PARALLEL IMPLEMENTATION OF 3-D IMAGE PROCESSING

By

Huan Lin

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Electrical Engineering

2012

ABSTRACT

PARALLEL IMPLEMENTATION OF 3-D IMAGE PROCESSING

By

Huan Lin

A structured light based single-direction 3-D navigation system consists of a projector, a camera, and an infrared light filter. The system serves as a navigation sensor to help the robot to perform the task of grabbing an object in front of the robot. It uses a unique algorithm to calculate 3-D wold coordinates of each point viewed by the camera from the pixel information obtained from the camera. Through experiments, the computation time of the existing algorithm of this navigation system is relatively long. To improve the system, optimization needs to be applied. There are nine functions executed in the existing algorithm. Comparing the processing time of each function, decoding is the one which takes more than half of the entire program's computation time. The existing decoding algorithm in the single-directional 3-D navigation system is designed and implemented using CPU sequential serial computing algorithm. To improve, GPU's (Graphics processing unit) which has unique parallel architecture can be added to the system. However, how to modify and re-design the serial computing algorithm to efficiently utilize the GPU's parallel architecture is a challenge. In this paper, I am going to present a parallel decoding algorithm using CUDA-based GPU to accelerate the existing serial decoding algorithm more than 2000 times.

To Dr. Xi for always guiding me and supporting me during my study time in MSU.

To the committees who spend time to review my thesis and give me suggestions.

To all my professors in the ECE Department for always coaching me and supporting me.

To all the faculties and stuff for always helping me and assisting me.

To my awesome parents for always believing in me and loving me.

To my grandparents for inspiring me in my childhood, for always loving me and supporting me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction to the Parallel Computing

1.1 What is Parallel Computing

Microprocessors based on a single central processing unit (CPU) have played a major role in science computation speed increases and cost reductions in computer applications. In the past, scientists have been focused on CPU application software developing to improve the computer's performance. These performance improvements have allowed applications software to provide more functionality user interfaces, and generate more useful results. During this period, CPU serial computing was used to develop computing algorithms to increase the speed of their applications under the hood. Software written for serial computation to be run on a single Central Processing Unit (CPU) first breaks a problem into a discrete series of instructions and then executes each instruction sequentially. Each instruction may execute only one time. For large scale data stream, the hardware architecture of the CPU and the software program structure of the CPU cause applications have disadvantages on computation time and application cost. To improve the functionality of CPU serial computing, scientists innovate and developed a new computing method called parallel computing.

Parallel computing is an evolution of serial computing. It significantly increases the

computation time and reduces the cost of applications. Parallel computing was first introduced to use a single computer with multiple processors to solve a computational problem simultaneously. The program structure is to first break a problem into discrete pieces of work which can be solved concurrently and then execute multiple program instructions from each part simultaneously on different CPUs. Using multiple compute resources (CPUs) in a single compute resource to solve computational problem simultaneously significantly improved the CPU serial computing computation ability. In this approach, CPUs need to perform the computation tasks as well as the data transformation tasks between CPUs. However, for many applications used in the areas of atmosphere, earth and environment analysis, applied and nuclear physics and biomedical image processing which involve very large scale data stream computation and transformation, the performance level of parallel computing using multiple CPUs is still not enough. This problem can be solved by adding an additional hardware to independently handle the parallel computing task from the CPU. In fact, CPU can be used to perform other necessary serial computing tasks and handle data transformation. In this approach, the large data stream transformation time will be saved.

The add-on hardware of this approach has to have high computation ability and good communication ability with the CPU. GPU, Graphics Processing Unit, which is majorly used in image processing, now has been found as a better developed hardware to use to speed up large data stream program computation speed. It serves as an accelerator to the CPU in the performance of parallel computing applications. This type of parallel computing was named GPU parallel computing. Due to the significant improvement of the GPU parallel computing, the

characteristic of Parallel computing using GPU started to be more and more evaluated and researched by hardware and software developers. Developing softwares using GPU parallel computing, scientists do not only need to consider the program structure of each programming languages but also need to consider the hardware architecture characteristics. Not every serial computing algorithm can directly translate to GPU parallel computing algorithm. How to implement and redesign the existing serial computing algorithm using GPU parallel computing and how to efficiently utilize the GPU for the best performance of parallel computing are the main focus of researchers.

1.2 Motivation

Structured light vision system has been successfully used to obtain the 3-D model of an object. To achieve 3-D measurement accurately in realtime, having fast computation speed is important. A structured light based single-directional 3-D camera system, consisting of a projector, a camera, and a infrared light filter was used in this research as a navigation sensor on the robot to generate the 3-D image information in real-time for robot remote control, which can be seen on Figure 1.2. [13] Figure 1.1 shows the hardware components of this 3-D camera system. The flow chart of the software working principle of this single-directional 3-D camera system is shown in Figure 1.3. The camera takes a picture of the object in front of the robot and returns the pixel information of the image to the computer. Switched the camera mode to an infrared mode, we then re-takes the image and assigns the pre-defined projector marker pattern

to the image. The pre-defined projector marker pattern is shown in figure 1.4. The computer extracts each marker's camera coordinates information to calculate the orientation of each marker 'o', then returns the camera coordinates and orientation of each marker to the next step decoding. The pre-defined projector marker pattern has four types of markers shown as in figure 1.5. The orientation 'o' is defined by the angle of the central moment of each marker and the x-axis shown in figure 1.5. On a 360° bases, if the angle is 0°or 180°, we assign the orientation as 0; if the angle is 45°or 225°, we assign the orientation as 1; if the angle is 90°or 270°, we assign the orientation as 2; if the angle is 135°or 315°, we assign the orientation as 3. Decoding then reads the information of each marker and decodes them to several 3×3 matrices shown in figure 1.6. Each marker is looked as a target marker. Assign each target marker's camera coordinates and orientation information in the center of each individual 3×3 matrix. The rest eight positions are filled by the eight shortest distance markers from the targe marker. How to allocate these eight markers is based on their angle ' θ ', which is shown in figure 1.6. After the matrices are formed, we use these nine markers' orientation to form a nine digit codeword which shown in figure 1.6. Then we send these codewords to the computer and perform the next step, corresponding matching with the codeword extracted from the projector. After matching is confirmed, we perform 3-D reconstruction to get the 3-D world coordinates of the object and transfer this information to form a color fusion image for the robot to view.

Of all the 3-D sensing methods, speed and accuracy are the challenges for all researchers. To overcome these challenges, we have to consider how to perform real-time computation of a large array data in fast time. The goal of this research is to speed up the computation time of the entire

program and finally achieve video rate to better assist the robot performance, so the robot can quickly and accurately grab the object in front of it. Due to the study of experiments, among all nine functions, decoding is the one which takes the most processing time. The average computation time of Decoding is about an average of 57% of the entire software execution time. Therefore, optimizing the Decoding algorithm is the first and the most important step. The details of the serial decoding algorithm is shown in the flow chart in Figure 1.7. This paper is going to introduce a new parallel decoding algorithm for the single-directional 3-D camera system to achieve significantly faster computation speed for high dimensional large data stream.

1.3 Requirement of the Application

The requirement of this application are hardware selecting, software sorting algorithm developing and implementation. Data mining has become a hot research domain in nowadays. It has been widely applied to a variety of fields, such as business intelligence, customer relationship management, navigation systems, scientific simulation, statistical model configuration, e-commerce, biomedical data computation etc. Instead of implementing large data input on serial computed CPU, parallel computed GPU emerged as the co-processor of the CPU to achieve a high overall throughput to speed up the high computing performance of general-purpose applications. The GPUs provide tremendous memory bandwidth and computational horsepower not only for vertex and pixel processing pipelines but also for non-graphical general purpose applications.

Sorting is an algorithm used for a wide range of applications such as geographic information systems, computational biology, search engines and so on. Any application utilized by database operations may benefit from an efficient sorting algorithm. In this research, I will present an efficient parallel sorting algorithm to fast decode the 3-D image information generated from the single-directional camera system.

1.4 Challenge and difficulties

Due to the trend of parallel architectures being commonly applied into consumer hardware, parallel algorithms such as parallel sorting are becoming more and more important for researchers. Various parallel data mining algorithms have been introduced and studied, such as parallel clustering, parallel decision tree, KNN (k nearest neighbor), parallel reduction, etc. CUDA-based general-purpose parallel computing has become popular in the past few years.

The Challenge of implementing a one-shot algorithm for this single-directional 3-D camera system in realtime is to minimize the computation time of decoding. How to accurately and quickly decode large data input from the camera's information is the focus of this research. To overcome these difficulties, selecting and implementing an efficient sorting method for the large data stream to optimize the peer's decoding method is the key. The single-directional 3-D camera system Chi [13] presented was implemented with OpenCV based serial computed CPU. This decoding algorithm is a serial computing algorithm which breaks the problem into a discrete series of instructions and then executes each instruction sequentially. The program structure of

this algorithm is to execute one instruction at a time which causes much wait time between instructions. Using this serial decoding algorithm, it is impossible to achieve the goal of video rate for the 3-D navigation sensor. Therefore, to slove the problem I do not only need to implement a new efficient parallel sorting method but also need to add a computation accelerator in the 3-D camera system. From the study, the computing power of GPU is equivalent to a medium-sized supercomputer. As introduced in the previous section of parallel computing, we can say GPU parallel computing is the best approach to help us to achieve the goal of fast computing speed of the 3-D camera system for high dimensional large data stream. NVIDIA's GPU with CUDA environment which uses extended C as programming language for the GPU is the newest and most powerful parallel computing platform. The unique hardware architecture is well known in the market to perform parallel computing tasks. The program structure is a CUDA-based parallel structure. CUDA is the program language for NVIDIA's GPU written in extended C language. In this research, I am going to use NVIDIA's GPU GeForce GTX 470 to implement a parallel decoding algorithm for the single-directional 3-D camera system to achieve significant speed increase of the existing serial decoding algorithm. In addition, the accuracy of the new designed parallel decoding algorithm has to have the same accuracy as the existing serial decoding approach for consistency.

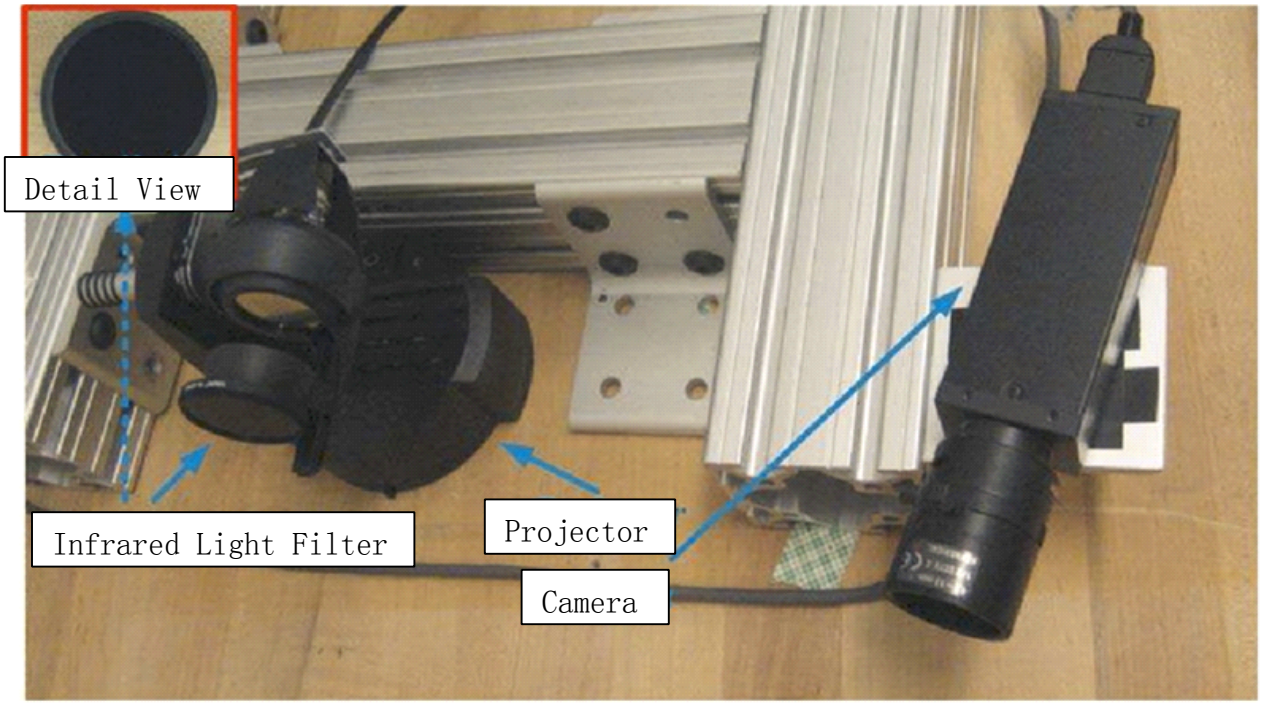


Figure 1.1 single-directional Camera System

For interpretation of the references to color in this and all other figures, the reader is referred to the electronic version of this thesis

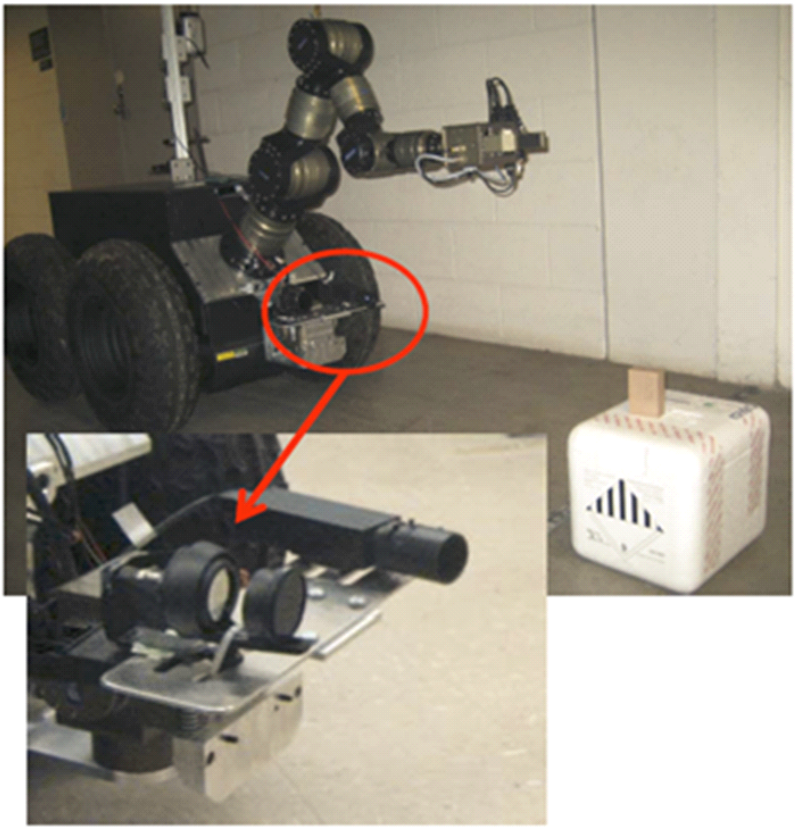


Figure 1.2 single-directional Camera System used as a navigation sensor on the Robot

