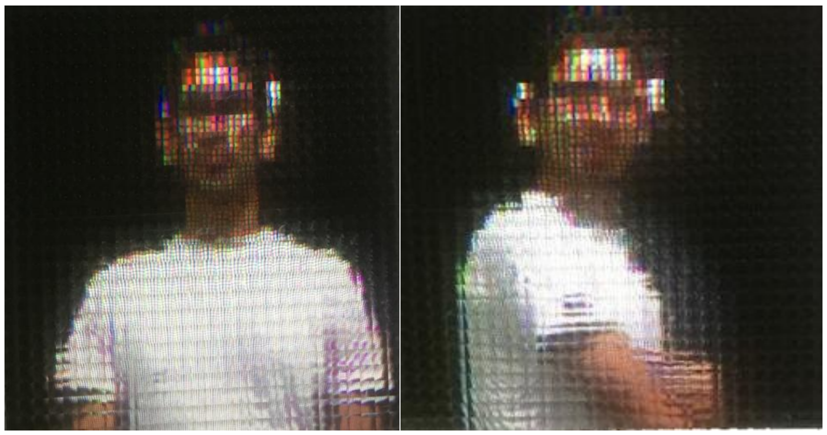


Figure 6: Elemental image generated

2.3 Reconstruction/Display

After the elemental image is accurately calculated by using the color and depth data acquired from the depth camera, this image is displayed through a lens array with the correct position placement. Figure 7(a) shows the final integral image acquired and viewed from a lens array. Figure 7(b) shows the integral image acquired from a video showing the generation of elemental image being viewed through a lens array.



(a) (b)

Figure 7: (a) Integral image acquired from lens array (b) Visualization of Integral Image

Chapter 3: GPU

3.1 Background Information of GPU

A graphics processing unit (GPU), is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. Modern GPUs are very efficient at manipulating computer graphics and image processing, and their highly parallel structure makes them more efficient than general-purpose CPUs for algorithms where the processing of large blocks of data is done in parallel. The term GPU was popularized by Nvidia in 1999, who marketed the **GeForce 256** as "the world's first GPU", or Graphics Processing Unit. It was presented as a "single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second". GPU-accelerated computing offloads compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. From a user's perspective, applications simply run much faster. GPUs compute much faster than CPUs because GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously unlike that of CPUs who have consists of a few cores optimized for sequential serial processing.

3.2 GPU Architecture

There are couple of contrasts amongst GPU and CPU processor design. NVIDIA's GPU comprise of various streaming multiprocessor (SMs) and each streaming multiprocessor comprises of numerous scalar processor also called as cores. NVIDIA came up with different GPU architectures such as Kepler, Fermi Tesla etc.

3.2.1 Tesla Architecture

Tesla is Nvidia's first microarchitecture implementing the unified shader model. The driver supports Direct3D 10 Shader Model 4.0 / OpenGL 2.1(later drivers have OpenGL 3.3 support) architecture. The design is a major shift for NVIDIA in GPU functionality and capability, the most obvious change being the move from the separate functional units (pixel shaders, vertex shaders) within previous GPUs to a homogeneous collection of universal floating point processors (called "stream processors") that can perform a more universal set of tasks. This GPU was the first GPU which has unified shader with 128 processing elements which distributed in 8 shader core.

3.2.2 Fermi Architecture

Fermi is the codename for a GPU microarchitecture developed by Nvidia as the successor to the Tesla microarchitecture. The Fermi architecture is one of the most significant step forward towards GPU architecture since the Original G80. A whole new approach was followed to create the World's First Computational GPU. NVIDIA gathered extensive user feedback on GPU computing since the introduction of G80 and GT200 and made improvements on Fermi architecture on various areas such as Double Precision Performance, True Cache Hierarchy, ECC support, Faster Context Switching, Faster Atomic Operations and More Shared Memory.

The first Fermi based GPU, implemented with 3.0 billion transistors, features up to 512 CUDA cores. A CUDA core executes a floating point or integer instructions per clock for a thread. The 512 CUDA cores are organized in 16 SMs of 32 cores each. The GPU has six 64-bit memory partitions for a 384-bit memory interface, supporting up to a total of 6GB of GDDR5 DRAM memory. A host interface connects the GPU to the CPU via PCI-express. The Giga Thread global scheduler distributes thread blocks to SM thread schedulers.

3.2.3 Kepler Architecture

Kepler GPU microarchitecture was presented by NVIDIA after Fermi. This engineering was centered on power effectiveness. GeForce arrangement from 600 to 700 and some of 800 arrangement utilized Kepler design and all are manufactured in 28nm. Later on Kepler was supplanted by Maxwell engineering. In this GPU microarchitecture NVIDIA designer concentrated on effectively utilization of force and furthermore programmability and execution in the interim engineering before this was concentrating on expanding execution. To be exact, two Kepler cores use around 90% of force of a Fermi cores whereas unified GPU clock diminish half power utilization.

Kepler based members add the following standard features:

- PCI express 3.0 interface
- Display Port 1.2
- CUDA compute capability from 3.0 to 3.5
- Dynamic Parallelism

3.2.4 Maxwell Architecture

Maxwell is the codename for a GPU microarchitecture developed by NVIDIA as the successor to the Kepler microarchitecture. These new chips provided few consumer-facing additional features, as Nvidia instead focused more on GPU power efficiency. The L2 cache was increased from 256 KiB on Kepler to 2 MiB on Maxwell, reducing the need for more memory bandwidth. Accordingly, the memory bus was reduced from 192 bit on Kepler to 128 bit, further saving power. The streaming multiprocessor design from Kepler was also redesigned and partitioned, renaming it to SMM for Maxwell. The structure of the warp scheduler was taken from Kepler, with the texture units and FP64 CUDA cores still shared, but the layout of most execution units were divided so that each warp schedulers in an SMM controls one set of 32 FP32 CUDA cores, one set of 8 load/store units and one set of 8 special function units. This is in contrast to Kepler, where each SMX has four schedulers that schedule to a shared pool of execution units

3.2.5 Pascal Architecture

Pascal is the codename for a GPU microarchitecture developed by NVIDIA as the successor to the Maxwell microarchitecture. The Pascal microarchitecture was introduced April 2016 with the GP100 chip. The architecture name is derived from Blaise Pascal, the 17th century mathematician. In Pascal, an SM (streaming multiprocessor) consists of 64 CUDA cores. Maxwell had 128, Kepler 192, Fermi 32 and Tesla only 8 CUDA cores into an SM; the GP100 SM is divided into two processing blocks, each having 32 single-precision CUDA Cores, an instruction buffer, a warp scheduler, 2 texture mapping units and 2 dispatch units. It supports CUDA Compute Capability 6.0.

3.3 CUDA

CUDA was introduced in 2007 by NVIDIA to enable programming general purpose computation on parallel GPU architectures. CUDA’s full form is Compute Unified Device Architecture which is NVIDIA GPU architecture that is in GPU card. It has positioned itself as a whole new meaning for general purpose computing with GPUs. CUDA uses extension of C++ known as CUDA C for programming purpose. CUDA provides advantage of huge computational power to the programmer and it is famous among them since it gives a lot of freedom to work on. CUDA has many co-operating cores depending on the GPU model. Here cores can communicate and also they can exchange information with each other so that, running multithreaded application there is no need for streaming computing in GPU.As previously mentioned CUDA uses C programming language and main idea of CUDA is that GPU consist of thousands of thread that can execute in parallel and all those thread can execute same function or code. Here all the threads are executed using same code but different data. CUDA programs consist of one or two parts that is executed either on host (CPU) or device (GPU). When there is little parallelism involved then it is better to execute the code CPU since copying data to host to device takes time but when the parallelism is huge then it is better to execute that in GPU because it overcomes copy speed and provides performance boost.