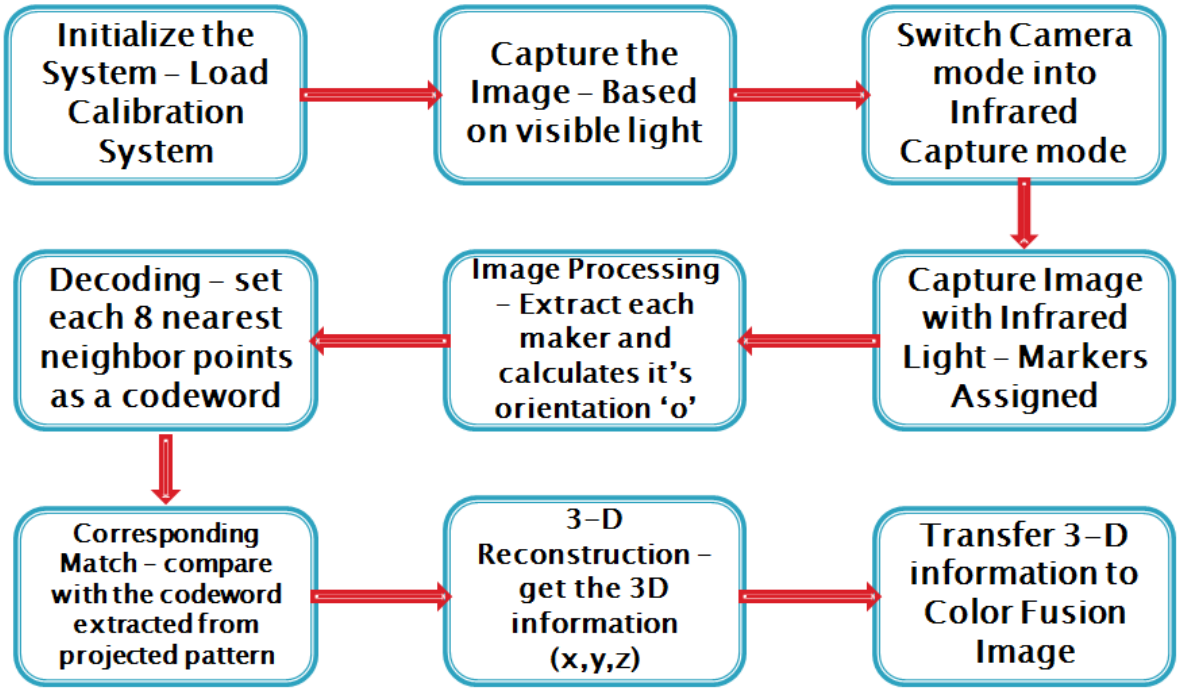


Figure 1.3 Software Working Principle

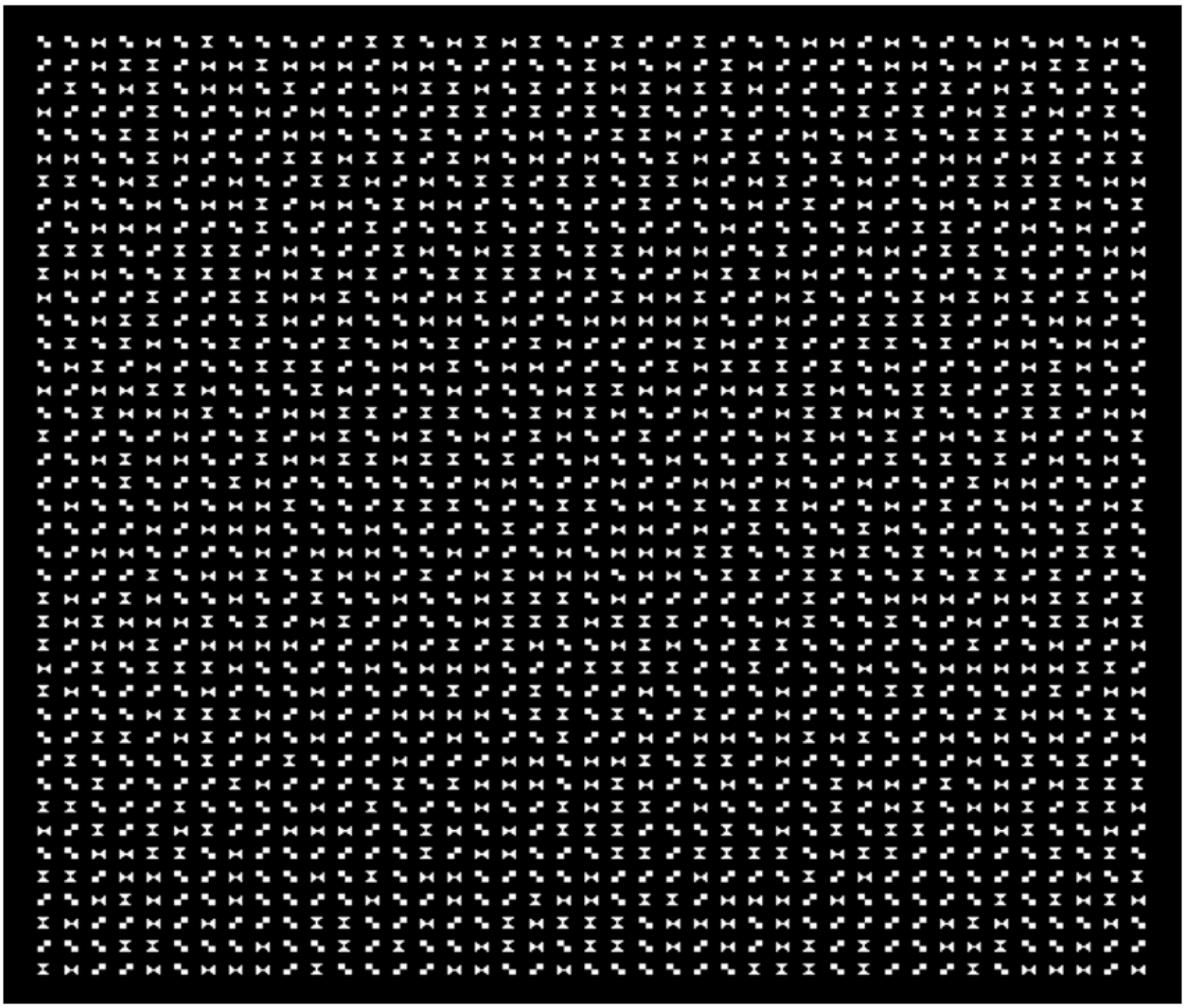


Figure 1.4 Projected Marker Pattern

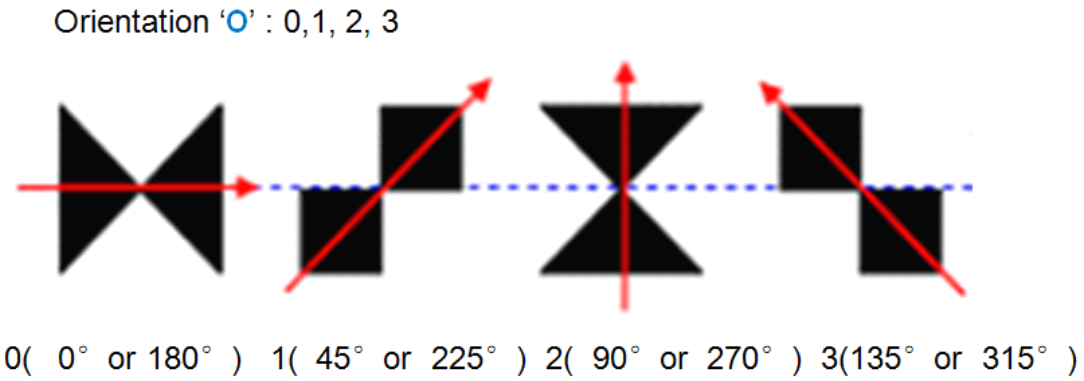


Figure 1.5 Markers orientation

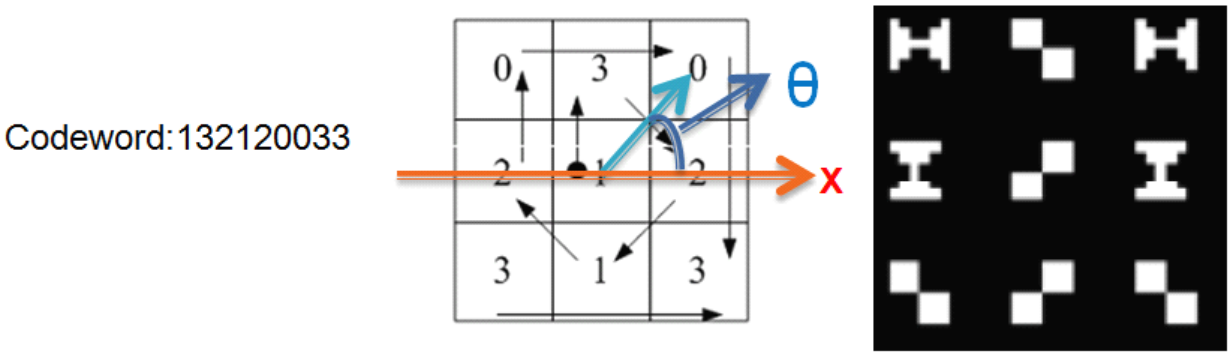


Figure 1.6 3×3 Matrices and Codeword formation

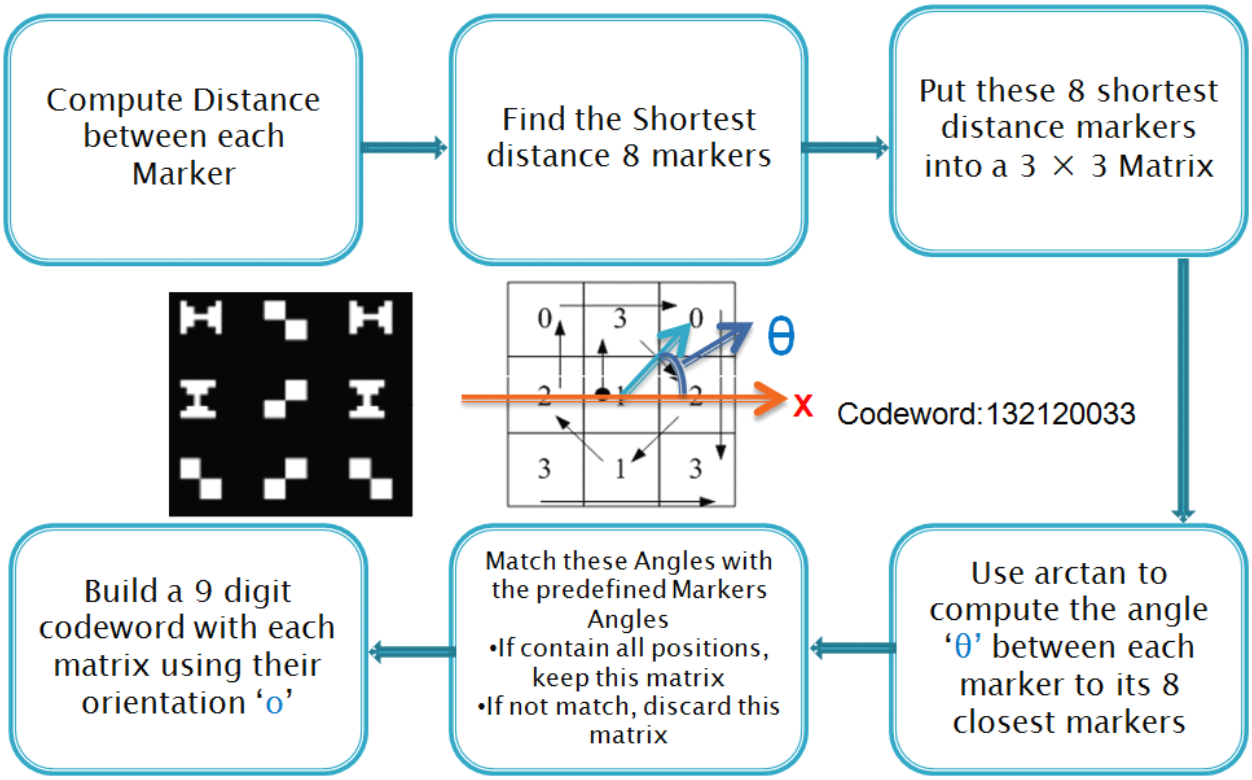


Figure 1.7 Serial Decoding Algorithm Flow Chart

Chapter 2

CUDA Parallel Computing Architecture

2.1 History of GPU

GPU stands for Graphics processing unit. For many years, GPU has played an important role on displaying images and motion on computer displays. Common usages of GPUs include embedded systems, mobile phones, personal computer, workstations and game consoles. However, modern GPUs are efficiently used in manipulating computer graphics as well as large data stream parallel computing algorithms. GPU parallel computing is considered to be "the high end of computing" approach. It has been widely used to model difficult problems in many areas of science and engineering such as atmosphere and earth stricture analysis, applied physics applications, particle movement examinations, biotechnology invention, etc. Not only in the science and engineering filed, GPU is also commonly used to satisfy the commercial applications requirement of processing large amounts of data in sophisticated ways, such as data mining, oil exploration, web search engine, web based business services, medical imaging and diagnosis, financial and economic modeling, networked video and multi-media technologies and collaborative work environments.

The term GPU was defined and popularized by Nvidia in 1999, who marketed the GeForce256（“the world’s first GPU”. It was a single-chip processor which integrated transform, lighting, triangle setup, clipping and rendering engines that were capable of processing a minimum of 10 million polygons per seconds. Several parallel computing architectures have been developed for GPU. The most powerful one was developed by Nvidia called Compute Unified Device Architecture (CUDA). CUDA gives developers access to the virtual instruction set and memory of the parallel computation elements in CUDA GPUs. Rather than executing instructions in a single thread sequentially computed CPUs structure, GPUs has a parallel throughput architecture which can execute many threads concurrently to implement each instruction individually at the same time.

2.2 NVDIA's GPU Architecture

The large performance gap between many-core GPUs and general-purpose multicore CPUs is from the differences in the fundamental design philosophies between these two types of processors illustrated in Figure 2.1. The design of a CPU was optimized for sequential instructions performances versus GPU makes use of sophisticated control logic to allow instructions from a single thread of execution to execute in parallel while maintaining the appearance of sequential execution. In addition, large cache memories are provided to reduce the instruction and data access latencies of large complex applications. Neither control logic nor cache memories contribute to the peak calculation speed. [1]

CUDA-capable GPU is organized into an array of highly threaded streaming multiprocessors (SMs) illustrated in Figure 2.2. In this example, there are four SMs to form a building block; however, the number of SMs in a building block can vary from one generation of CUDA GPUs to another generation. Also, each SM has a number of streaming processors (SPs) that share control logic and instruction cache. Each GPU currently comes with up to 4 gigabytes of graphics double data rate (GDDR) DRAM, referred to as global memory [1]. These GDDR DRAMs are essentially the frame buffer memory which is used for graphics. They can store video images and texture information for three-dimensional rendering.

The NVIDIA's GeForce GTX 400 family of GPUs is based on NVIDIA's Fermi architecture. The initial vision of what a unified graphics card compute processor should look like for NVIDIA is G80. The next GT200 extended the performance and functionality of G80. The Fermi architecture is the most significant leap forward in GPU architecture and is the key of what makes parallel computing on NVIDIA's GPU more reliable efficient than the others [2].

Comparing the Fermi Architecture to the Non-scalable Architecture, the traditional GPU designs use a single geometry engine to perform tessellation. The performance of pixel shading in the early GPU designs used a single pixel pipeline. By noticing how much impact that pixel pipeline grew from a single unit to many parallel units have and the impact on 3-D realism, Nvidia designed their tessellation architecture to be parallel [3]. The most successful parallel tessellation units Nvidia made is this Fermi Architecture GPUs, which implement up to fifteen parallel tessellation units shown in Figure 2.3. In this Architecture, each tessellation unit has its own dedicated shading resource. Up to four parallel Raster Engines transform newly tessellated

triangles into a fine stream of pixel of shading [3]. The close coupling for tessellation, shading, and raster units provides enormous on-chip bandwidth and high execution efficiency [3]. The result is a breakthrough in tessellation performance at over 1.6 billion triangles per seconds [3]. Comparing to other products, Fermi GPUs are 2-8X faster as measured by independent reviews using Microsoft's DirectX 11 software development kit. And this is why I decide to use this Fermi GPU as our hardware to optimize the decoding approach [3].

The heart of the GPU is the Streaming Multiprocessor (SM) which performs the vital functions such as tessellation, pixel shading, physics and compute calculations, etc. The structure of the SM can be seen in Figure 2.4 and Figure 2.5. The graphic card used for this research is GeForce GTX470. The SM of GeForce GTX470 is a highly parallel processor unitizing superscalar excution for optimal performance. Unlike thread level parallelism, superscalar execution is a technique that allows the program to execute sequential instructions in parallel. Since the same program executes in less time, superscalar excution not only improves throughput but also improves latency [3]. The GTX 470 SM is designed to perform two instructions per thread, per clock. Two warps (groups of 32 threads) execute concurrently in the SM. Therefore, a peak of four instructions per clock is realized [3]. The GeForce GTX 470 has fourteen tesselation engines, 448 CUDA cores, and 1GB of memory [3]. It has fifteen cores or streaming multiprocessors (SMs). Comparing with the GeForce GTX 480 introduced by NVIDIA, the GTX470 can facilitate greater instruction throughput by expands the number of execution units of the SMs [3]. Each SM features from 48 CUDA cores. The texture units and the number of special function units (SFUs) is eight.

Compared to the CPU's memory access performance, Fermi is the first GPU architecture with fully cached memory access which is able to cache to all graphic and compute programs. As shown in Figure 2.6 Fermi Memory Hierarchy, programs can access to a Texture Cache, an L1 Cache and an L2 Cache. The texture cache enables fast and efficient texture filtering [3]. And the L1 and L2 caches improve the performance for programs with random memory access patterns such as ray-tracing and physics [3]. General purpose GPU applications such as video trans-coding and photo processing can also fast access the shared memory for storage and data transformation.