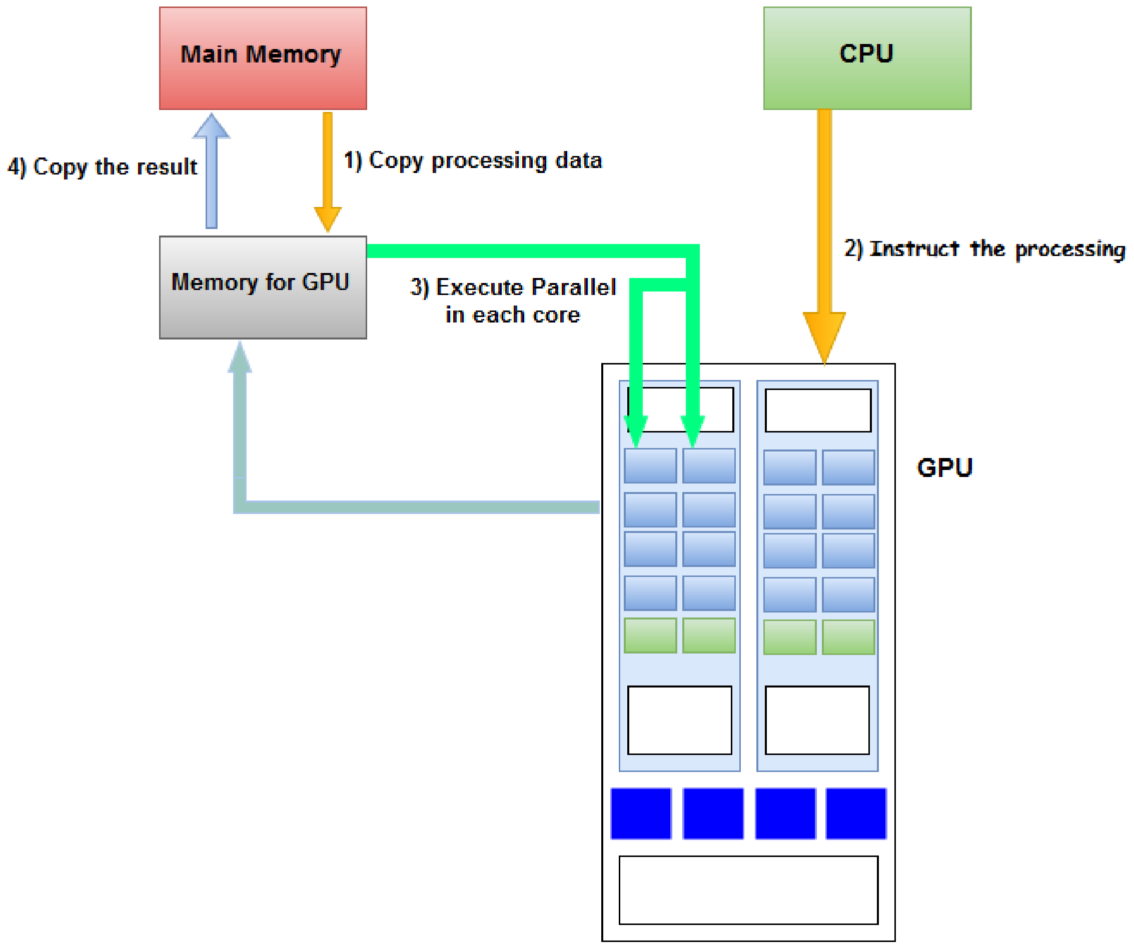


Figure 8: Processing Flow on CUDA

<u>3.3.1 Units of CUDA</u>

Grids

A grid is group of threads all running in the same kernel. Grids can't be shared between GPUs. There's no synchronization at all between the blocks within a grid. Grids are executed in GPUs where an entire grid is handled by a single GPU.

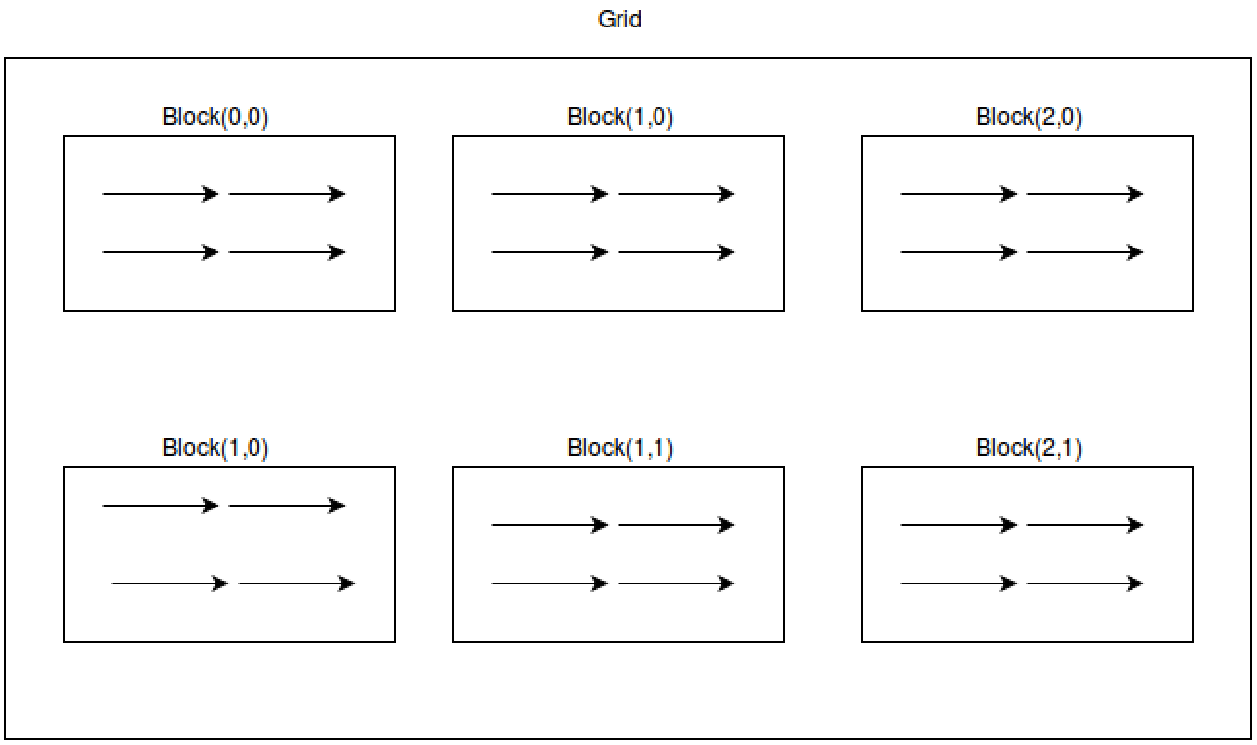


Figure 9: 1D Grid with 2D Blocks and Threads

Blocks

Grid is composed of multiple blocks. Each block is composed of multiple threads which could execute concurrently or serially and in no particular order but all the threads uses the same program in a block. Blocks have unique ids and it can be interpreted as 1D, 2D or 3D as well depending on the architecture. Blocks are executed in Multiprocessors(MPs).The

GPU chip is organized as a collection of multiprocessors (MPs), with each multiprocessor responsible for handling one or more blocks in a grid. A block is never divided across multiple MPs.

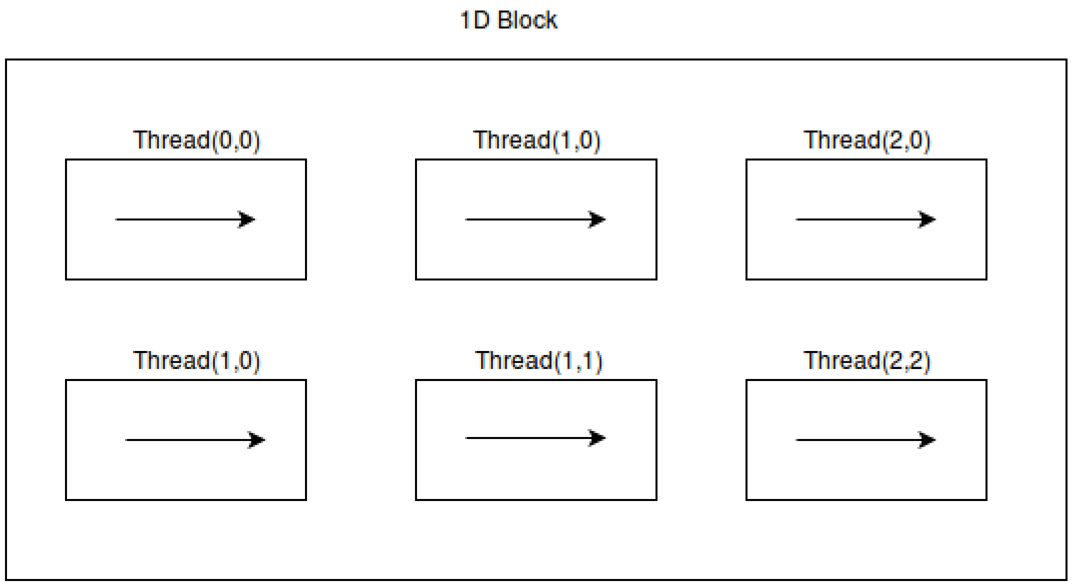


Figure 10: 1D Block with 2D Threads (3x2)