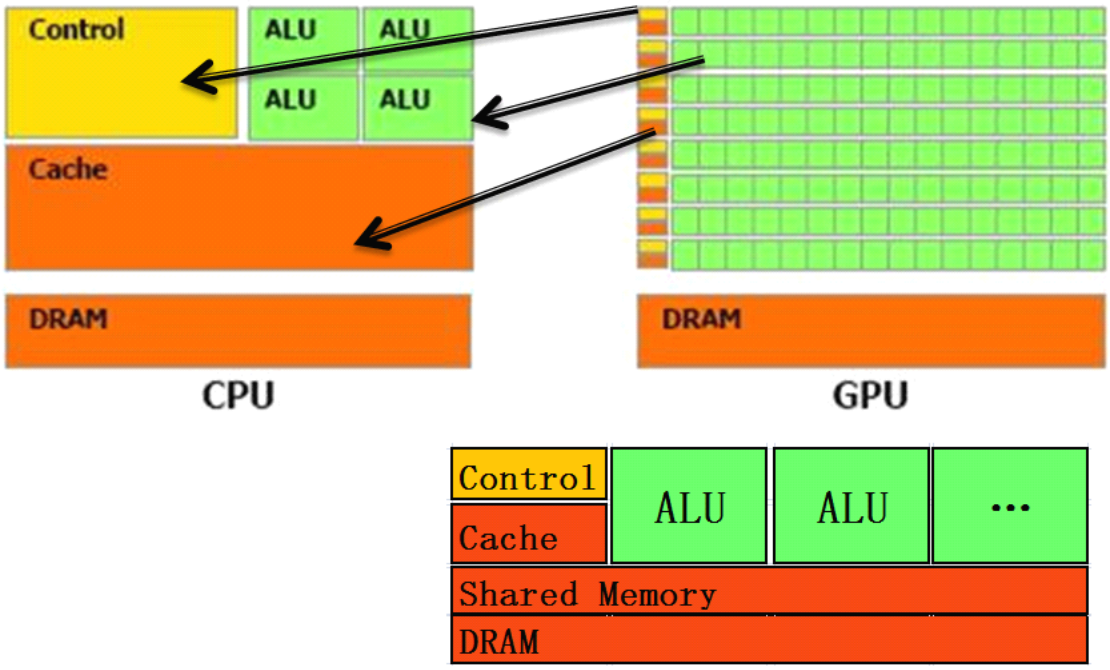


Figure 2.1 Comparison of CPU and GPU Architecture

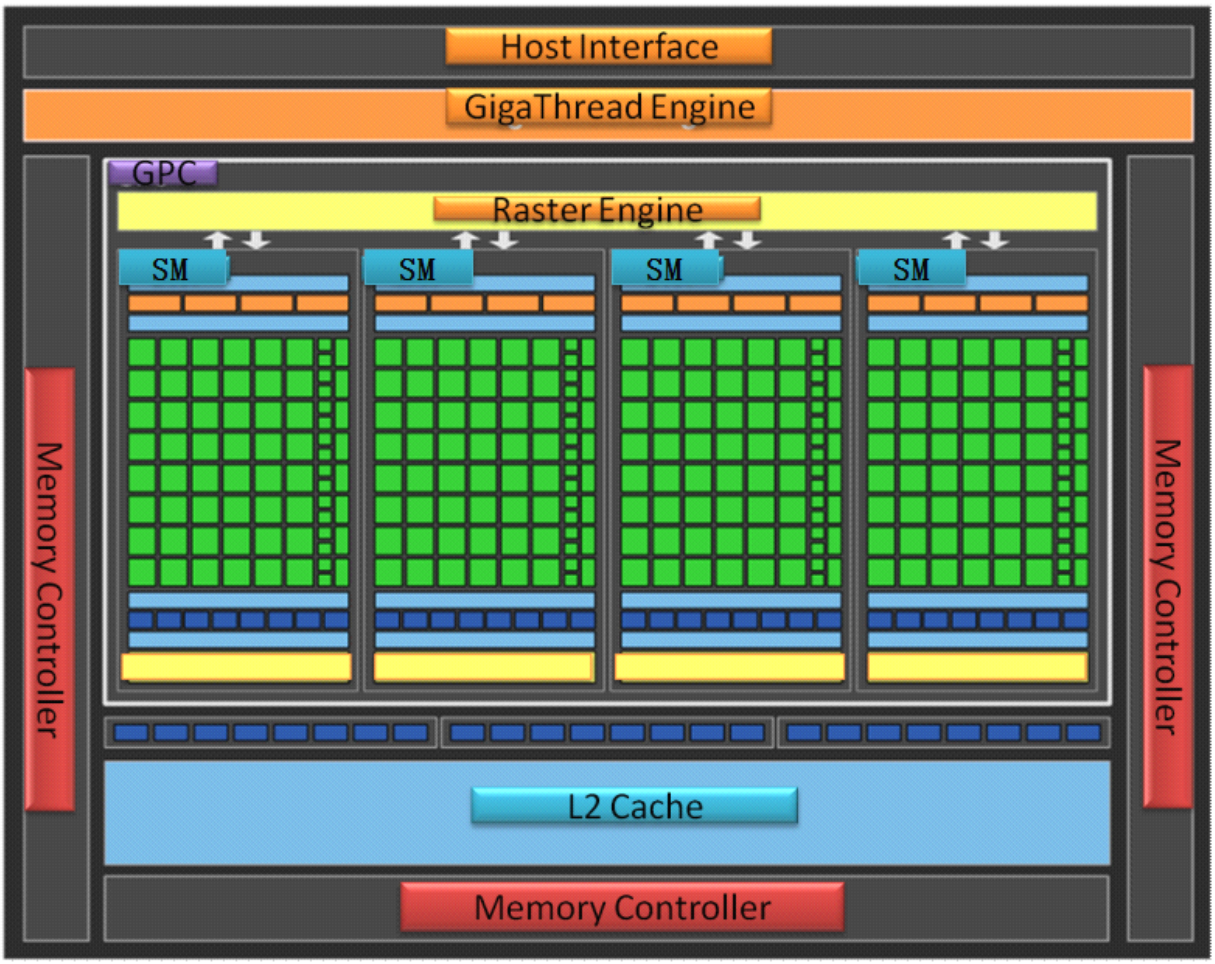


Figure 2.2 Nvidia CUDA-capable GPUs - Fermi Architecture

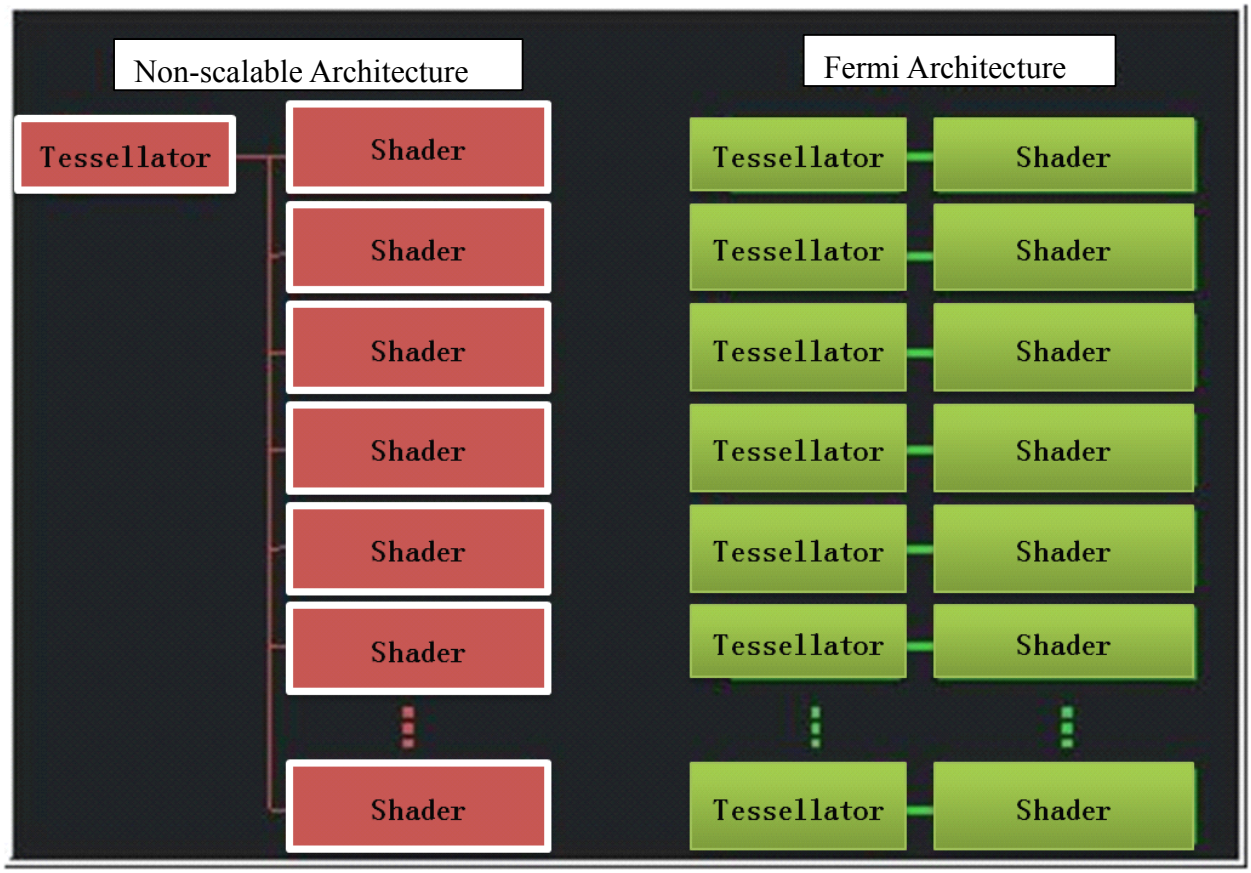


Figure 2.3 Comparison of traditional GPU Architecture and Fermi Architecture

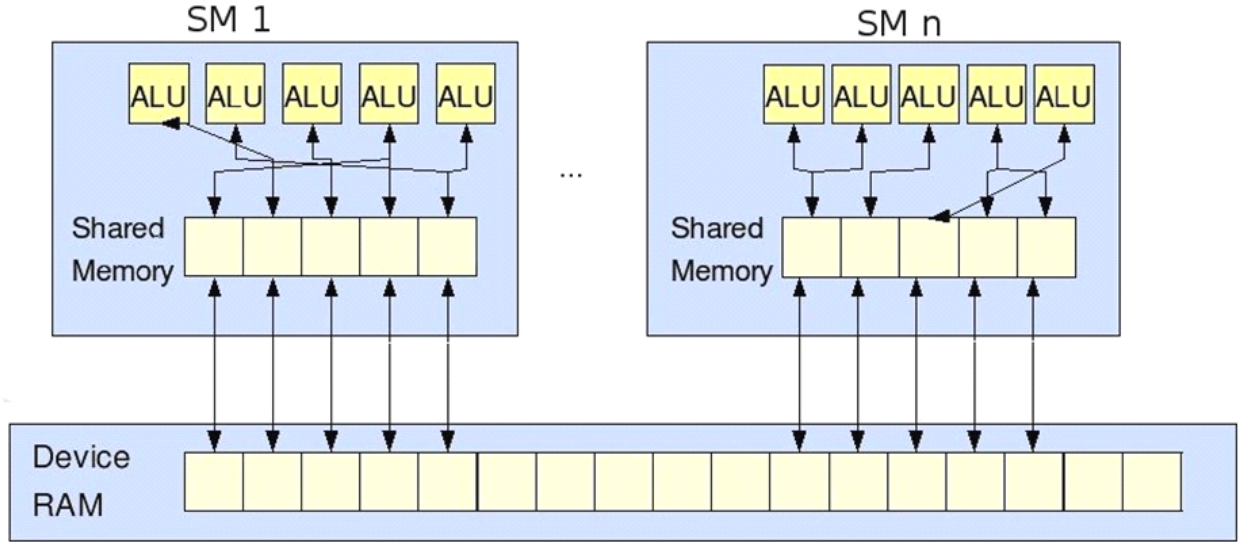


Figure 2.4 SM structure of a GPU

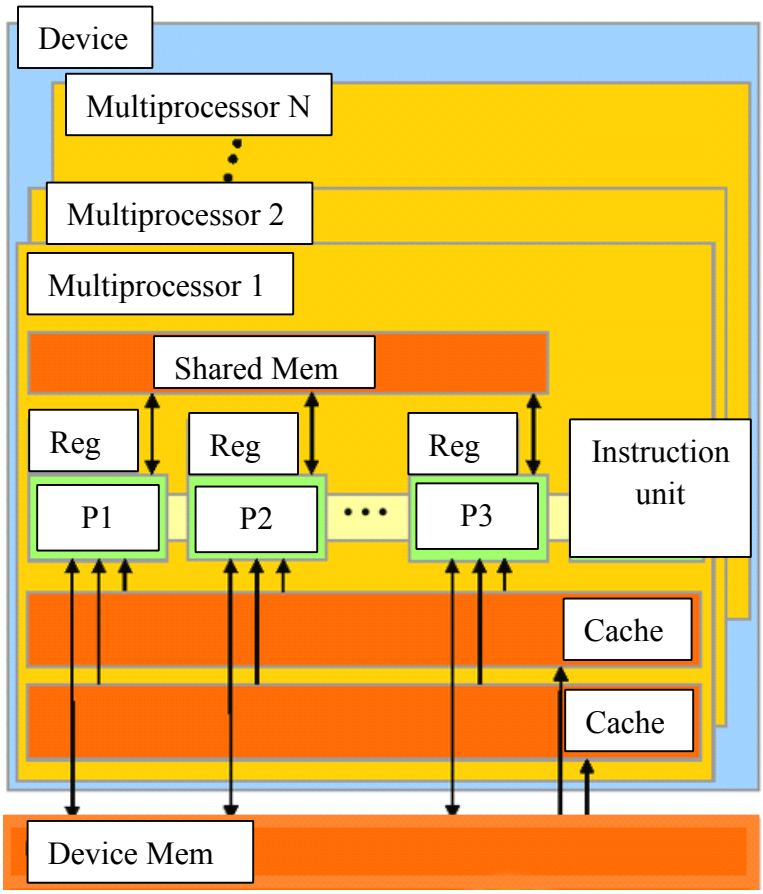


Figure 2.5 Detail Structure of each SM

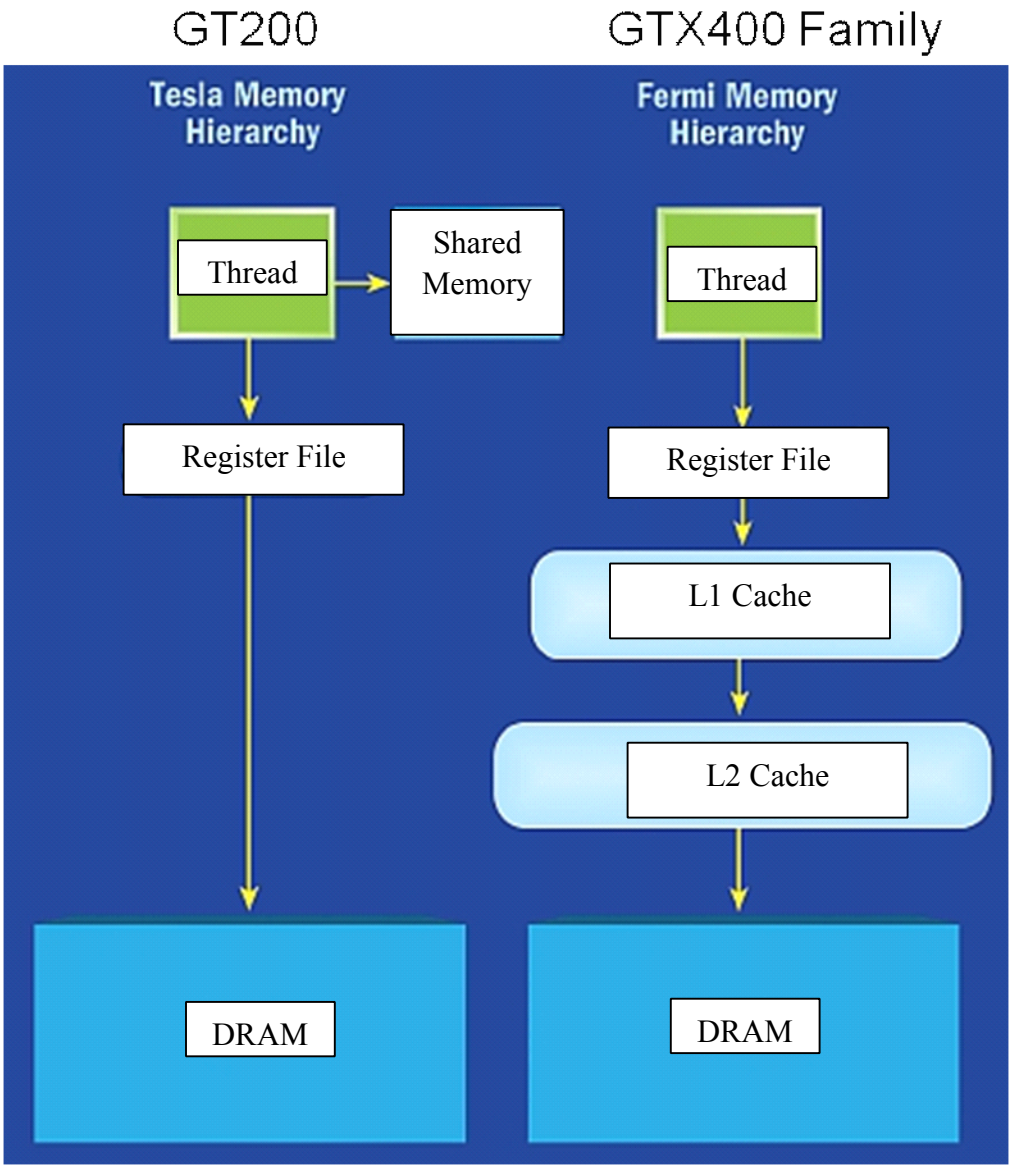


Figure 2.6 Fermi Memory Hierarchy

2.3 Data Parallelism

Nowadays, data parallelism has been widely successfully implemented to improve human life. Consider the satisfaction with digital high-definition television, all the processing that is necessary for the HDTV is a very parallel process, as are 3-D imaging and audio coding and manipulation. Another benefit of parallelism is that it allows developer to develop much better user interfaces offered by greater computing speed comparing to serial computing. Apple iPhone_interface is one of the example. Because of the increase of the speed, customers can enjoy a much more natural interface with touch screen compared to other cell phone devices. In the future, these devices will be able to have more functions which incorporate higher definition, 3-D imaging and computer vision based. GPU-accelerating computer vision is leveraging the computational resources of the GPU for vision to speed up the frame-rate of the camera. How to exploit the computational resources and parallelism offered by modern programmable GPUs in the context of computer vision is a challenge. A good implementation on a GPU can speed up more than 100 times over sequential execution on CPU. However, in order to achieve this, the application must be suitable for parallel execution. Therefore how to design the algorithm and how to implement it to fit and optimize GPUs parallel computing characteristic is a challenge.

Data parallelism refers to the program to perform many arithmetic instructions on the data structures simultaneously. A simple example used in many books to illustrate the concept of data parallelism is a matrix-matrix multiplication. Here, we can use the same example to explain how

data parallelism is processed. In the example of Figure 2.7, performing a dot product between each element of rows in matrix M and each element of columns in matrix N, we generate the product matrix P. The dot product of each row of matrix M and each column of matrix N performs simultaneously. The width of matrix M, matrix N and matrix P has to be the same in order to perform simultaneously. In this multiplication, none of these dot products in matrix P will affect the result in each other. This is called parallel multiplication. For large amounts of data parallelism, GPU with CUDA can significantly accelerate the execution of the matrix multiplication over a host CPU.

During the study, if the percentage of time spent in the parallelized portion of the application is 30%, a 100 times speedup of the parallel portion will reduce the execution time by 29.7%, the speedup for the entire application will be only 1.4 times faster [1]. However, if 99% of the program in parallel computation mode, a 100 times speedup will reduce the application execution to 1.99% of the original time [1]. Which means the entire application is 50 times faster. Therefore, the more execution processes in parallel computation mode, the faster speed the entire application can achieve [1].

Nowadays, various approaches of data parallelism do exist, however four pre-request must be proved before implementing the program, shown as Table 2.1.

1) Select supportable hardware which suits for the programming model and facilitates parallel implementation.
2) Determine the portion of the application that can be parallelized
3) Programming model must not hinder parallel implementation.
4) Data delivery must be properly managed in order to optimize the use of GPUs

Table 2.1 Four pre-request for data parallelism implementation

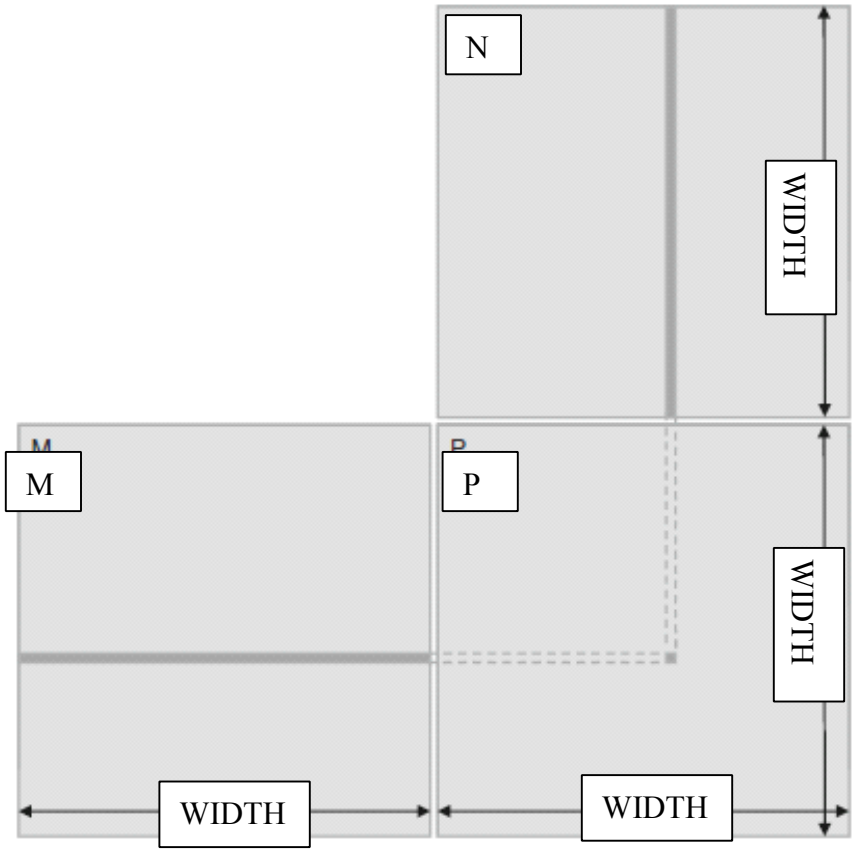


Figure 2.7 Data parallelism in matrix multiplication

2.4 CUDA Program Structure

There are two type of phases of a CUDA program. These phases are executed on either the host (CPU) or a device such as a GPU. The phases that compute little or no data parallelism are implemented in host code [20]. The phases that compute large amount of data parallelism are implemented in the device code [20]. A CUDA program is a structured source code with both host and device code. These two codes are separated by the NVIDIA C compiler during the compilation process. The host code is straight ANSI C code which compiled with the standard C compilers on the host CPU. A function compiled for the device is called a kernel which is written using ANSI C extended with keywords for labeling data-parallel functions and their associated data structures. A kernel is executed on the device as many different threads. These threads work parallelized. Furthermore, the kernel codes will be compiled by the nvcc and executed on the GPU device.