Threads

Each block consists of threads. This is just an execution of a kernel with a given index. Each thread uses its index to access elements in array such that the collection of all threads cooperatively processes the entire data set. Threads are run on individual cores of microprocessors but they are not restricted to a single core. Thread has its unique ID and threads can be interpreted as 1D, 2D or 3D which depends on block dimension. Threads are executed in stream processors (SPs). Each Multi Processor (MP) is further divided into a number of Stream Processors (SPs), with each SP handling one or more threads in a block. If a GPU device has, for example, 4 multiprocessing units, and they can run 768 threads each: then at a given moment no more than 4*768 threads will be really running in parallel (if we planned more threads, they will be waiting their turn).

CUDA built-in variables

- **blockIdx.x, blockIdx.y, blockIdx.z** are built-in variables that returns the block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code.
- **threadIdx.x, threadIdx.y, threadIdx.z** are built-in variables that return the thread ID in the x-axis, y-axis, and z-axis of the thread that is being executed by this stream processor in this particular block.
- **blockDim.x, blockDim.y, blockDim.z** are built-in variables that return the "block dimension" (i.e., the number of threads in a block in the x-axis, y-axis, and z-axis)

Thread identification and manipulation

In order to make the most out of the features of CUDA, we need to know how to manipulate the blocks and threads in order to carry out intense calculations in the most efficient manner.

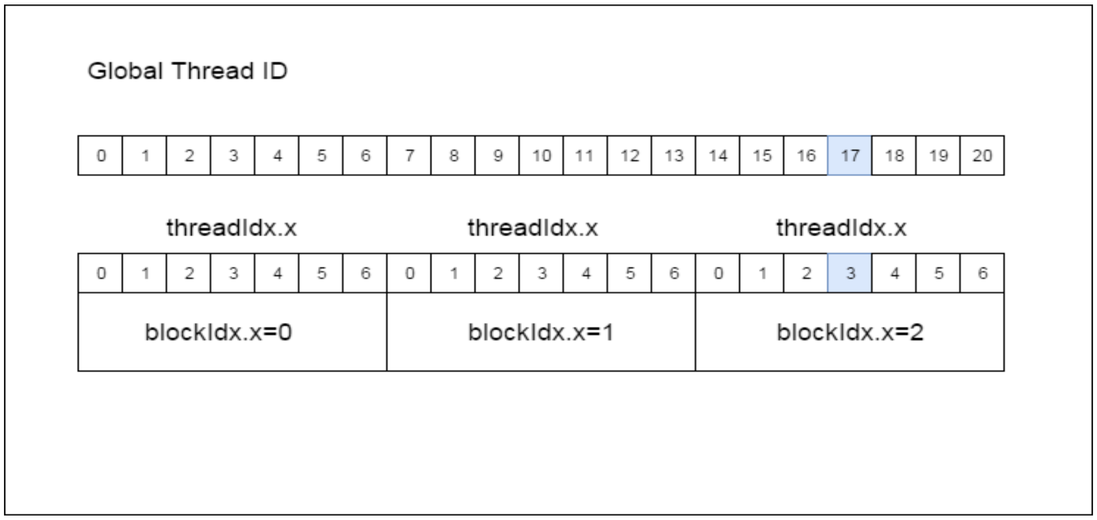


Figure 11: Global Thread ID generation from 1D block

Assume a hypothetical 1D grid and 1D block architecture: 3 blocks, each with 6 threads. So for example, if we want to calculate the global thread ID of 26:

- gridDim.x = 3 x 1
- blockDim.x = 7 x 1
- Global Thread ID = (blockIdx.x * blockDim.x) + threadIdx.x
- Global Thread ID = (2 x 7) + 3 = 17

3.3.2 Memory units in CUDA

Global memory: This memory is built from a bank of SDRAM chips connected to the GPU chip. Any thread in any MP can read or write to any location in the global memory. Sometimes this is called *device memory*.
Texture cache: This is a memory within each MP that can be filled with data from the global memory so it acts like a cache. Threads running in the MP are restricted to read-only access of this memory.
Constant cache: This is a read-only memory within each MP.
Shared memory: This is a small memory within each MP that can be read/written by any thread in a block assigned to that MP.
Registers: Each MP has a number of registers that are shared between its SPs.

Figure 1 shows different types of memory available: global memory, texture memory,s and shared memory. Global memory is device memory visible to every thread in the same compute grid with large size; Shared memory is visible to threads in the same compute block, and is very fast to access, but much smaller capacity than global memory.

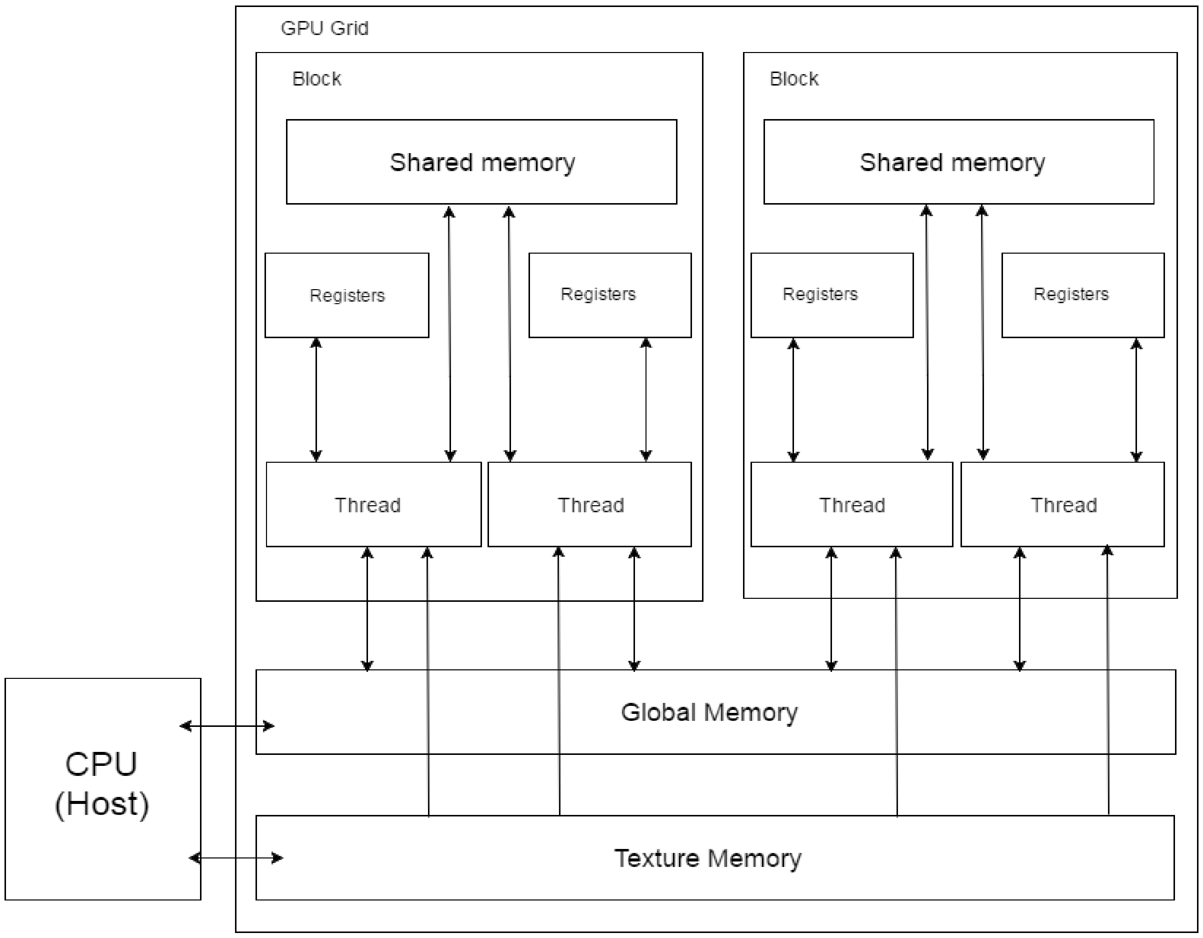


Figure 12: Memory model of CUDA

3.3.3 CUDAfy

CUDAfy .NET allows easy development of high performance GPGPU applications completely from the Microsoft .NET framework. It's developed in C#. Modern graphics cards enable the potential of massive speed increase over CPUs for non-graphics related intensive numeric operations. Many large data set operations such as matrices can see a 75x or more speed up. CUDAfy allows .NET developers to easily create complex applications that split processing without any interference between host and GPU. There are no separate CUDA cu files or complex set-up procedures to launch GPU device functions. It follows the CUDA programming model and any knowledge gained from tutorials or books on CUDA can be easily transferred to CUDAfy, only in a clean .NET fashion. CUDAfy supports both CUDA and OpenCL code generation and therefore has the ability to run the same applications on:

- NVIDIA GPUs (CUDA or OpenCL)
- AMD GPUs (OpenCL)
- Intel CPUs (OpenCL)

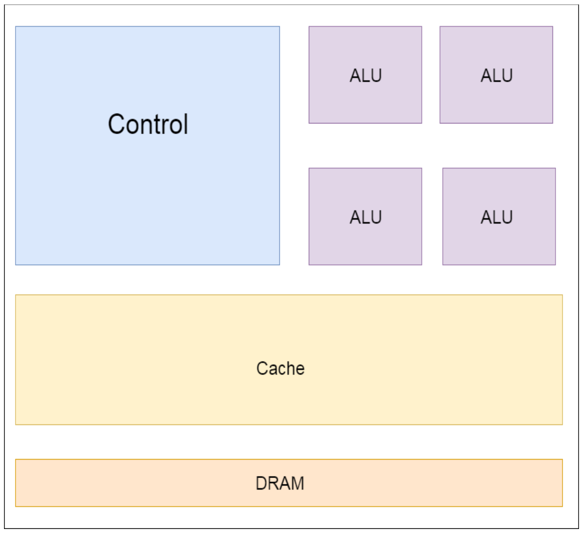
3.4 Difference between CPU and GPU Processing

The CPU (central processing unit) is often regarded the brains of the PC. But now, that brain is being enhanced by another part of the PC – the GPU (graphics processing unit), which is its main component. All PCs have chips that render the display images to monitors. But not all these chips are created equal. Intel's integrated graphics controller provides basic graphics that can display only productivity applications like Microsoft PowerPoint, low-resolution video and basic games. The GPU is in a class by itself – it goes far beyond basic graphics controller functions, and is a programmable and powerful computational device in its own right. With regards to the architecture, the CPU is composed of just few cores with lots of cache memory that can handle a few software threads at a time. In contrast, a GPU is composed of hundreds of cores that can handle thousands of threads simultaneously. The ability of a GPU with 100+ cores to process thousands of threads can accelerate some software by 100x over a CPU alone. What's more, the GPU achieves this acceleration while being more power- and cost-efficient than a CPU. The following table x shows some of the basic differences between a CPU and GPU.

CPU	GPU
CPU is the brain of the computer where all the programs instructions are executed and stored.	A GPU is meant to alleviate the load of the CPU by handling all the advanced computations necessary to project the final display on the monitor.
Really fast caches (great for data reuse)	Lots of math units
Lots of different processes/threads	Run a program on each fragment/vertex
High performance on executing a single thread	High performance on parallel tasks or execution
CPU optimized for high performance on sequential codes	GPU optimized for higher arithmetic intensity for parallel nature

Table 1: Differences between CPU and GPU

Figure 13 and 14 shows the basic architecture of CPU and GPU respectively.



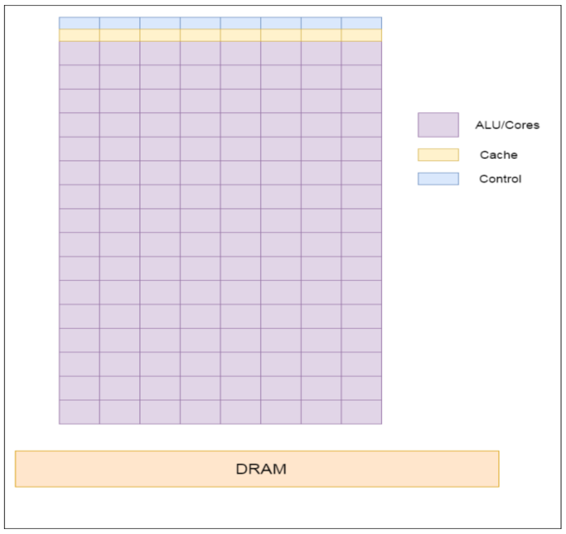


Figure 13

Figure 14

Chapter 4: Proposed Method

4.1 PRINCIPLE OF PROPOSED METHOD

We utilized the architecture and concurrent operation features of the GPU along with the algorithm for acquisition of the input data (i.e. Depth and color) and generation of the EI's to produce EI's at a faster rate. In order to do this, we first have to analyze on which part of the code we can implement our GPU code on. We must use the GPU code on areas where there are large amount of calculations (usually in a loop). The figure 15(a) shows a flowchart of our proposed method.