Any source file containing CUDA language extension must be compiled with nvcc, the compiler driver of CUDA. The outputs nvcc generates are host CPU code, which is compiled together with other parts of the application, written in pure C. The Executable files with CUDA code require CUDA runtime library (cudart) and CUDA core library (cuda).

To exploit data parallelism, the kernel functions generate a large number of threads. In the matrix multiplication example Figure 2.7, the parallel computation of each multiplication of the elements in matrix N and matrix M can be implemented as a kernel where each thread is utilized to compute to obtain each element of matrix P. And the number of threads used by the kernel is determined by the dimension of the matrix. For a 1000×1000 matrix multiplication, when the kernel is invoked, 100,000,000 threads will be generated to perform the parallel computation to obtain each element on matrix P simultaneously. The CPU threads typically require thousands of clock cycles to generate and schedule. In contrast to the CPU threads, using CUDA-based GPU, these threads can be assumed to take very few cycles to generate and schedule due to the efficient hardware architecture design.

A device (GPU) is viewed as a compute device operating as a coprocessor to the main host (CPU). Compute intensive functions should be off-loaded to the device [2]. Functions that are executed many times, but independently on different data are prime candidates [2]. Once a kernel is invoked, the execution moves to a device (GPU) to perform the calculation of abundant data parallelism using a large number of threads. All the threads that are generated by a kernel during an invocation are collectively called a grid [20]. KernelA<<<nBIK,nTID>>> tells the device how many blocks needs to be generate and how many threads are generated in each block. In Figure 2.8, there are two grids of threads. Once all threads of the first kernel complete their execution, the corresponding grid terminates. The execution continues with the CPU serial code on the host (CPU) until the seconds kernel is invoked.

Figure 2.10 illustrates the structure of the Grid of thread blocks in more details. Each computational grid consists of a grid of thread blocks. Each thread executes the kernel. The application specifies the grid and block dimensions. The shape of the grid can be 1, 2, or 3-dimensional. The maximal sizes of each grid are determined by the hardware GPU's memory and kernel complexity. Each block has a unique block ID. Within one block, each thread has a unique thread ID. The challenge of the design is how to specify the shape of the grid, the number of the blocks, the number of the threads and the data type being executed on each thread.

Host and device both manage their own memory, which are called host memory and device memory. Data can be copied between them [2]. The CUDA device memory model is illustrated in Figure 2.9. How to allocate and use of the various memory types of a device is very important on accelerating the application. As shown in Figure 2.9, global memory and constant memory are the memories which the host code can transfer data to and from the device. Constant memory allows read-only access by the device code. Mostly only constants locate under constant memory. Most of the variables are likely being put under global memory because these variables can be transferred to and from the device, such as Figure 2.9. The transfer is asynchronous.

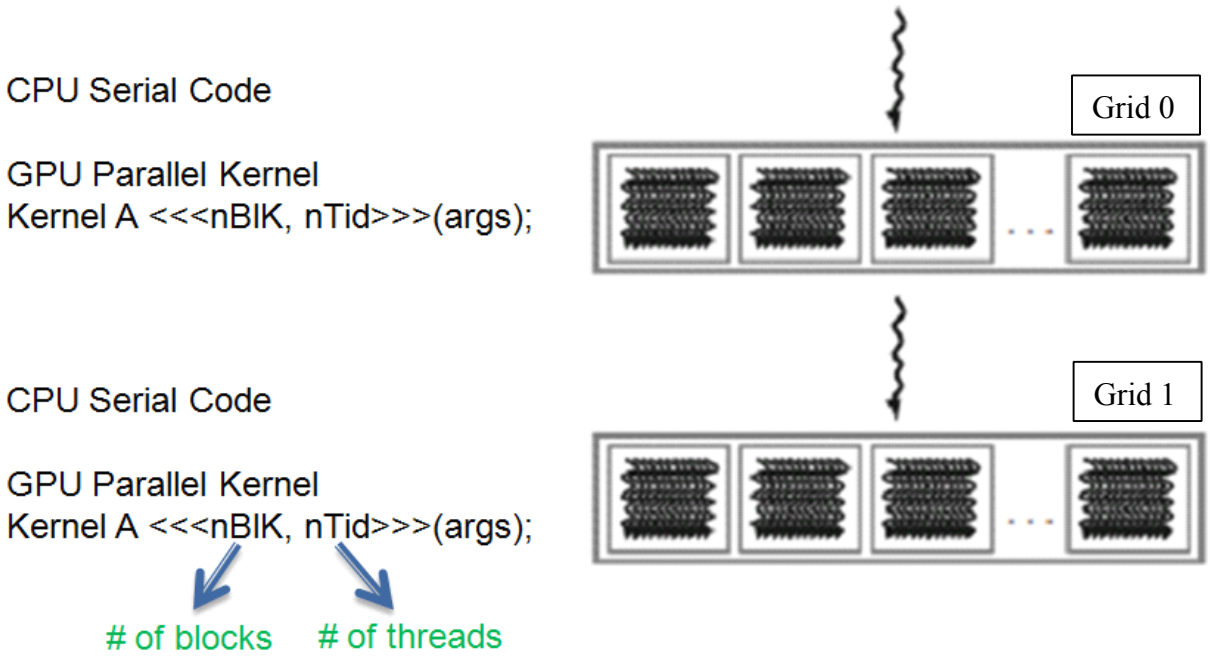


Figure 2.8 CUDA Program Execution Structure

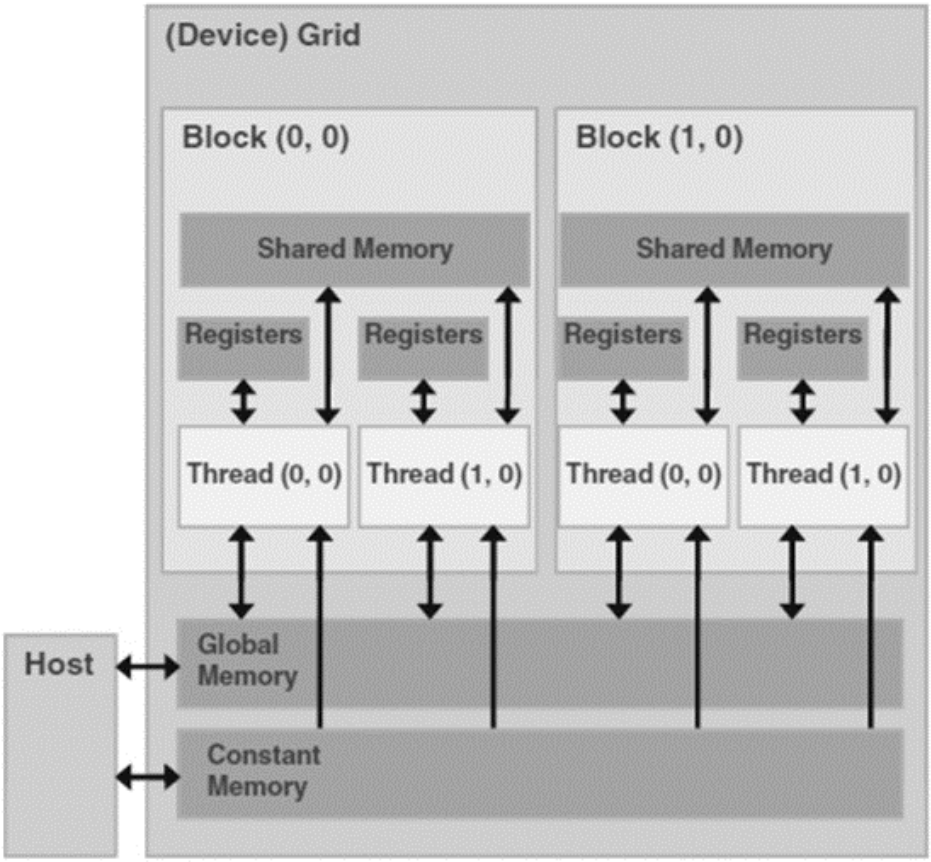


Figure 2.9 Memory transfer process

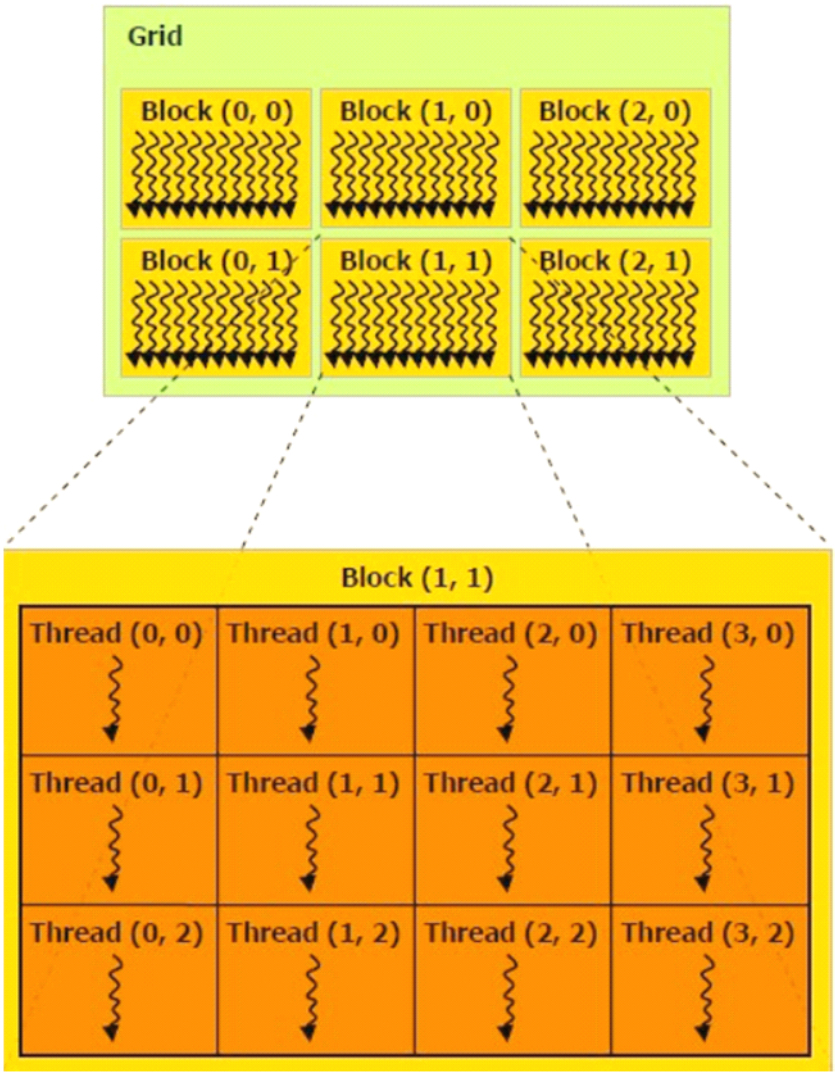


Figure 2.10 The Grid and thread blocks

Chapter 3

Parallel Computing Algorithm of 3-D Image Processing

3.1 Software working principle

The flow chart of the software working principle of this single-directional 3-D camera system is shown in Figure 1.3. The camera takes a picture of the object in front of the robot and returns the pixel information of the image to the computer, then switches to the infrared mode and re-take the image and assign pre-defined projector marker pattern to the image. The pre-define projector marker pattern is shown in figure 1.4.

The computer extracts each marker's camera coordinates information to calculate the orientation of each marker 'o', then returns the camera coordinates and orientation of each marker to the next step decoding. The pre-defined projector marker pattern has four types of markers shown as in figure 1.5. The orientation 'o' is defined by the angle of the central moment of each marker and the x-axis shown in figure 1.5. On a 360° bases, if the angle is 0°or 180°, we assign the orientation as 0; if the angle is 45°or 225°, we assign the orientation as 1; if the angle is 90°or 270°, we assign the orientation as 2; if the angle is 135°or 315°, we assign the orientation as 3. Decoding then reads the information of each marker and decodes them to several 3×3 matrices shown in figure 1.6. Each marker is looked as a target marker. Assign each target marker's

camera coordinates and orientation information in the center of each individual 3×3 matrix. The rest eight positions are filled by the eight shortest distances marker away from the targe marker. How to allocate these eight markers are based on their angle ' θ ', which shown in figure 1.6. After the matrices are formed, we use these nine markers' orientation to form a nine digit codeword which shown in figure 1.6. Then we send these codeword to the computer and perform corresponding match with the codeword extracted from the projector. After matching confirmed, we perform 3-D reconstruction to get the 3-D world coordinates of the object and transfer this information to form a color fusion image for the robot to view.

3.2 Sorting Algorithm

A sorting algorithm is an algorithm that puts elements of a list in a certain order. As one of the most widely studied problems in computer science, sorting is a fundamental problem in many applications which uses a database. Applications, such as geographic information systems, search engine and computational biology all need efficient sorting approaches to accelerate searching as a preprocessing step.

The output of any sorting algorithm must satisfy two conditions:

1) The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);
2) The output is a permutation (reordering) of the input.

Table 3.1 Sorting Algorithm Conditions

Comparing with the main trend in current CPU architecture, highly parallel GPU architecture support and effect thousands of concurrent threads to optimize the utilization of computational resources. Parallel architectures start to find their way into the development of non-parallel applications as it provides an efficient implementation which allows data-parallel processing of individual input tiles by blocks of fine-grained concurrent treads. Parallel algorithms such as parallel sorting are becoming more and more important in the study of programming. There are many different sorting approaches for different usage, such as bubble sort , quicksort , sharker sort, and heapsort , etc. To the benefit of our application, how to choose, modify and create a parallel sorting approach is the key to speed up the computation time and still maintain the same accuracy.

3.3 Decoding Algorithms

The goal of the single-directional 3-D camera system is to provide real-time 3-D coordinates information of the tested object image captured by the vision sensor to the robot. This type of vision sensor can be utilized in many applications, such as, robot navigation, remote manipulation, auto-driving system, map reconstruction, and pedestrian detection. These applications all require fast, accurate and real-time response. Therefore, for large scale data information obtained from the camera, an efficient algorithm which meets these requirements is the key to implement this application.

As described in Section 1.2 Motivation, there are nine steps to implementing the algorithm

for this single-directional 3-D camera in realtime shown in figure 1.3. Comparing the time each step spent, decoding is the most time consuming one. This can been seen on table 3.2. For example, a image which takes about 13 seconds to implement complete the entire program, decoding takes about 8.7 seconds The average computation of decoding is about 57% of the entire program. Therefore, to optimize the single-directional 3-D camera system with faster processing speed and maintain the same accuracy, re-design the previous decoding algorithm is necessary.

The existing serial decoding algorithm contains six steps shown as Figure 1.7. Summarizing these necessary steps, we can re-organize the decoding algorithm into two major computation steps shown in table 3.2.

1) Sort the point clouds to form a list of nine shortest distance neighbor points. This list has the information of each point's 2-D coordinates (x,y) and Orientation information 'o'.
2) Perform angle calculation within each group in the sorted list from step 2. Use these angels and the previous generated soring list to search the necessary points' coordinates and form the codeword list for the next step Correspondence match.

Table 3.2 Decoding Implementation steps

A efficient sorting approach can lead to a major improvements on computation speed of the entire decoding algorithm. It is important for optimizing the use of other algorithms in the following steps. Analyzing these two steps of Decoding, the first step, sorting, takes up the most

time of decoding due to serial computation of large scaled data information. Therefore, implementing an efficient sorting algorithm which can be implemented to meet the speed and accuracy requirement is important.

There are many data sorting algorithms to find paths with the minimum costs for point clouds. As described the sorting purpose of step 1 in Table 3.2, the k-nearest neighbor algorithm (k-NN), a method for classifying objects based on closest training examples in the feature space, is the technique we decide to implement for our application.

3.3.1 k-NN Algorithm

K-nearest neighbor (k-NN) has been widely applied in the performance in classification applications in terms of computational complexity or accuracy, such as decision tree, neural network, etc. It is a machine learning algorithms which an object is classified by a majority vote of k neighbors, where k is a positive number. For example, if k=1, then the output is simply the result of each object's nearest neighbor among all the class.

This algorithm is commonly applied to a small size of database. To find k nearest neighbor for each target object, k-NN algorithm has to scan all the objects in the reference data points. If there is n target objects, and looking for m nearest neighbors for each object, there are $m \times n$ computation steps needed for the performance. Because of the complexity of the serial k-NN computation with CPU, the larger the database is the slower the application response. The challenge of this research is to fast implement the decoding process with large-scale datasets.

Therefore, slow response of k-NN computation with large-scale database needs to be improved. Parallel computing with GPU will help us to resolve this issue. In section 3.22, we will present a efficient parallel implemented sorting algorithm with k-NN to accelerate the computation process of the 3-D single-directional camera system.

The k-NN algorithm computes in the following procedures:

1) For each target object m, compute Euclidean distance from target object m to the others in the reference database.
2) Order samples by calculated distance
3) Choose optimal k closest points to target object m

3.3.2 Decoding Implementation

The input data structure for the decoding function is a three dimensional array which contains each marker's camera coordinates (x,y) and an orientation 'o' generated from the previous image processing function. In C++ serial programming, these data are defined as vectors (x,y,o). Even though 'o' does not represent dimension z for depth information, in the parallel decoding algorithm, we still look at the data information as three dimensional data structure because we need to use all three pieces of information to perform the decoding algorithm.

CPU serial decoding algorithm and implementation

The previous sequential decoding algorithm in CPU is first sequentially find the nearest nine points of each point, then put these nine points into a 3*3 matrix, name them mask [0] to mask [8] (mask [0] represents the original point), then calculates the angle from the rest 8 points to the original point mask [0]. Compare these angle to nine pre-defined angle positions "45°, 225°, 90°, 270°, 135°, 315°," to classify the codeword for the next function in the single-directional camera system algorithm. For any group of nine contains all nine angle positions, return a nine digit codeword built by their corespondent orientation value 'o'. Then pushback their respected x and y candidates with this codeword to form a decoding list for future use.

As described in previous chapter, if I can implement this decoding algorithm using GPU parallel architecture, the computation time would be faster. I can improve the whole application response speed. However, the question is if I just directly use this sequential algorithm in the GPU, will it utilize the usage of the GPU and will it fit for the GPU parallel architecture? The answer is no. The reasons of why it cannot be directly implemented as it is into GPU are demonstrated as follows:

1) GPU parallel computing is easy on implementing simple computation but not on many looping sequential calculations. When invoking the kernel, many threads are generated at the same time and calculate at the same time, each thread has to read all data points in the reference database to do the comparison with all the other points to find the 9 nearest points.

Therefore, it creates a heavy sequential calculation work flow for each thread. If the data is very large, the GPU might corrupt.

2) The data type cannot be defined as three dimensional in GPU, because in (x,y,o), o is not direction z. Therefore it is hard to use o to design the grid and block shape. However, o is key characteristic to create the decoding list, which means o cannot be ignored. Therefore each combination of (x,y,o) cannot be represented as an vector array, which increase the difficulties of this task.

Since the original sequential execution algorithm of C++ does not fit for GPU's parallel computing architecture. Therefore modification and re-design of the current arithmetic is needed. Keep the same logic but use different arithmetic and implement the application based on GPU parallel computing characteristic is the challenge.