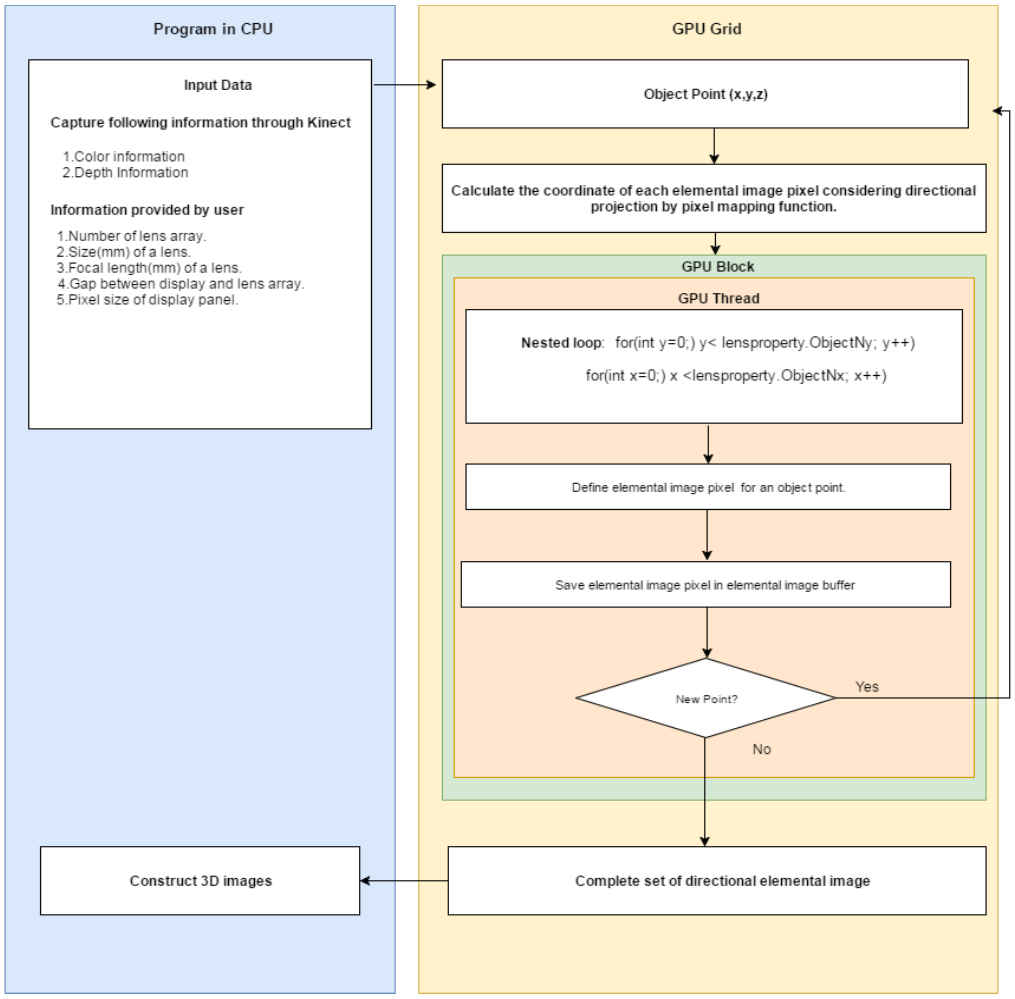

Program in CPU
Input Data
Capture following information through Kinect
1.Color information
2.Depth Information
Information provided by user
1.Number of lens array.
2.Size(mm) of a lens.
3.Focal length(mm) of a lens.
4.Gap between display and lens array.
5.Pixel size of display panel.
Construct 3D images
GPU Grid
Object Point (x,y,z)
Calculate the coordinate of each elemental image pixel considering directional projection by pixel mapping function.
GPU Block
GPU Thread
Nested loop: for(int y=0;) y< lensproperty.ObjectNy; y++)
for(int x=0;) x <lensproperty.ObjectNx; x++)
Define elemental image pixel for an object point.
Save elemental image pixel in elemental image buffer
New Point?
Yes
No
Complete set of directional elemental image

Figure 15.1: Flowchart of the proposed method

Figure 15.2 shows the visual implementation of our proposed method.

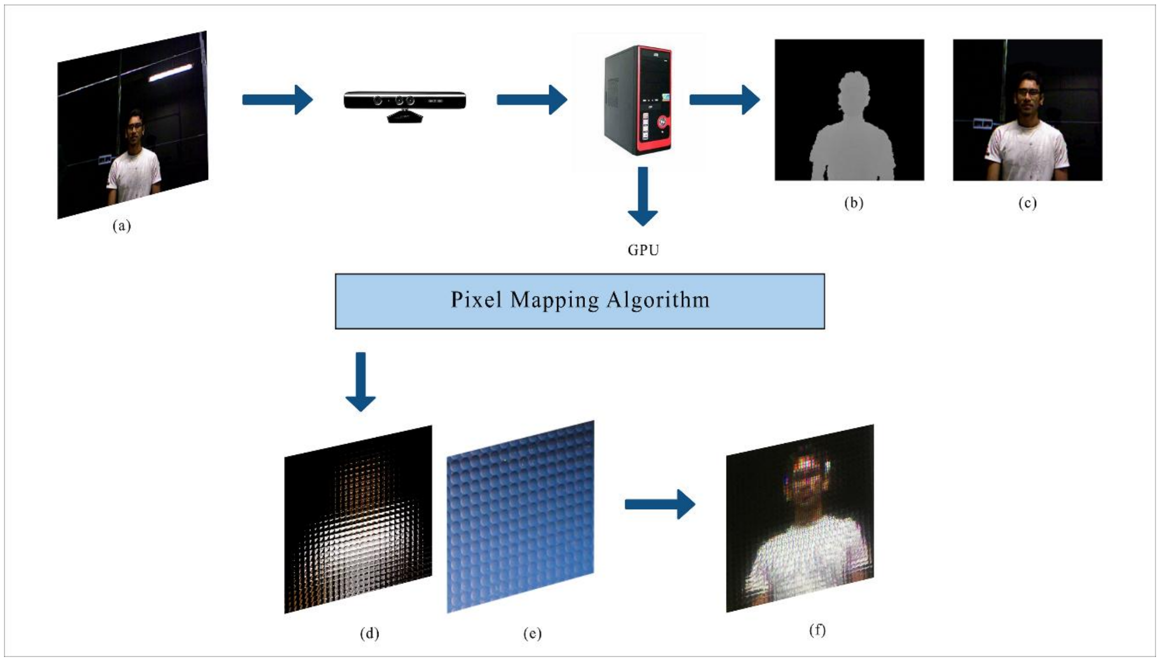


Figure 15.2: Visual implementation of our proposed method (a) Object point (b) Depth Image (c) Color Image (d) Elemental Image (e) Lens array (f) Integral image

For acquisition, we first obtain the color and depth data of the real-time object using depth camera. Here, we store the depth and color data in two separate bitmaps. Then we set the inputs of the lens properties such as gap, focal length, pixel pitch of display, etc. Using those values, we implement the pixel mapping algorithm to calculate elemental coordinate pixels. The pixel mapping algorithm could be implemented in two ways. In the first method, the depth and color bitmaps (from input) are used to compute the elemental coordinates for all pixel points and then we create new bitmaps from those data and use them to generate elemental images. However, those bitmaps are created using the Bitmap class of C# which is generally very slow to calculate and thus slows down the generation of EIs. To tackle this, we introduced a second method which is a modification of the first. Instead of computing pixel mapping algorithm from bitmaps directly, we break bitmaps of color and depth into two separate byte type arrays of which holds the values of red, green, blue and alpha (RGBA) for color and depth respectively. Then we manipulate the arrays to implement the pixel mapping algorithm and then we manually create a bitmap at the end for EI generation. As this method did not use the Bitmap class, it proved to be a little faster than the previous method.

At this point, there was no GPU implementation. So, in order to make it even much faster, we implemented GPU parallel processing on the second method.

The resolution was set at 240x320 pixels so we have to calculate the elemental image coordinate for 320x240=76800 pixels. So, it has about 76800 computations. Furthermore, for each pixel in the display panel, we have to calculate the coordinate for each lens in the lens array which is about 30X30. So, there are 900 small lens in the lens array. The total time unit is at least 76800x900 or 69,120,000 calculations. So, if we can somehow reduce the computing time, our frame rate might be improved considerably in theory as the frame rate is determined on how fast the machine calculates all the coordinates in real time. We know that a GPU can process computations much faster than a CPU. If we apply GPU code in that part of the calculation instead, we can reduce the computation time considerably. The remaining codes are handled by the CPU.
For this, we have used a library called CUDAfy (c# wrapper) which joined with c# enables us to write the code in the GPU. For our experiment our graphics card was Nvidia GTX 660 which is of compute capability 3.0. GPU contains different sizes of block numbers in different axis in a grid. Each block can contain maximum of 1024 threads per axis. Our motive is to assign specific threads for performing specific tasks thus proper utilization of GPU. There are two methods by which we could have achieved this. Since we have to reduce computation of four loops, we may launch GPU for first two outer loops and then launch inner kernel for the two innermost loops which could have been the most efficient way. But this requires a feature called dynamic parallelism where compute capability 3.5 is required but due to hardware limitations we decided to go with the second method which is only launching GPU kernel for the two outermost loops which would contain the most calculations compared to the two inner loops. Therefore, we launched kernel with 320 blocks in the x direction and 240 blocks (one block is used to calculate one pixel coordinate

and the threads in the block are used to compute the lens array elements required to calculate the elemental image coordinate) in the y direction along with 32x32 threads within each block and manipulated the global thread ID to carry out our computations.

4.2 Experimental Setup

We have carried out the experiment to verify the proposed method. The setup we used consisted of a high resolution LCD monitor, depth camera, and a PC as shown in the figure below. Table 1 shows the specifications of each apparatus used.

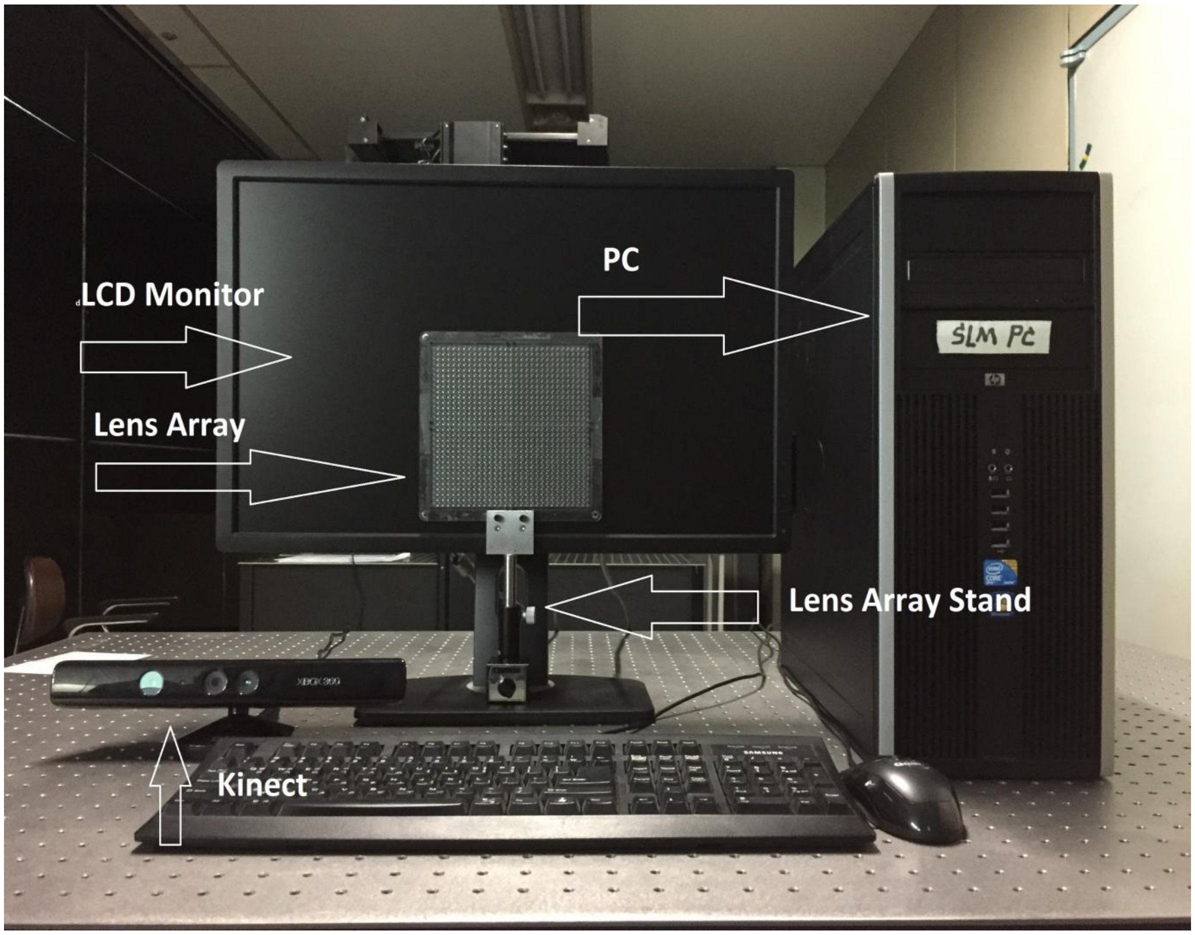


Figure 16: Experimental Setup

Components	**Specifications**	**Features**
Depth-camera	● Model	● XBOX 360 KINECT Sensor
PC	● Microsoft Windows 8.1 Pro ● 64 bit ● Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 3401 Mhz, 4 Core(s), 8 Logical Processor(s) ● 16 GB RAM	● NVIDIA GeForce GTX 660 ● 2GB GDDR5 Memory
Display	● Pitch of pixel Resolution	● 0.265mm ● 1366x768 Resolution
Lens array	● Number of lenses ● Focal length ● Pitch of elemental lens	● 30X30 ● 150 mm ● 5mm each lens

Table 2: Components, Specifications and Features of the experimental setup

For the experiment, we were limited to only use 1 GPU which is NVIDIA GeForce GTX 660

ADVANTAGES OF NVIDIA GeForce GTX 660:

- Decent performance for the price, even at higher resolutions
- Introduces Kepler technologies to the mainstream market

DRAWBACKS OF NVIDIA GeForce GTX 660:

- Not decisively faster than similarly priced AMD cards in many tests
- Power usage higher than that of competitors

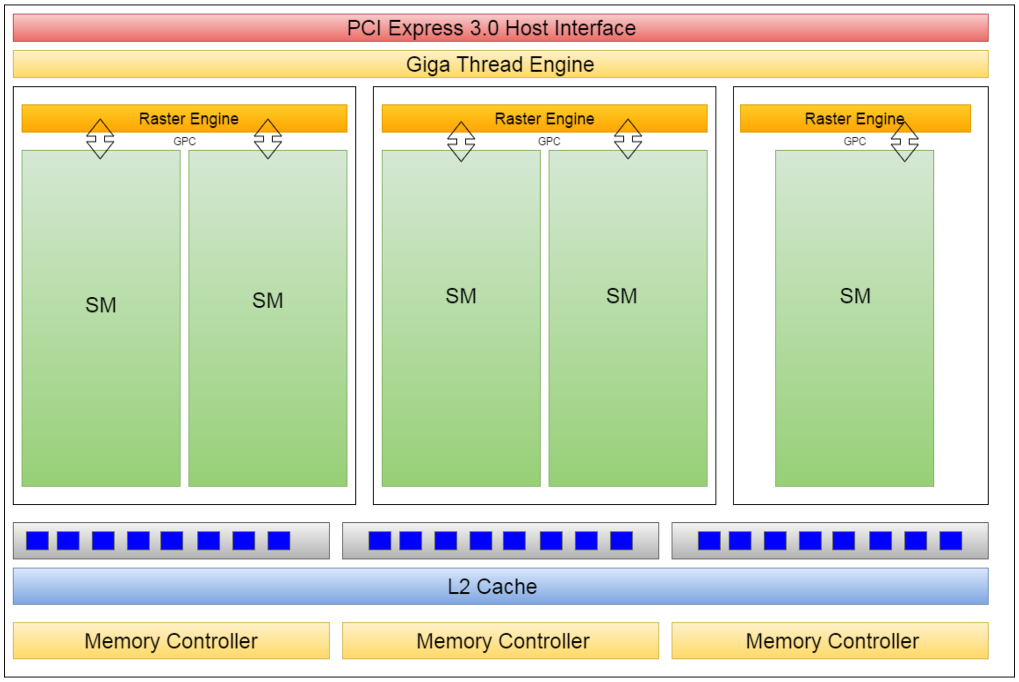


Figure 17: Basic architecture of NVIDIA GeForce GTX 660

Some of the features of NVIDIA GeForce GTX 660 are as follows:

GPU Engine Specs:

- ❑ 960 CUDA Cores
- ❑ 980 Base Clock (MHz)
- ❑ 1033 Boost Clock (MHz)
- ❑ 78.4Texture Fill Rate (billion/sec)

Memory Specs:

- ❑ 6.0 Gbps Memory Clock
- ❑ 2048 MB Standard Memory Config
- ❑ GDDR5 Memory Interface
- ❑ 192-bit GDDR5 Memory Interface Width
- ❑ 144.2Memory Bandwidth (GB/sec)

Feature Support:

- ❑ GPU Boost, PhysX, TXAA, NVIDIA G-SYNC-ready Important Technologies
- ❑ 3D Vision, CUDA, Adaptive VSync, FXAA, 3D Vision Surround, SLI Other Supported Technologies
- ❑ 4.3OpenGL
- ❑ 12 API Microsoft DirectX
- ❑ PCI Express 3.0 Bus Support
- ❑ Yes Certified for Windows 7
- ❑ Yes 3D Vision Ready

CHAPTER 5: Results

5.1 Results and Discussion

We were successful in verifying our proposed method for faster computation of EIs. The whole system was programmed and developed under Microsoft Visual Studio Professional 2010 and Kinect SDK version 1 in Windows 8.1 Operating System. To fill out our data table, we took readings for EI generation for approximately 1000 frames for a total of 5 attempts. For comparison we prepared the datasets using two different methods. At each attempt, we took data by orienting the depth camera in different directions.