GPU parallel decoding algorithm and implementation

The parallel decoding algorithm based on CUDA GPU architecture needs to be designed in the consideration of the following conditions:

1) Use each thread to calculate all the necessary calculation steps for one point. And allocate the number of threads based on the number of points. The number of threads has to be greater and equal than the number of points.
2) Change data type: build a structure. Instead of using 3-dimensional vector array, putting coordinates (x, y), orientation 'o' and angle ' θ ' into a structure, so that each of them can be

singly pulled from the structure to do calculation and do not conflicts each other. In addition, the shape of the grid and the block can be designed using only the dimension size of the x and y. “o” can be ignored at this step, so it won’t affect any future calculation.

3) Define a structure which contains 9 nearest points which defined as a neighbor. Give up the 3*3 matrix, using k-nearest neighbor algorithm (k = 9) to compute the distance and sort the points instead of decreasing the sequential calculation loops for each thread.
4) Use less data to try out in simple block, and then implement large scaled data in multi-blocks. Modification of the code might needed due to the heavy sequential calculation in each thread.
5) Since data in _global_ memory can be read both host and device freely, and in the computation process of decoding algorithm, the k-nearest neighbor sorting is the most time consuming computation process, and the final decoding list is only used to store the final output for the next programming function, to allocate the memory of the GPU, I put the final decoding list under _device_ memory and put the k-nearest neighbor sorting process under _global_memory. In this way, I can reduce the work flow for memory reading, in addition, reduce the kernel invoking time, in fact, accelerate the decoding implementation.
6) A timer needs to add to the program to calculate the time of invoking kernel and the time of transferring data back to the Host (CPU) from Device(GPU) to compare the computation time with the CPU serial decoding algorithm.

The designed parallel decoding algorithm is only implemented in two steps shown in figure 3.1. Comparing to the serial decoding algorithm, the complexity of the computing processing is decreased from 6 steps to 2 steps.

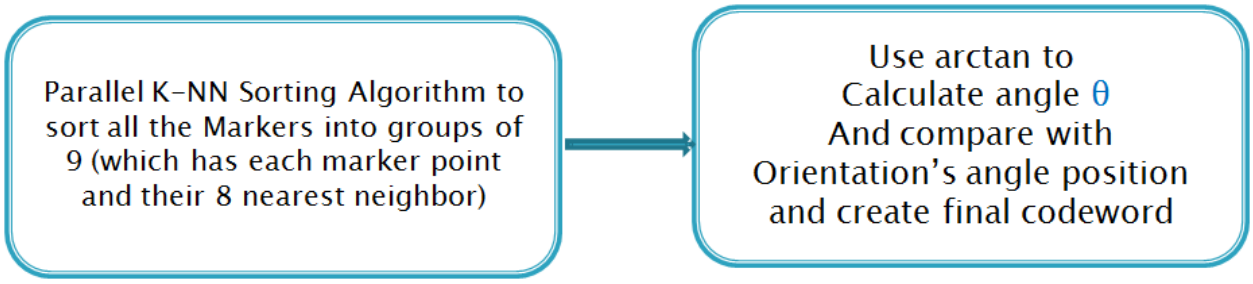


Figure 3.1 Decoding Algorithm

The computation of the distance can be calculated using the Euclidean Distance. The information of the reference data base is loaded from the global memory to the shard memory of each block. Each thread takes care of only one target point to compute the distance with the other points in the reference data sheet shared with the other threads. Therefore, the distance calculation can be parallelized computed using GPU. The number of threads need to be generated in this kernel depends on how many points in the database. It has to be greater or equal than the number of the reference points in the data sheet. This distances are stored in the shared memory of each block for future sorting kernel.

After computing the distances between each target point to the other points in the reference data sheet, I use k-NN algorithm to find the 8 nearest neighbors of the target point. Since I stored all the distances for each target point in the shared memory in each block, the sorting of 8 nearest neighbors for each target point can pull the distance data inside of its block. Therefore, time of

transferring data from memory is being saved. All the blocks invoke this sorting kernel concurrently. Also all the threads generated by the kernel in a common block performs the sorting algorithm simultaneously. This parallel k-NN sorting process can be shown in Figure 3.2.

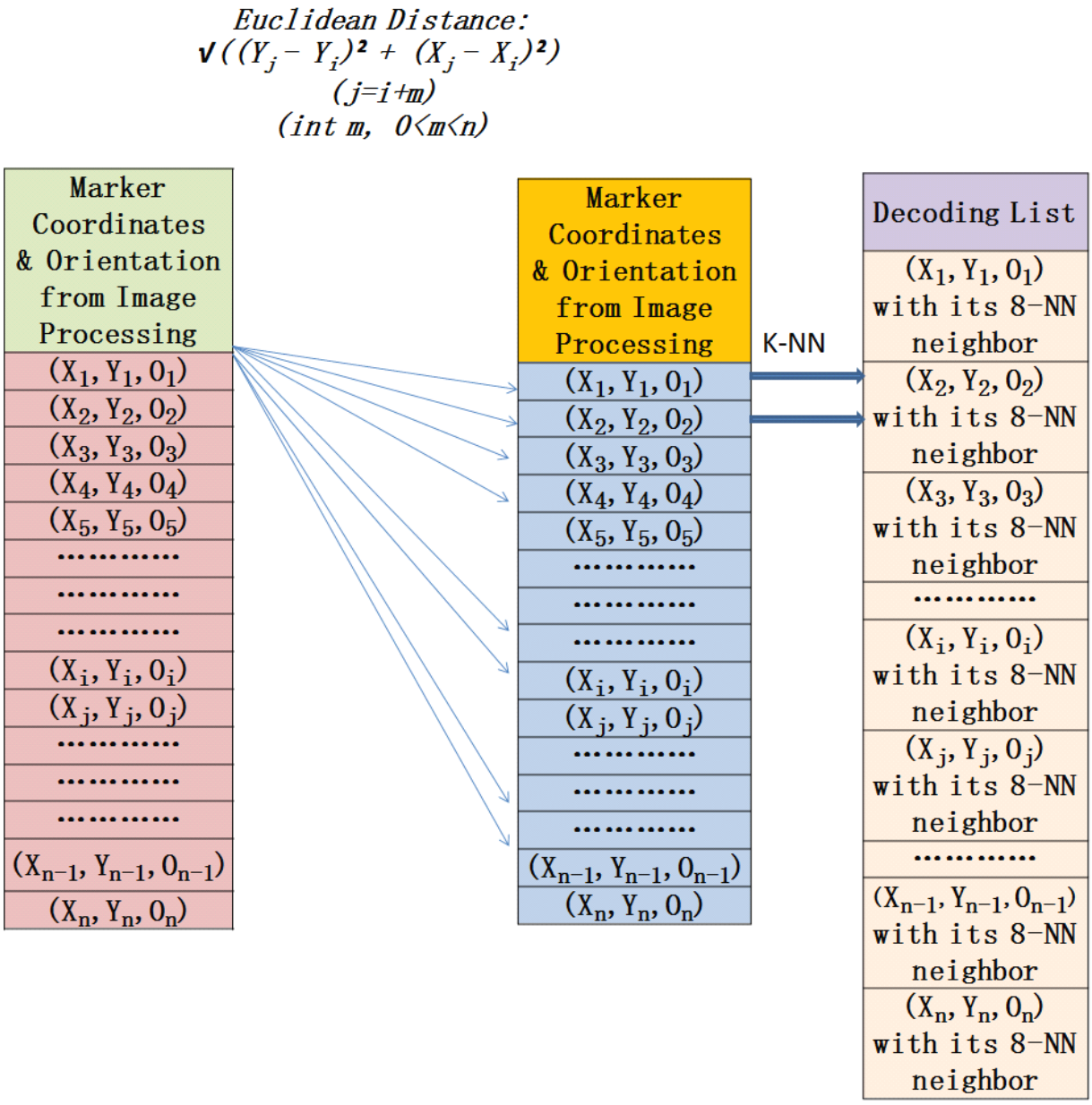


Figure 3.2 k-NN Sorting of Decoding Algorithm

In the k-NN sorting step, I have generated a data sheet contains groups of 9 nearest neighbors shown as in Figure 3.2. Then I perform the angle comparison sorting using arctan function. In terms of the standard arctan function, whose range is $(-\pi/2, \pi/2)$, I use atan2 function to prevent over eliminating the points in the range of $(-\pi, \pi]$.Compare these angle to the 9 positions defined by angle "45°, 225°, 90°, 270°, 135°, 315°." Each thread only handles one groups of 9-nearest neighbors. All threads in the same block and all blocks invoke this angle comparison kernel simultaneously in the GPU. After the result is obtained, I return a nine digit codeword built by their corespondent orientation value 'o'. The final decoding list will contain groups of 9-nearest neighbor's x and y coordinates and the nine digit codeword in each row. And this list is stored in the _device_memory. Whenever this list is needed, it can be pulled and transferred into the Host (CPU) memory for future use. Since the data is large, it saves the memory space of the CPU to do other six computations for this 3-D single-directional camera system.

	Decoding w/ CPU (s)	Total Application w/ CPU (s)	% (Decoding/Total Programing)
Image 1	8.7	13.3	65.4%
Image 2	7.1	12.6	56.3%
Image 3	5.3	10.8	49.1%
Image 4	9	14.8	60.8%
Image 5	8.6	15.8	54.4%
Image 6	6	10.2	58.8%
Image 7	7.1	11.9	59.7%
Image 8	7.3	12.1	60.3%
Image 9	5.1	10.4	49.0%
Image 10	7.2	12.4	58.1%
Image 11	8.4	12.8	65.6%
Image 12	7.3	12.1	60.3%
Image 13	8.5	15.3	55.6%
Image 14	8.3	14.9	55.7%
Image 15	8.2	15.8	51.9%
Image 16	7.6	12.8	59.4%
Image 17	7.3	12.2	59.8%
Image 18	7.9	13.3	59.4%
Image 19	7.8	13.2	59.1%
Image 20	8.4	14.2	59.2%
Image 21	6	10.5	57.1%
Average	7.5	12.9	57.9%

Table 3.3 Time consuming comparison (Decoding vs. Complete Program Implementation)

Chapter 4

Experiments and Improvements

I captured 21 different images with different patterns from different angle in different light source environments with the 3-D single-directional Camera as experiments to prove the parallel decoding algorithm improvements. Comparing to the serial CPU decoding algorithm, our parallel decoding algorithm efficiently utilizes CUDA-based GPU program structure to speed up the decoding process. In fact, as demonstrated in Section 3.3, decoding is the most time consuming function among all seven functions, it majorly speeds up the whole 3-D single-directional Camera system total implementation time.

Table 4.1 and Table 4.2 shows the comparison of the computation time with parallel decoding algorithm using GPU and serial decoding algorithm using CPU for 21 different images. Our parallel decoding algorithm with CUDA based GPU speeds up in the range from 770 to 2149 times of the original serial decoding algorithm with CPU.

From Figure 4.1 and Figure 4.2, I can see that no matter the kernel invoke 128 threads/block or 512 threads/block, the computation in our parallel decoding algorithm with CUDA-based GPU is way faster than the computation in serial decoding algorithm with CPU. Calculating the average time usage in each algorithm, Figure 4.3 shows the case if each kernel generates 128

threads/block , the decoding algorithm will take about 3.85 ms which speeds up about 1942 times, however, if each kernel generates 512 threads/block , it will take about 8.27 ms which speeds up about 905 times. Implement the parallel decoding algorithm using 128 threads/block is about twice faster than using 512 threads/block. This is because when I perform the k-NN sorting algorithm, each thread in the common block has to sequentially look for every other elements inside the block to sort the distances. And each thread handles only one point and perform this sequential k-NN sorting inside of the common block. Therefore, the less points inside a block, the less sequential computation need to be implemented inside the common block, the more blocks can be used to perform parallel computing using GPU, the faster the program can be implemented.

Since our parallel decoding algorithm has such significant improvement in computation speed and decoding is the most time consuming function in the whole application program, the total application implementation time should be optimized significantly as well. Table 4.3 and Table 4.4 shows how much times it improved for each experience.

From Figure 4.4, I can tell that no matter 128 threads/block or 512 threads/block each kernel can generate, the total application time of the single-directional camera system can be saved about 7.5 sec which speeds up the whole system about 1.7 times, in fact, almost speed up twice.

kernel invoke : 128 threads/block			
Image #	Serial Decoding w/ CPU (s)	Parallel Decoding w/ GPU (ms)	Parallel Decoding vs. Serial Decoding (# of Times Faster)
1	8.70	4.047776	2149
2	7.10	3.644288	1948
3	5.30	2.975008	1782
4	9.00	4.092448	2199
5	8.60	4.291360	2004
6	6.00	3.576512	1678
7	7.10	3.597312	1974
8	7.30	3.357568	2174
9	5.10	2.933344	1739
10	7.20	4.020896	1791
11	8.40	3.917984	2144
12	7.30	4.071840	1793
13	8.50	4.009440	2120
14	8.30	4.195232	1978
15	8.20	3.982304	2059
16	7.60	3.959104	1920
17	7.30	3.890592	1876
18	7.90	4.229504	1868
19	7.80	4.234304	1842
20	8.40	4.347168	1932
21	6.00	3.510752	1709

Table 4.1 Time comparison with 128 threads/block

kernel invoke : 512 threads/block			
Image #	Serial Decoding w/ CPU (s)	Parallel Decoding w/ GPU (ms)	Parallel Decoding vs. Serial Decoding (# of Times Faster)
1	8.70	9.484000	917
2	7.10	8.088288	878
3	5.30	3.942016	1344
4	9.00	9.363200	961
5	8.60	9.205408	934
6	6.00	7.046048	852
7	7.10	7.367872	964
8	7.30	5.888064	1240
9	5.10	3.884032	1313
10	7.20	9.428192	764
11	8.40	9.248288	908
12	7.30	9.475680	770
13	8.50	9.268384	917
14	8.30	9.394784	883
15	8.20	9.363968	876
16	7.60	9.222912	824
17	7.30	8.713696	838
18	7.90	9.430720	838
19	7.80	9.424864	828
20	8.40	9.480160	886
21	6.00	6.846112	876

Table 4.2 Time comparison with 512 threads/block

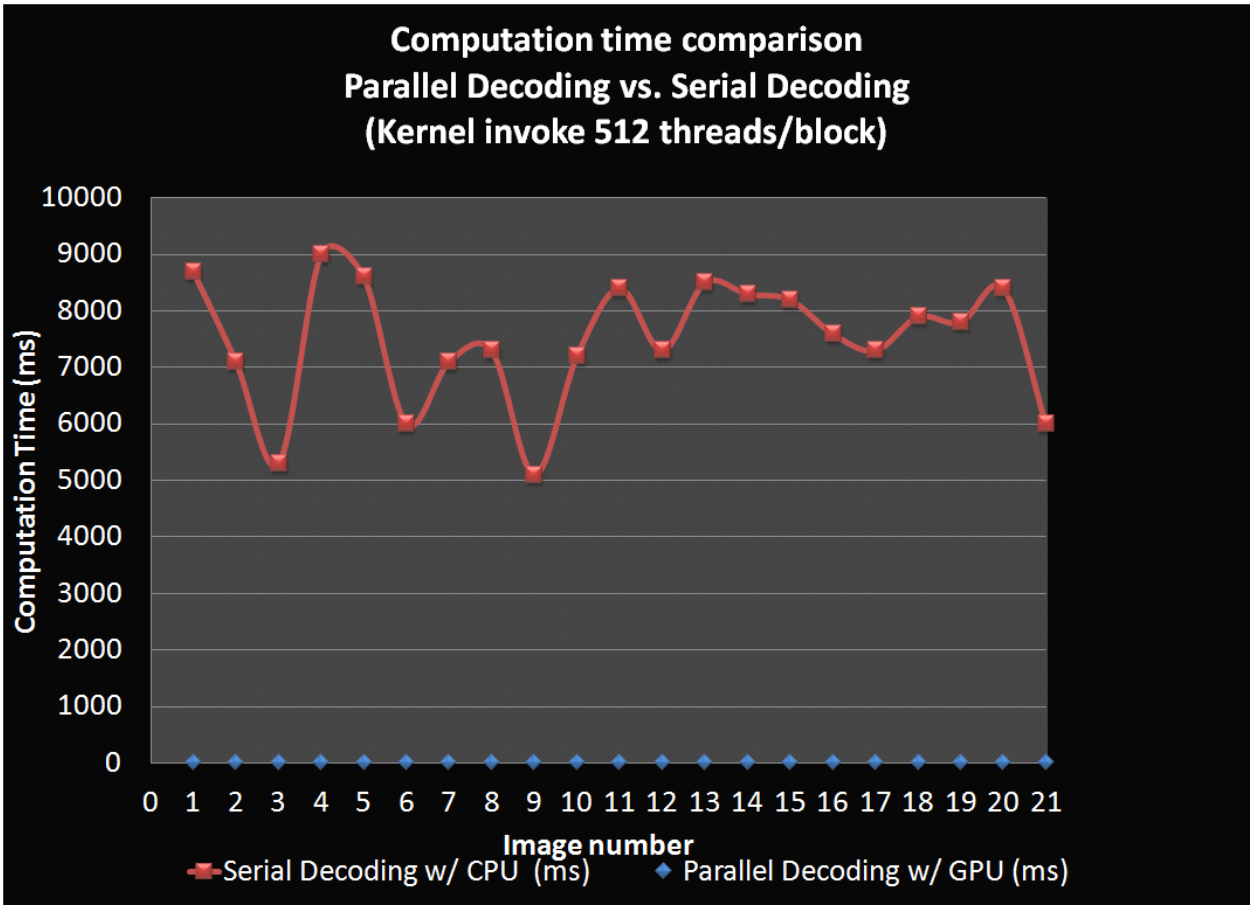


Figure 4.1 computation time comparison (128 threads/block)