Implementation using Bitmap Class

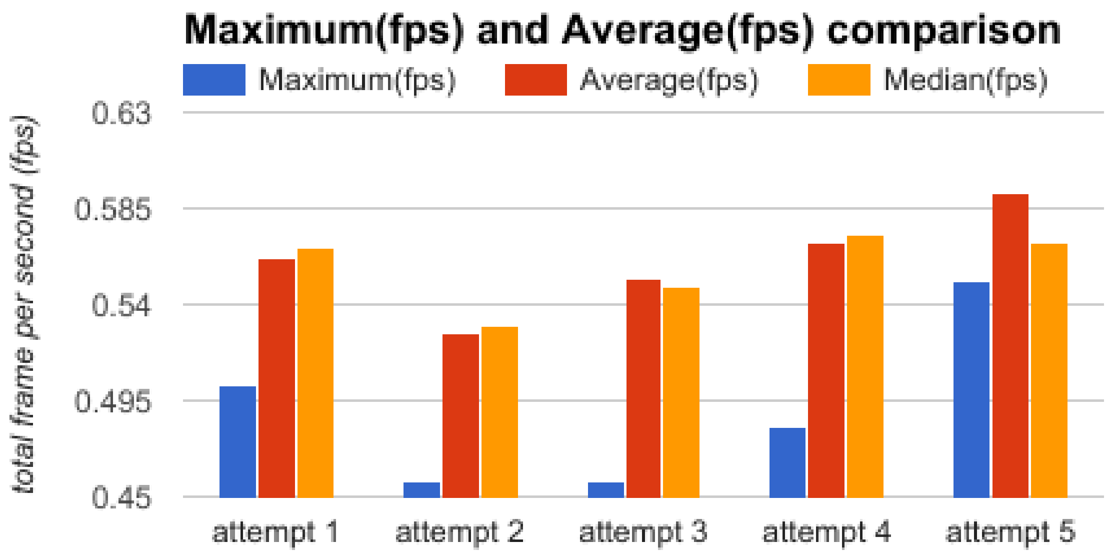


Figure 18: Bar chart of FPS against number of attempts by using Bitmap Class

As we can see from figure 18, attempt 5 have produced the best results, so we produced another graph for our best attempt.

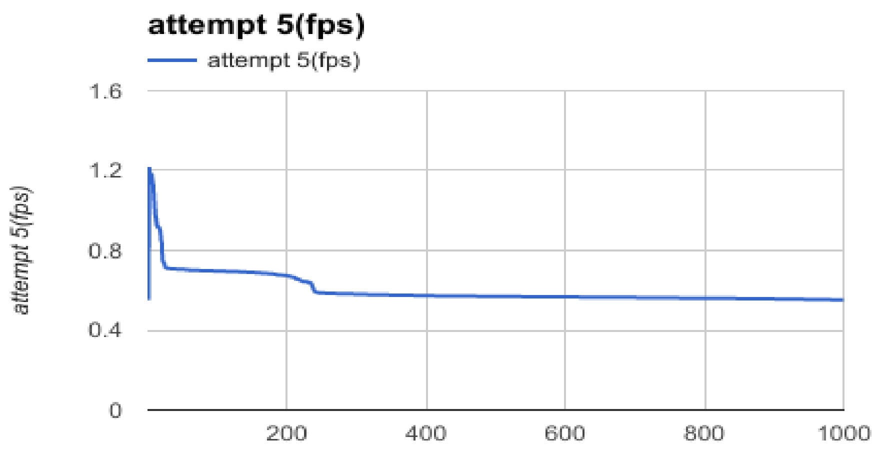


Figure 19: Graph of FPS against number of frames by using Bitmap Class

<u>Implementation using GPU parallel processing</u>

Figure 20 shows the time it takes for a frame to complete processing and arrive in milliseconds for a total of 1000 frames.

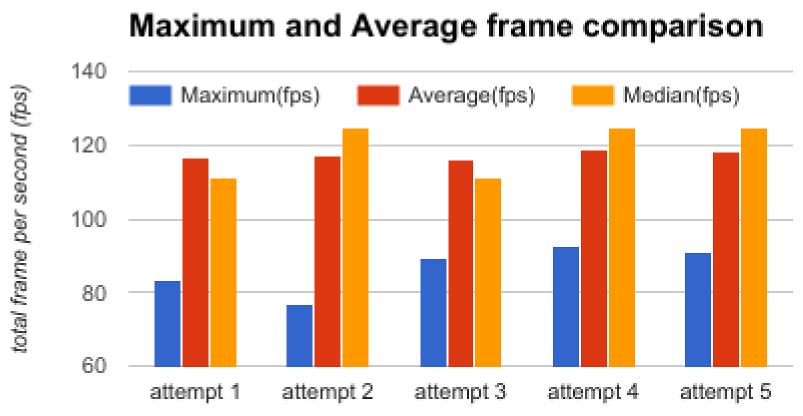


Figure 20: Bar chart of FPS against number of attempts using GPU

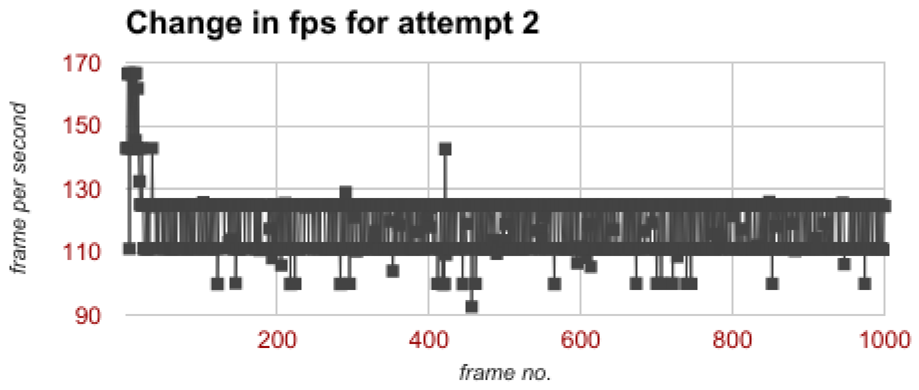


Figure 21: Graph of FPS against number of frames using GPU

Since we were able to generate EIs at a rate of more than 110 fps. We can implement this in multidirectional projection scheme [21]. Doing so decrease the fps by half but still it will provide around 30 fps which is enough for real time. Figure x shows the estimated frame rate per second as a prototype for projection in multidirectional.

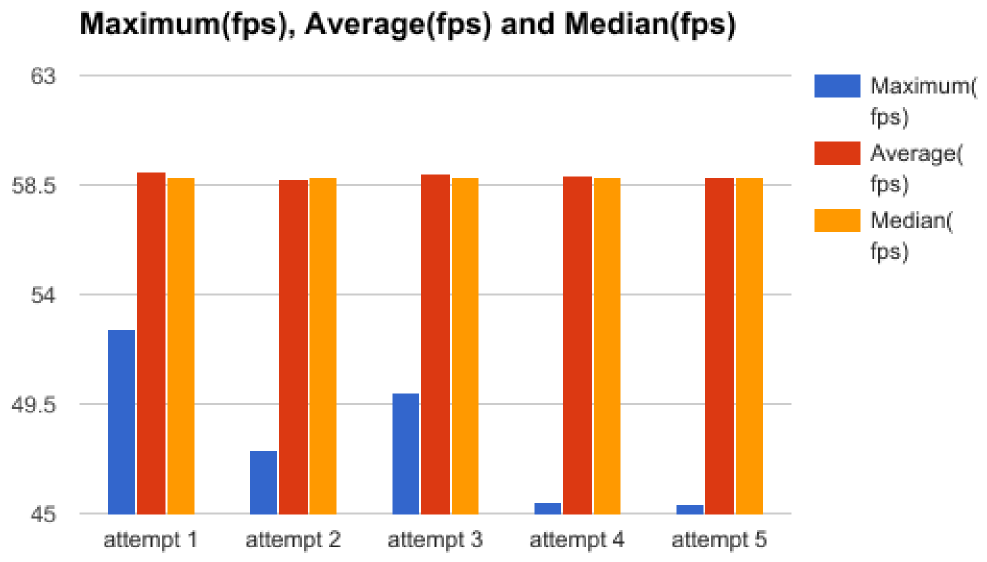


Figure 22: Bar chart of FPS against number of attempts for multidirectional

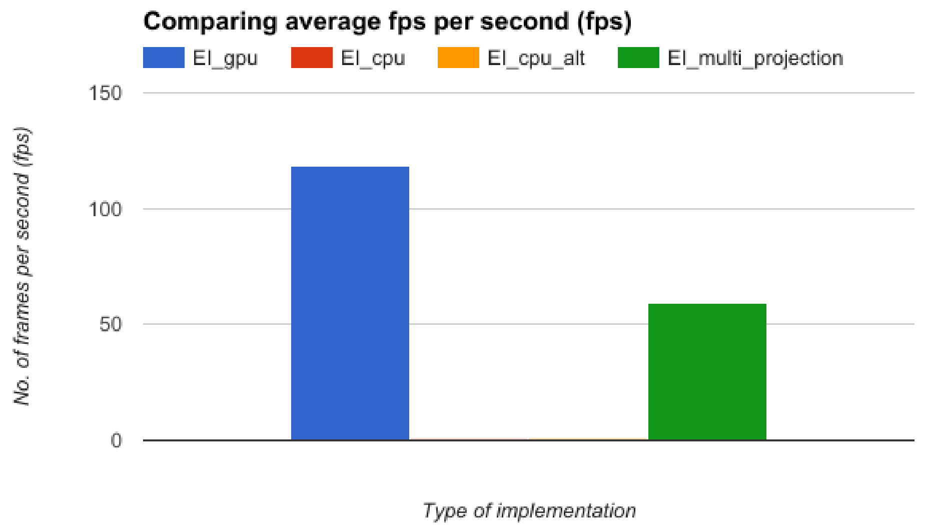


Figure 23: Bar chart of average FPS against the type of implementations

Figure 23 shows the overall comparison between all the methods that we stated and experiment including the prototype for projection in multidirectional.