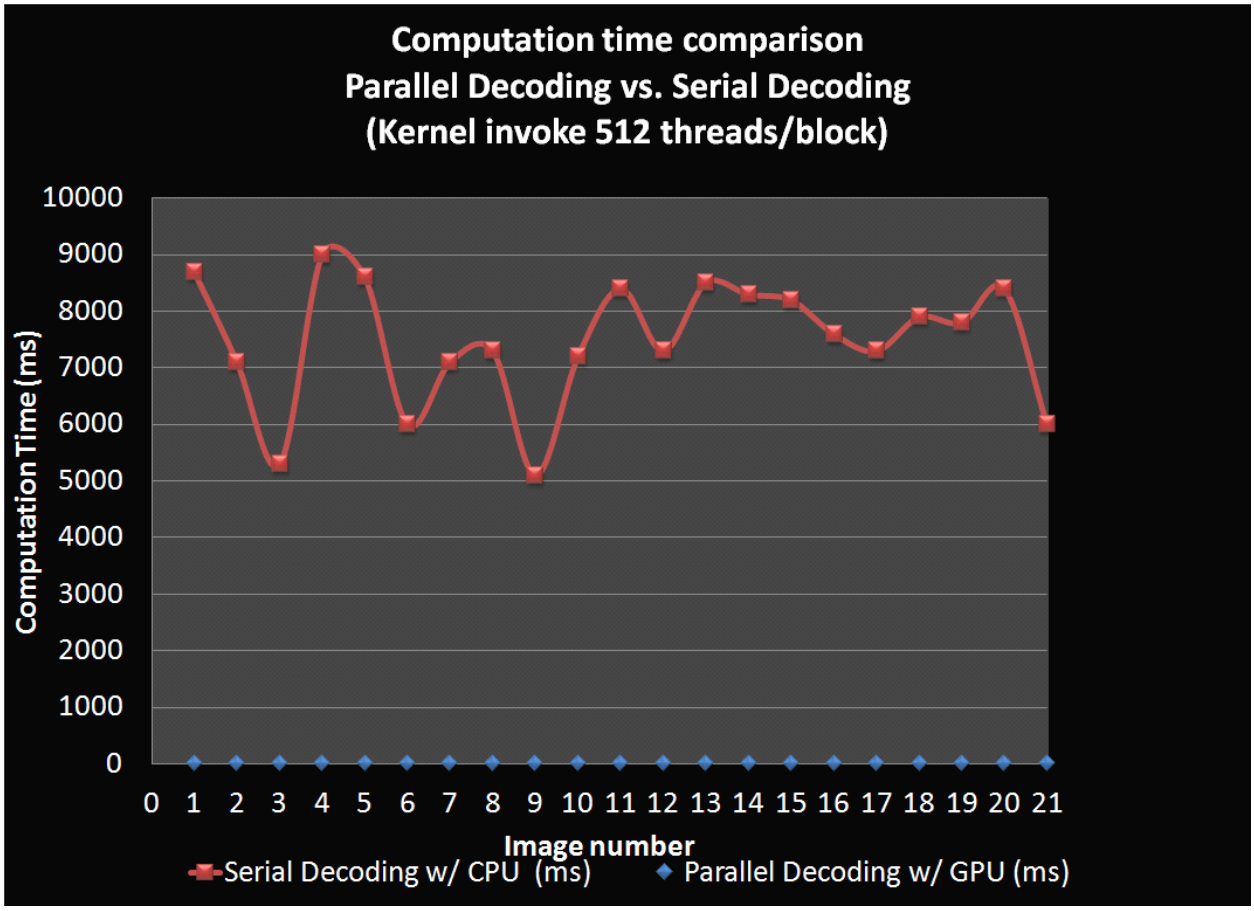


Figure 4.2 computation time comparison (512 threads/block)

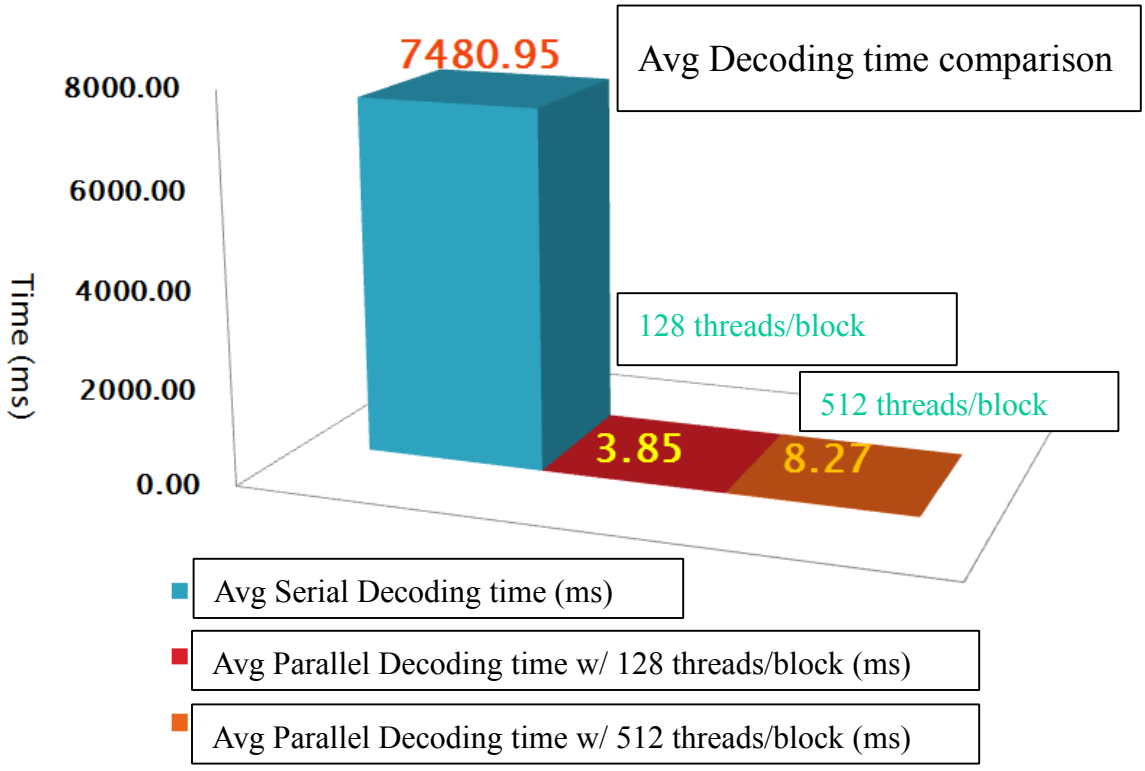


Figure 4.3 Average Computation Time Comparison

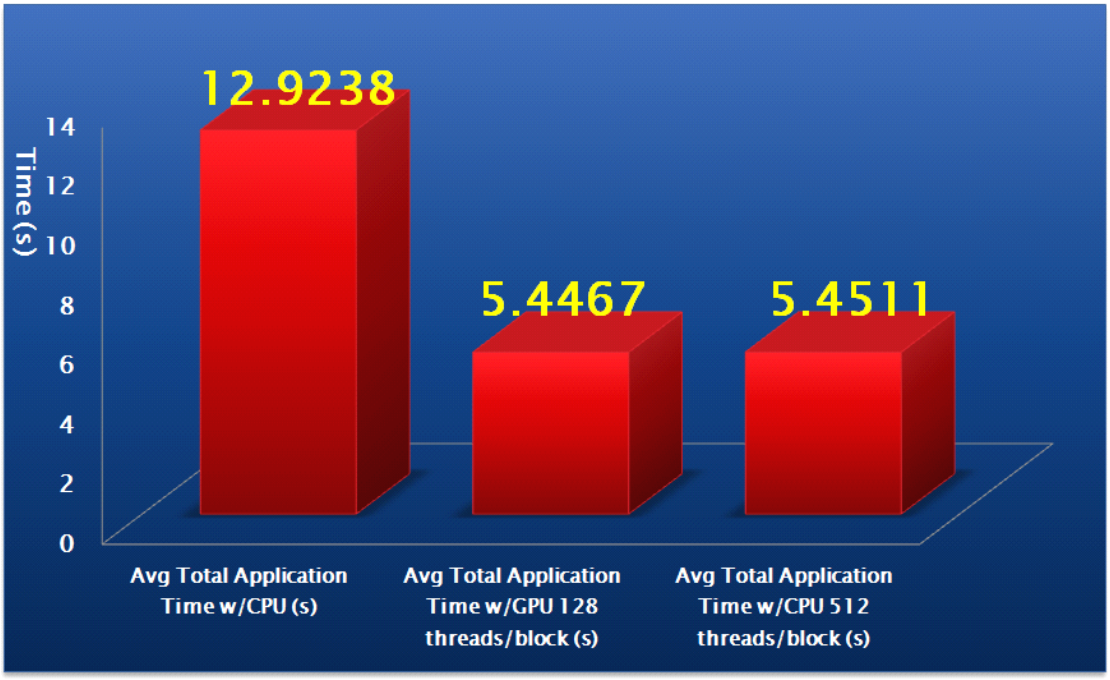


Figure 4.4 Total Application Performance Time Comparison

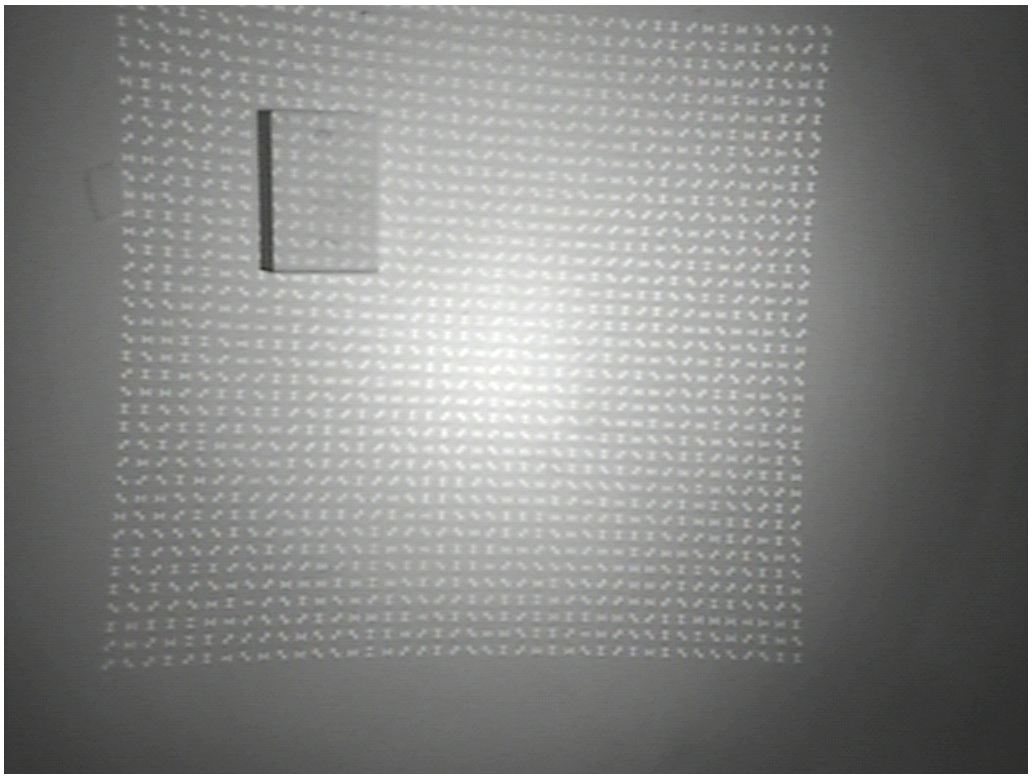
kernel invoke : 128 threads/block				
Image #	Total Application Time w/ CPU (s)	Total Application Time w/ GPU (s)	Total Application w/ Parallel Decoding vs. w/ Serial Decoding (Time saving in sec)	Total Application w/ Parallel Decoding vs. w/ Serial Decoding (# of Times Faster)
1	13.30	4.604	8.696	2.9
2	12.60	5.504	7.096	2.3
3	10.80	5.503	5.297	2.0
4	14.80	5.804	8.996	2.5
5	15.80	7.204	8.596	2.2
6	10.20	4.204	5.996	2.4
7	11.90	4.804	7.096	2.5
8	12.10	4.803	7.297	2.5
9	10.40	5.303	5.097	2.0
10	12.40	5.204	7.196	2.4
11	12.80	4.404	8.396	2.9
12	12.10	4.804	7.296	2.5
13	15.30	6.804	8.496	2.2
14	14.90	6.604	8.296	2.3
15	15.80	7.604	8.196	2.1
16	12.80	5.204	7.596	2.5
17	12.20	4.904	7.296	2.5
18	13.30	5.404	7.896	2.5
19	13.20	5.404	7.796	2.4
20	14.20	5.804	8.396	2.4
21	10.50	4.504	5.996	2.3

Table 4.3 Total Application Time comparison w/ 128 threads/block

kernel invoke : 512 threads/block				
Image #	Total Application Time w/ CPU (s)	Total Application Time w/ GPU (s)	Total Application w/ Parallel Decoding vs. w/ Serial Decoding (Time saving in sec)	Total Application w/ Parallel Decoding vs. w/ Serial Decoding (# of Times Faster)
1	13.30	4.609	8.691	2.9
2	12.60	5.508	7.092	2.3
3	10.80	5.504	5.296	2.0
4	14.80	5.809	8.991	2.5
5	15.80	7.209	8.591	2.2
6	10.20	4.207	5.993	2.4
7	11.90	4.807	7.093	2.5
8	12.10	4.806	7.294	2.5
9	10.40	5.304	5.096	2.0
10	12.40	5.209	7.191	2.4
11	12.80	4.409	8.391	2.9
12	12.10	4.809	7.291	2.5
13	15.30	6.809	8.491	2.2
14	14.90	6.609	8.291	2.3
15	15.80	7.609	8.191	2.1
16	12.80	5.209	7.591	2.5
17	12.20	4.909	7.291	2.5
18	13.30	5.409	7.891	2.5
19	13.20	5.409	7.791	2.4
20	14.20	5.809	8.391	2.4
21	10.50	4.507	5.993	2.3

Figure 4.4 Table 4.3 Total Application Time comparison w/ 512 threads/block

However, not only comparing the speed of the performance, I also compare the final output fusion images to prove the stability of the accuracy. I compared the output fusion images generated from the 3-D single-directional camera system using the serial CPU decoding algorithm to our parallel decoding algorithm using CUDA-based GPU. The results are all the output fusion images generates the same 3-D image coordinates information, which means the parallel decoding algorithm I designed not only optimizes the original serial decoding algorithm on the computation speed, but also maintains the same accuracy. Therefore, this is a efficient decoding algorithm. Each input images and the comparison of the output fusing images using both algorithms are shown below: from Figure 4.4.1 to Figure 4.4.21.



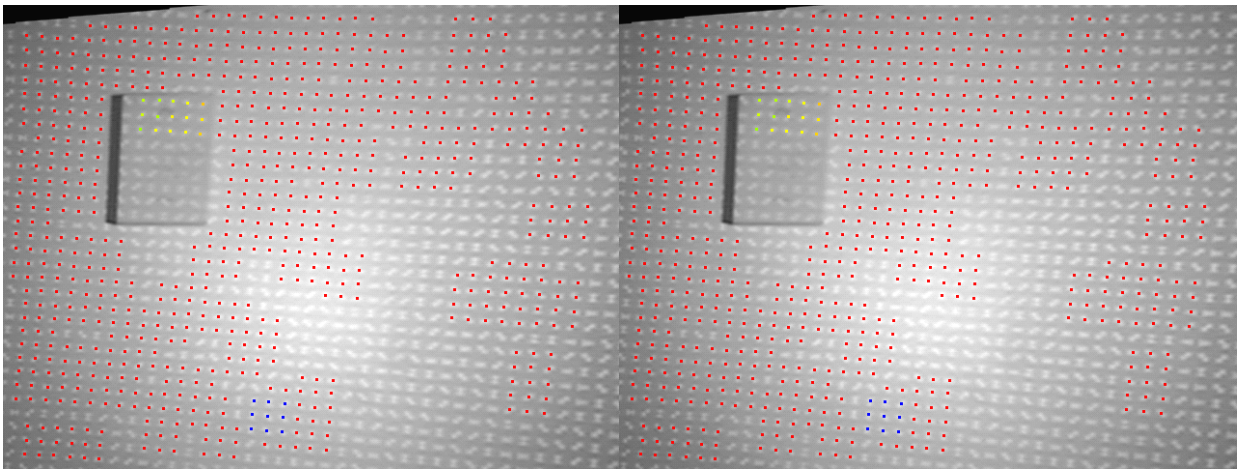
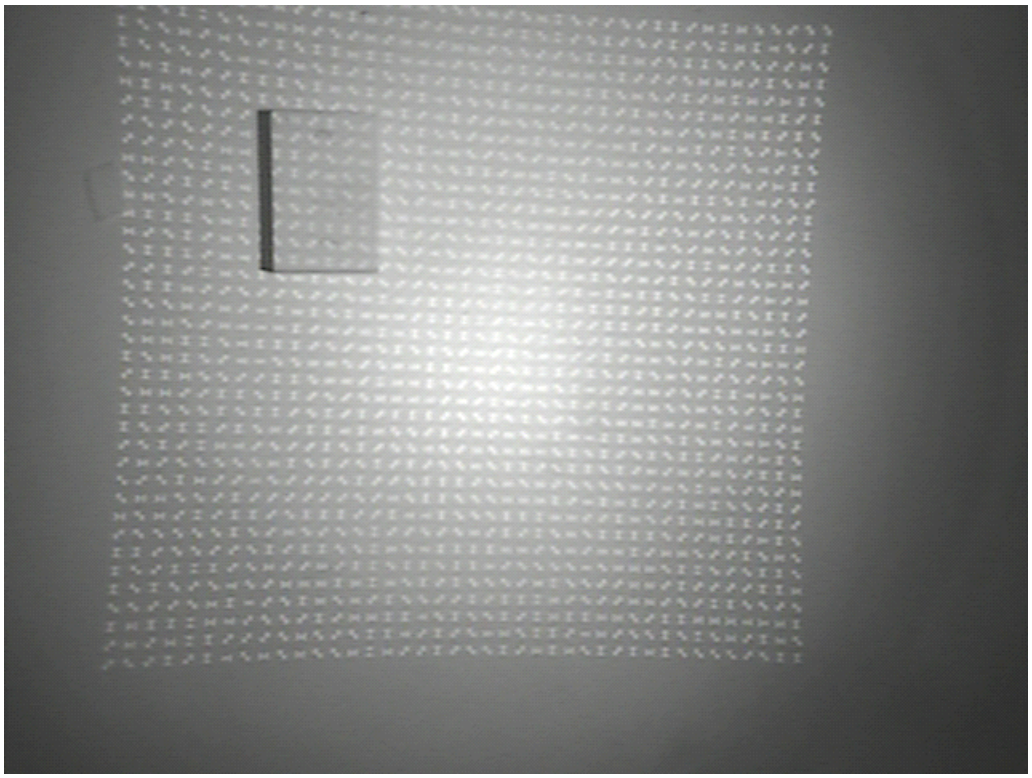