Chapter 6: Conclusion

6.1 Conclusion

In this paper, we proposed a faster computation method of generating EIs using GPU parallel processing in a real time integral image display system using a depth camera. The depth and color data of the real world objects are acquired by a depth camera. These data are then processed in generating EI's by implementing GPU parallel processing. However, after achieving these feats, there were still some limitations to this experiment which acted as a little barrier to our research. First of all, we didn't have the scope to try and implement this system on different computer systems with different graphics cards and different lens property setups which could have altered the performance with respect to each system spec and provided a better insight on our proposed system. In theory, the higher the spec of the graphics card, the better it will perform in computation as the GPU can compute more consequent blocks at a time so processing time is even faster. Furthermore, the GPU that we used was of compute capability 3.0. If we had a GPU system with 3.5 or higher compute capability then we could have implemented dynamic parallelism. Dynamic parallelism is generally useful for problems where nested parallelism cannot be avoided. It simplifies GPU programming by allowing programmers to easily accelerate all parallel nested loops, resulting in a GPU dynamically spawning new threads on its own without going back to the CPU. With this feature, we can improve the rate of generation of EIs even more.

This fast generation of EIs can aid in implementing multi direction projection system where two projectors will project two different EIs at a time in real time. With enough further improvements, more than two projections could be possible. Therefore, a lot work, research and analysis can still be done in this field in order to make the whole system more commercial friendly.

References

[1] G. Lippmann, “La photographie integrale,” Comptes-Rendus Acad. Sci 146, 446–451 (1908).

[2] N. Dodgson, J. Moore and S. Lang, "MULTI–VIEW AUTOSTEREOSCOPIC 3D DISPLAY", International Broadcasting Convention, pp. 497-502, 1999.

[3] Shaw, Terence. “Multiscopic neuro-vision for two and three dimensional object recognition and classification.” (1998).

[4] G. Li, K. Kwon, G. Shin, J. Jeong, K. Yoo and N. Kim, "Simplified Integral Imaging Pickup Method for Real Objects Using a Depth Camera", Journal of the Optical Society of Korea, vol. 16, no. 4, 381-385, (2012).

[5] Y. Kim, H. Choi, J. Kim, S.-W. Cho, Y. Kim, G. Park, and B. Lee, “Depth-enhanced integral imaging display system with electrically variable image planes using polymer dispersed liquid-crystal layers,” Appl. Opt. 46, 3766-3773 (2007).

[6] J. Hahn, Y. Kim, E.-H. Kim, and B. Lee, “Undistorted pickup method of both virtual and real objects for integral imaging,” Opt. Express 16, 13969-13978 (2008).

[7] T.poon, “DIGITAL HOLOGRAPHY AND THREE DIMENSIONAL DISPLAY,” Optical Image Processing Laboratory, Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg,Virginia 24061.

[8] J. Jeong et al.,“Development of a real-time integral imaging display system based on graphics processing unit parallel processing using a depth camera,” Proc. SPIE, 53(1),015103-4,015103-8 (2014).

[9] Yunhee Kim, Jae-Hyeung Park, Sung-Wook Min, Sungyong Jung, Heejin Choi, and Byoungho Lee, "Wide-viewing-angle integral three-dimensional imaging system by curving a screen and a lens array," Appl. Opt. 44,546-552 (2005).

[10] M. Shin, G. Baasantseren, K.-C. Kwon, N. Kim, and J.-H.Park, “Three-dimensional display system based on integral imaging with viewing direction control,” Jpn. J. Appl. Phys.49, 072501 (2010).

[11] B. Lee, S. Jung, and J.-H. Park,“Viewing-angle-enhanced integral imaging by lens switching,”Opt. Lett.27,818–820(2002).

[12] Y. Kim, J.-H. Park, S.-W. Min, S. Jung, H. Choi, and B. Lee,“Wide-viewing-angle integral three-dimensional imaging system by curving a screen and a lens array,”Appl. Opt.44, 546–552 (2005).

[13] J.-S. Jang and B. Javidi,“Improvement of viewing angle in integral imaging by use of moving lenslet arrays with low fill factor,”Appl. Opt.42, 1996–2002 (2003).

[14] J.-S. Jang and B. Javidi,“Three-dimensional projection integral imaging using micro-convex-mirror arrays,”Opt. Express12, 1077–1083 (2004).

[15] R. Martinez-Cuenca, H. Navarro, G. Saavedra, B. Javidi, and M. Martínez-Corral,“Enhanced viewing-angle integral imaging by multiple-axis telecentric relay system,”Opt. Express15, 16255–16260 (2007).

[16] H. Choi, J.-H. Park, J. Kim, S.-W. Cho, and B. Lee,“Wide-viewing-angle 3D/2D convertible display system using two display devices and a lens array,”Opt. Express 13,8424–8432 (2005).

[17] H. Choi, S.-W. Min, S. Jung, J.-H. Park, and B. Lee,“Multiple-viewing-zone integral imaging using a dynamic barrier array for three-dimensional displays,”Opt. Express 11, 927–932(2003).

[18] G. Baasantseren, J.-H. Park, K.-C. Kwon, and N. Kim,“Viewing angle enhanced integral imaging display using two elemental image masks,”Opt. Express 17, 14405–14417(2009).

[19] M. Shin, G. Baasantseren, K.-C. Kwon, N. Kim, and J.-H. Park,“Three-dimensional display system based on integral imaging with viewing direction control,”Jpn. J. Appl. Phys.49, 072501 (2010).

[20] M. A. Alam , M.-L. Piao,L. T. Bang ,N. Kim,"Viewing-zone control of integral imaging display using a directional projection and elemental image resizing method," Appl. Opt., vol. 52, no. 28, pp. 6969-6978, Oct. 2013.

[21] M. A. Alam, K.-C. Kwon, Y.-L. Piao, Y.-S Kim, N. Kim, “Viewing-Angle-Enhanced Integral Imaging Display System Using a Time-Multiplexed Two-Directional Sequential Projection Scheme and a DEIGR Algorithm,” IEEE Photonics Journal, Vol. 7, No. 1, February 2015.

[22] F. Okano, H. Hoshino, J. Arai, and I. Yuyama, “Real-time pickup method for a three-dimensional image based on integral photography,” Appl. Opt. 36, 1598-1603 (1997).

[23] H. Navarro, R. Martinez-Cuenca, G. Saavedra, M. Martinez Corral, and B. Javidi, “3D integral imaging display by smart pseudoscopic-to-orthoscopic conversion (SPOC),” Opt. Express 18, 25573-25583 (2010).

[24] D.-Q. Pham, N. Kim, K.-C. Kwon, J.-H. Jung, K. Hong, B. Lee, and J.-H. Park, “Depth enhancement of integral imaging by using polymer-dispersed liquid-crystal films and a dual-depth configuration,” Opt. Lett. 35, 3135-3137 (2010).

[25] Y. Kim, H. Choi, J. Kim, S.-W. Cho, Y. Kim, G. Park, and B. Lee, “Depth-enhanced integral imaging display system with electrically variable image planes using polymer dispersed liquid-crystal layers,” Appl. Opt. 46, 3766-3773 (2007).