Figure 4.4.1 Input Image #1, Output image comparison



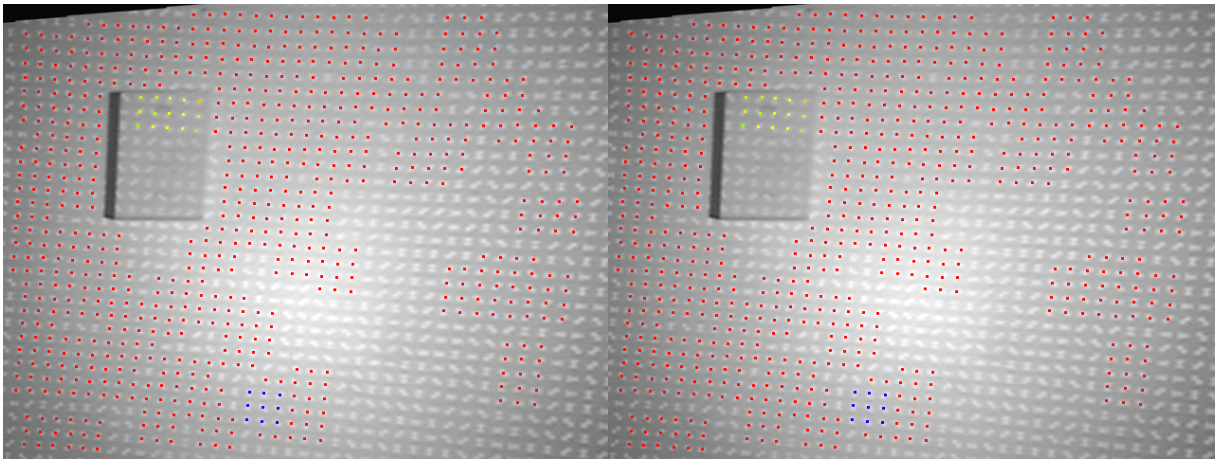
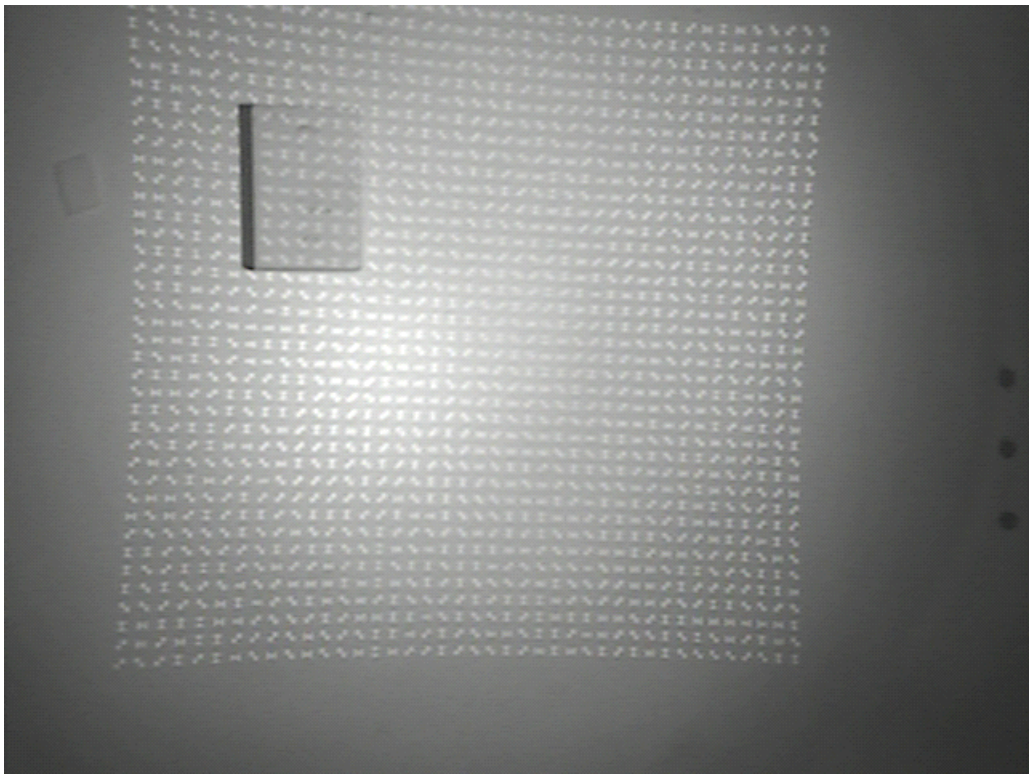


Figure 4.4.2 Input Image #2, Output image comparison



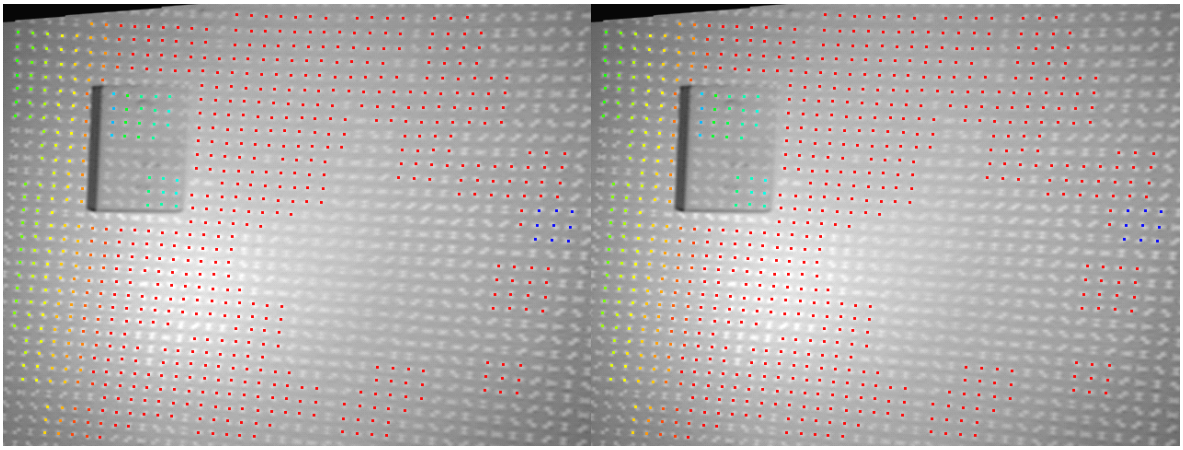
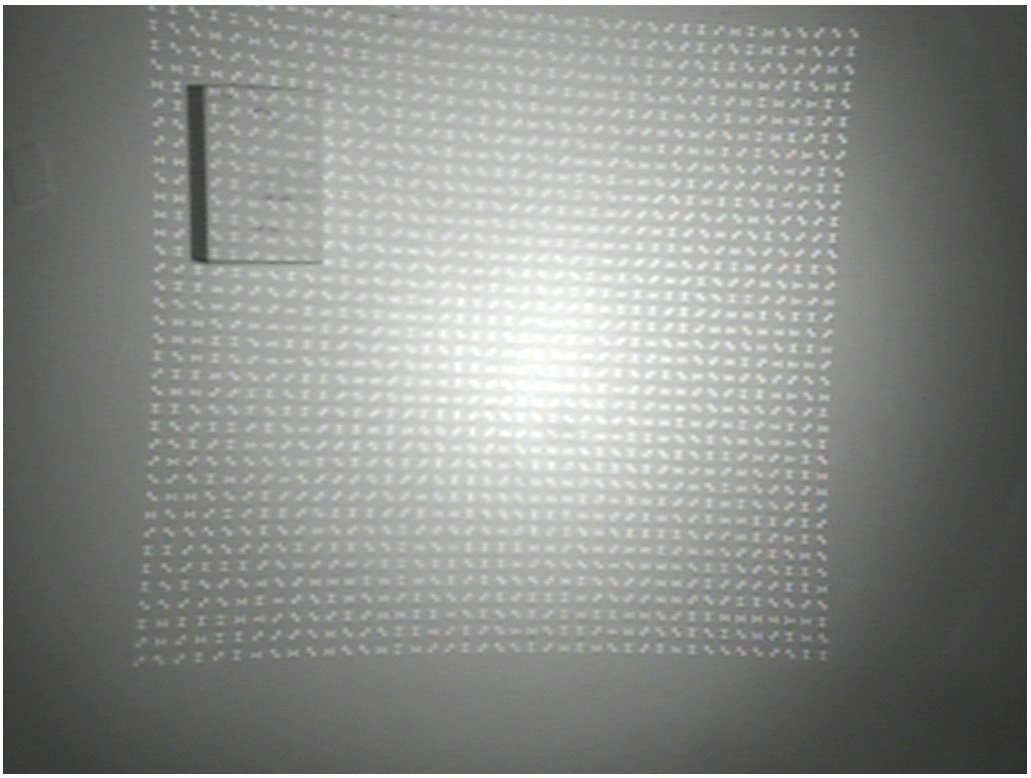


Figure 4.4.3 Input Image #3, Output image comparison



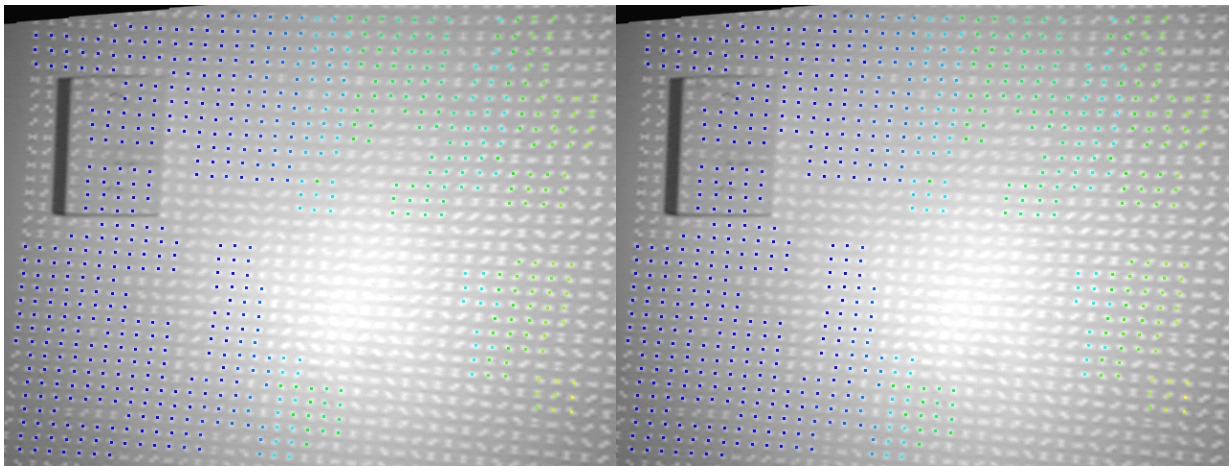
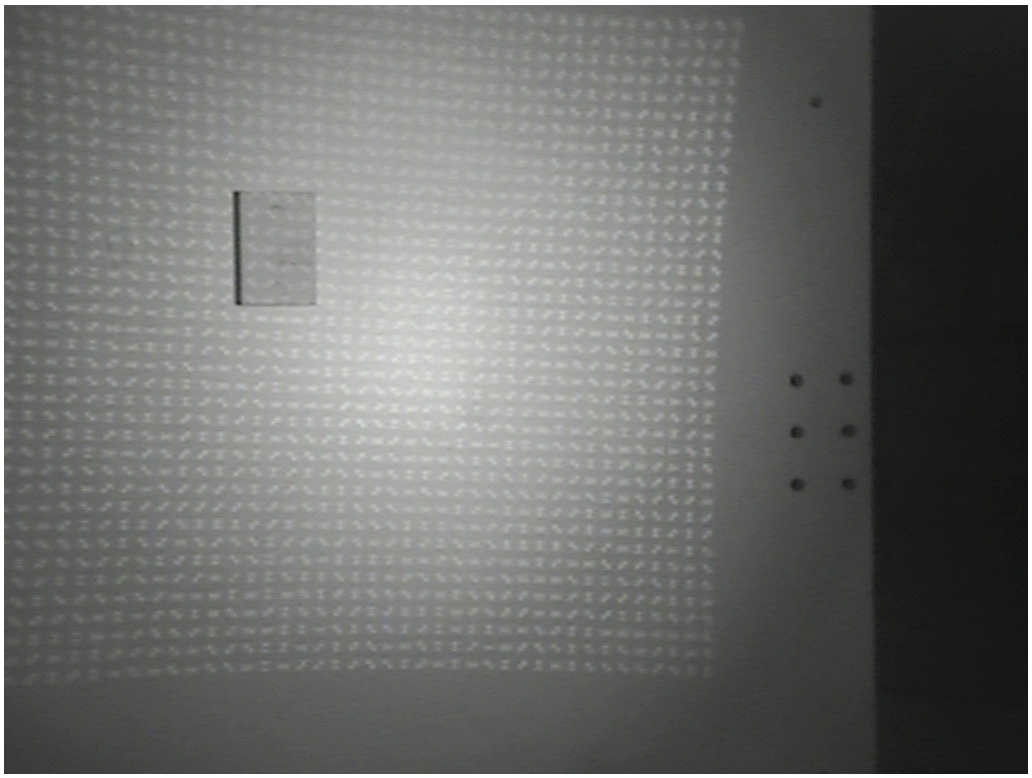


Figure 4.4.4 Input Image #4, Output image comparison



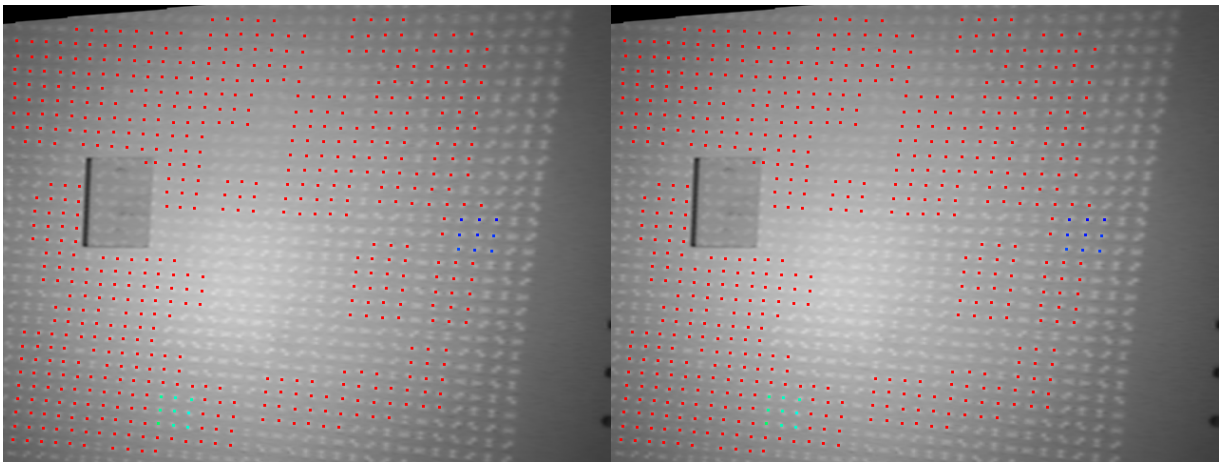
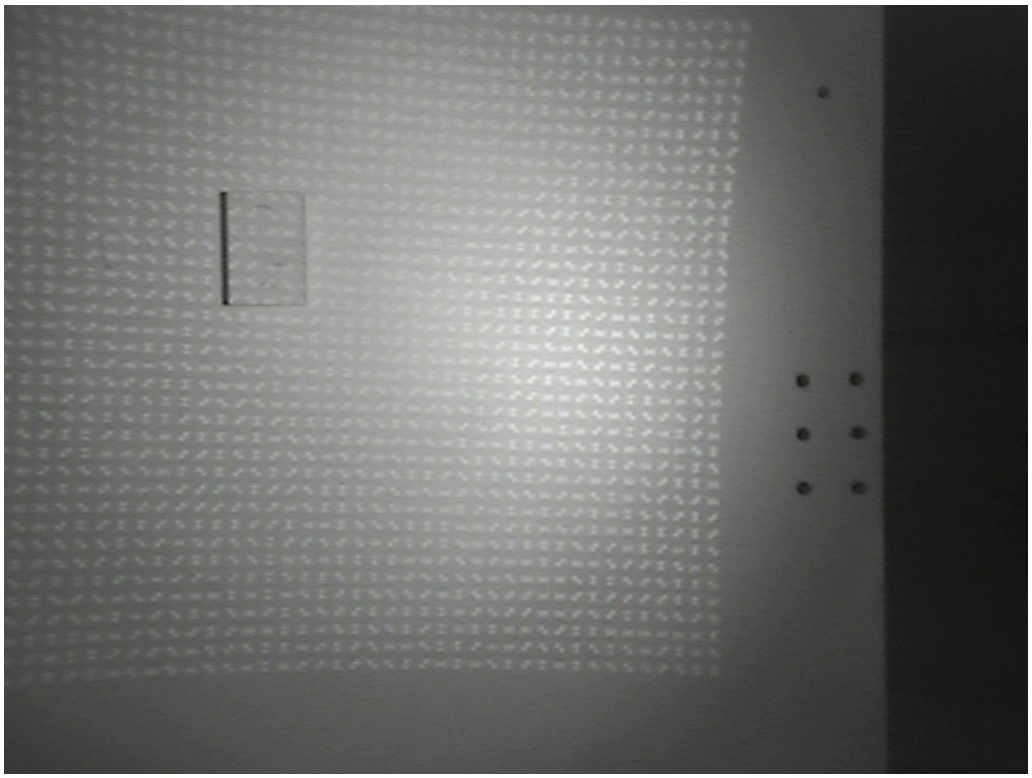


Figure 4.4.5 Input Image #5, Output image comparison



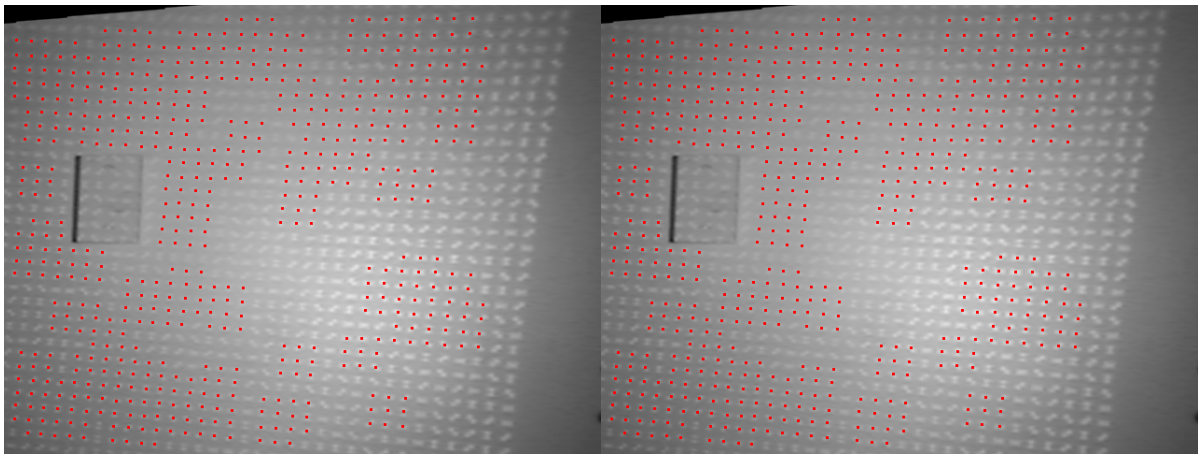
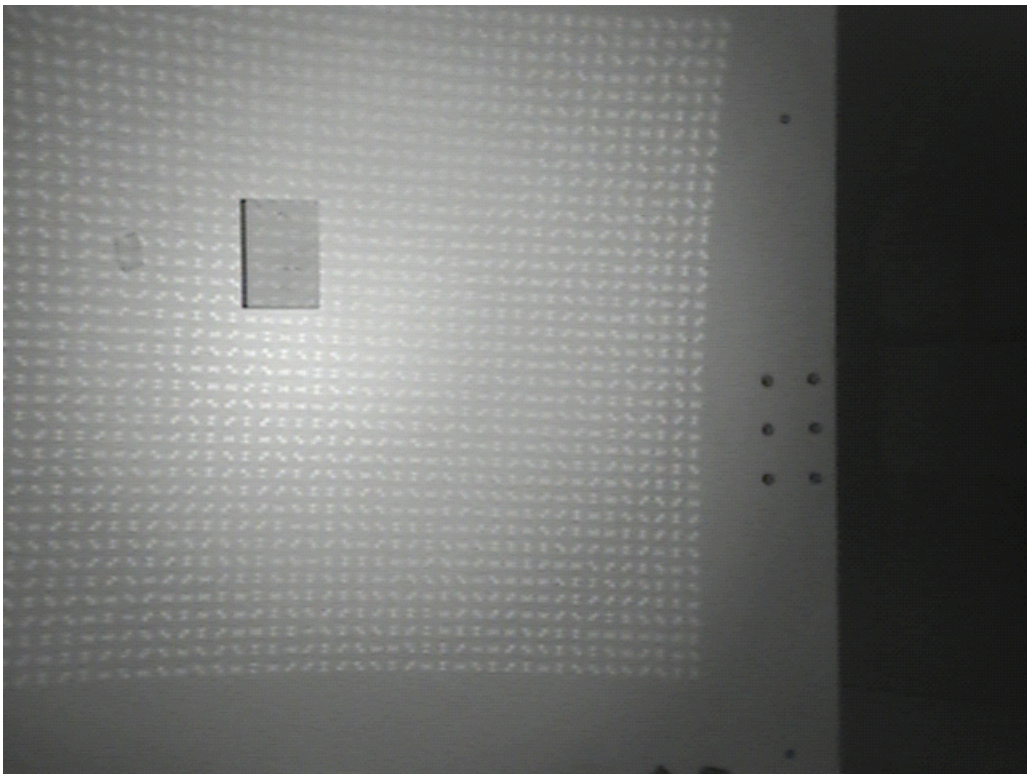


Figure 4.4.6 Input Image #6, Output image comparison



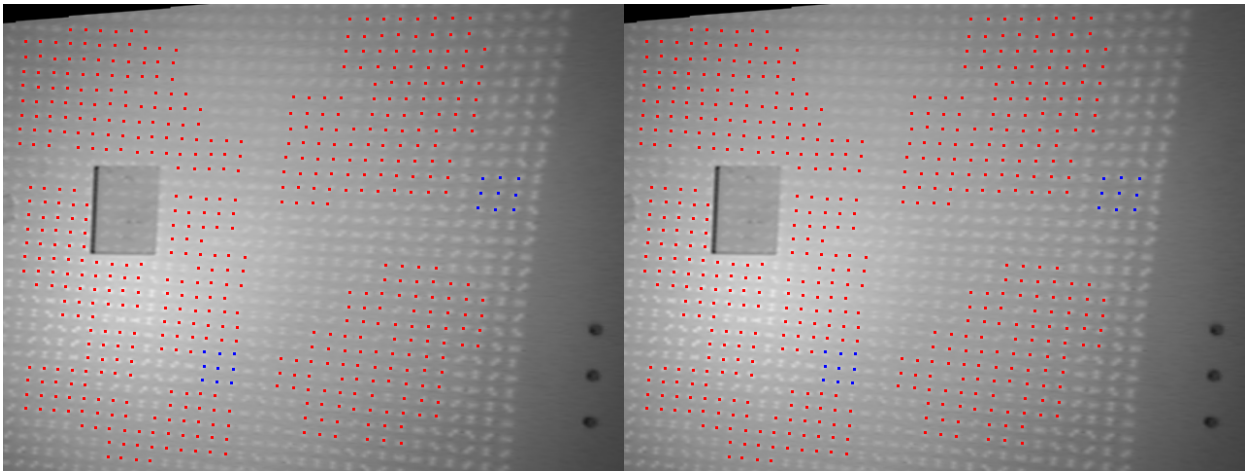
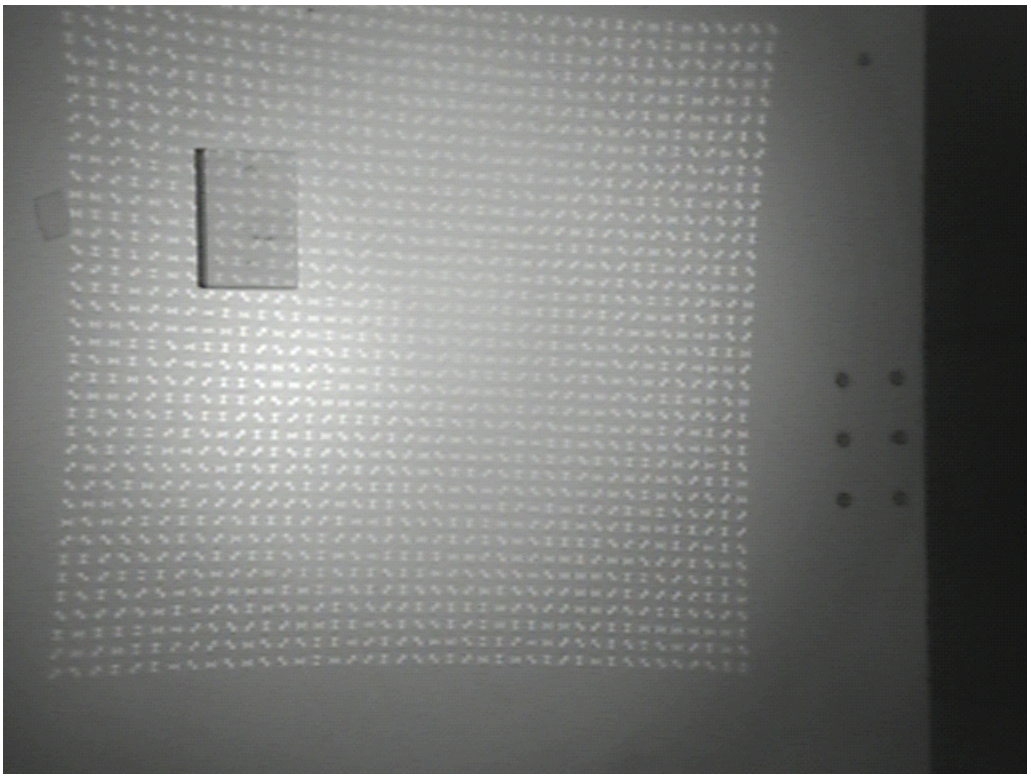


Figure 4.4.7 Input Image #7, Output image comparison



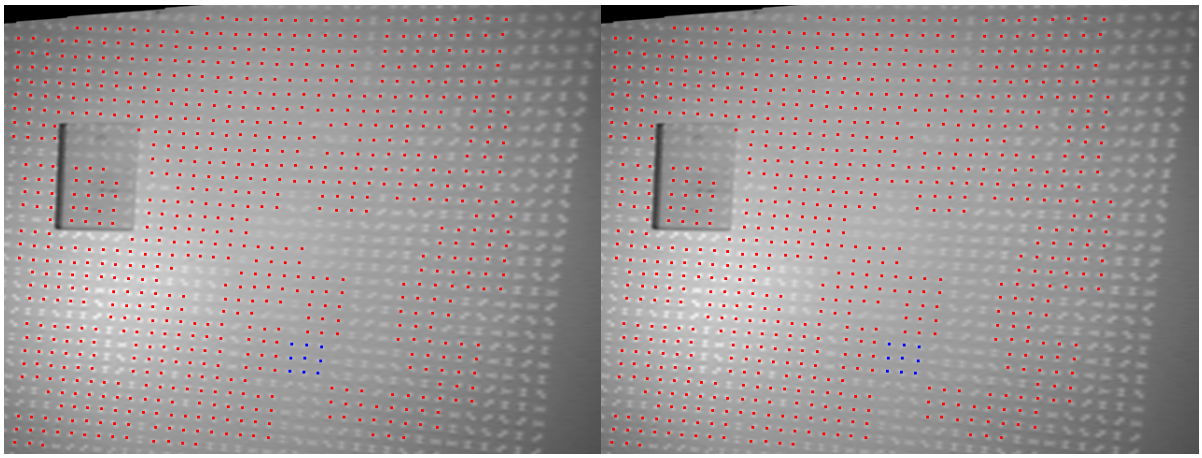
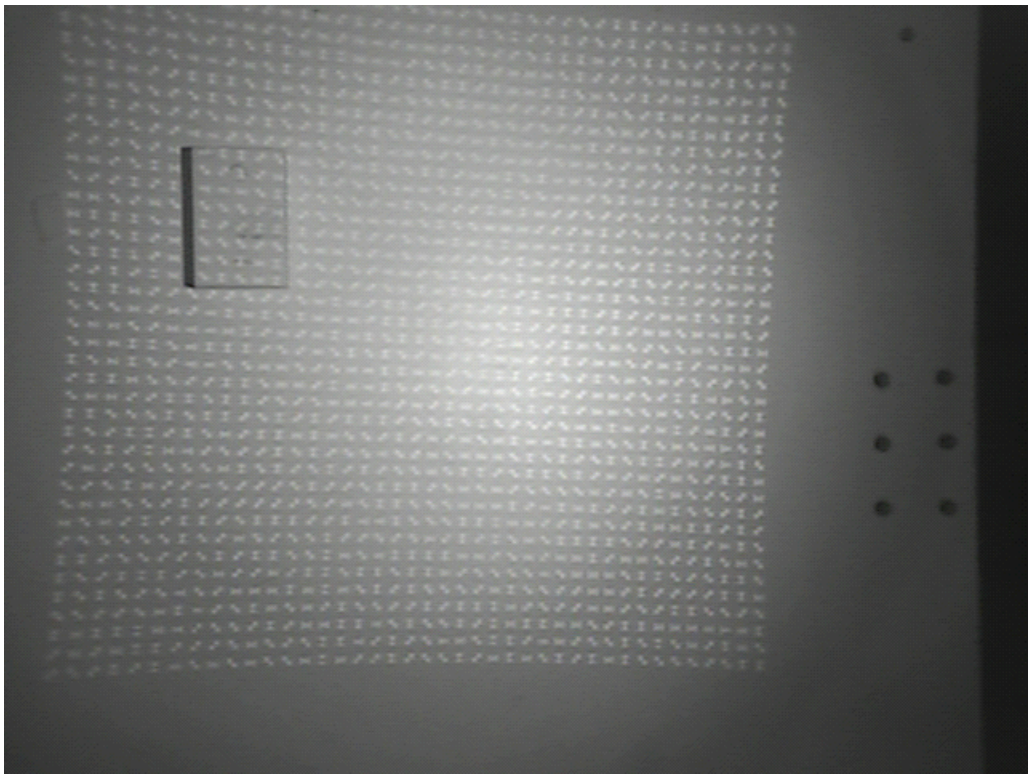


Figure 4.4.8 Input Image #8, Output image comparison



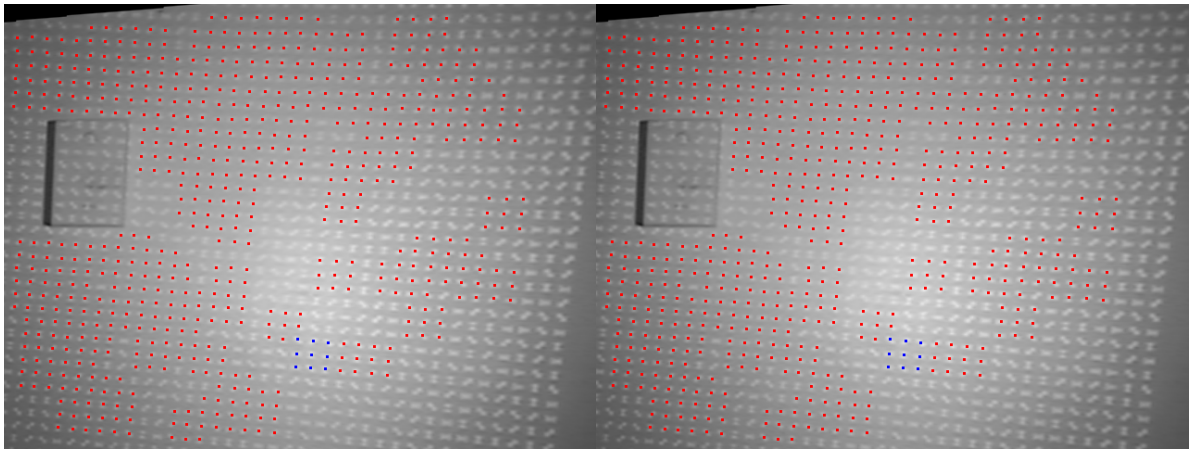
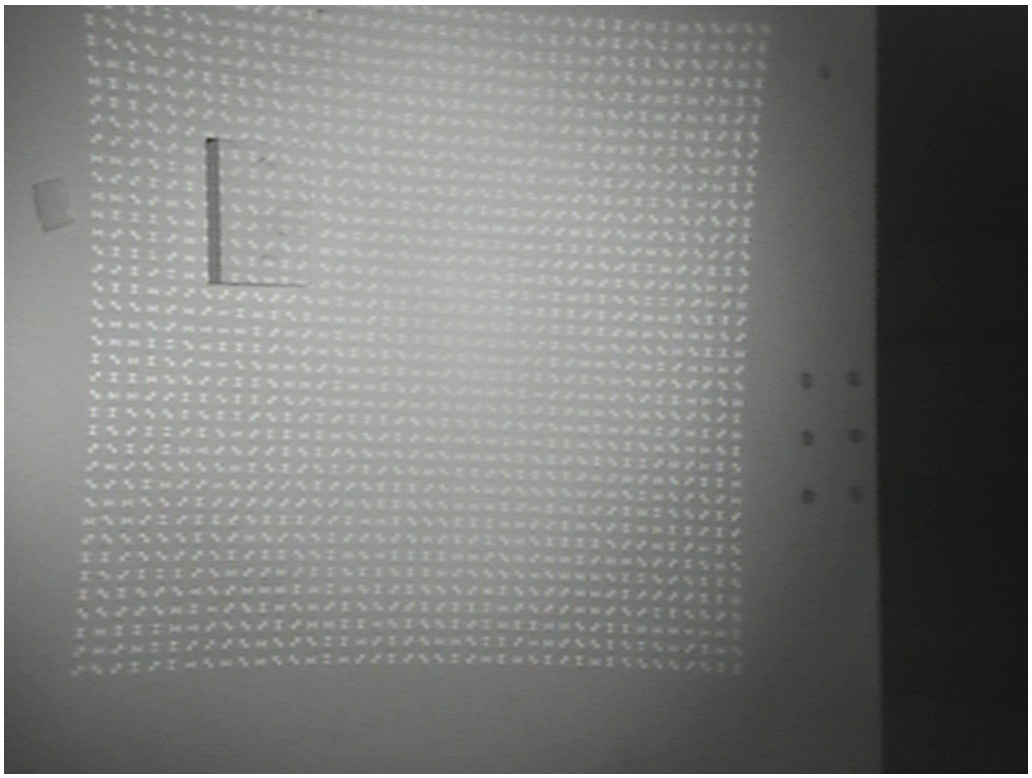


Figure 4.4.9 Input Image #9, Output image comparison



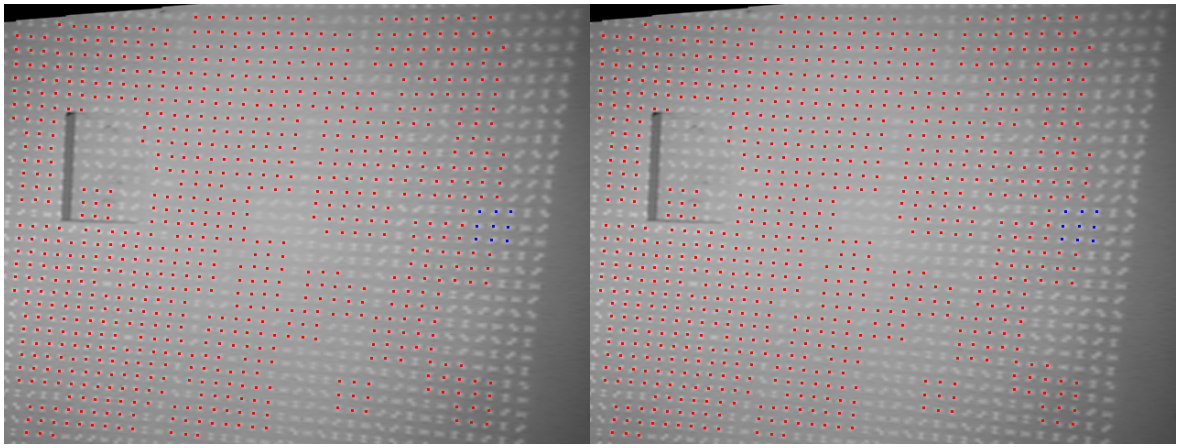
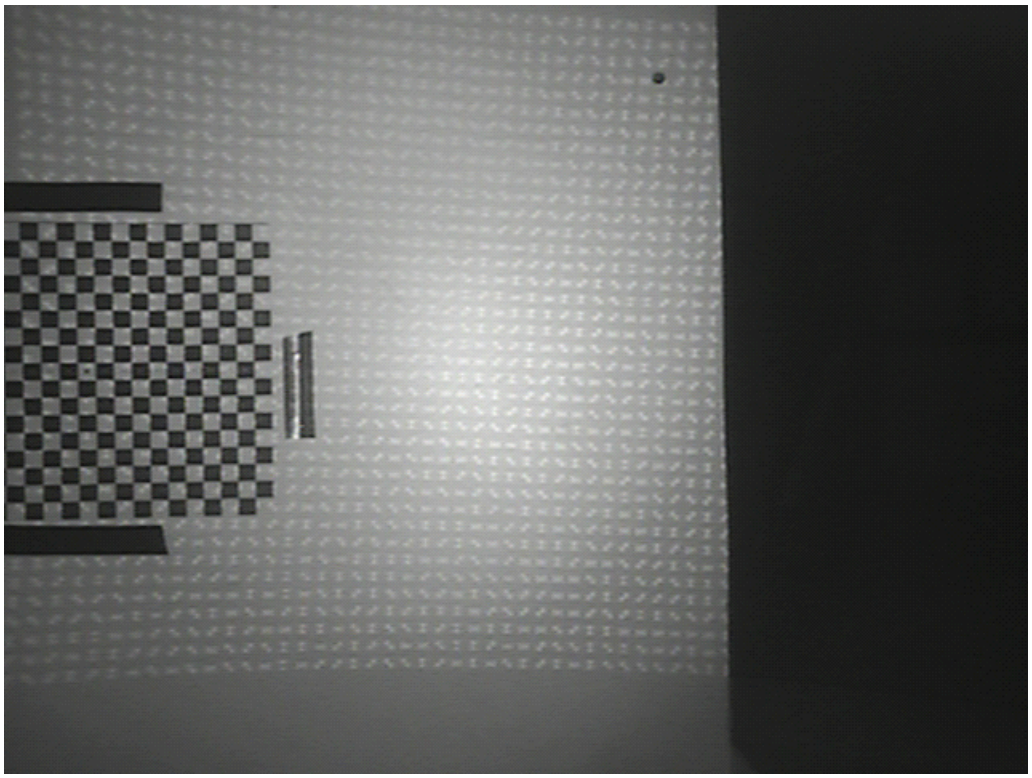


Figure 4.4.10 Input Image #10, Output image comparison



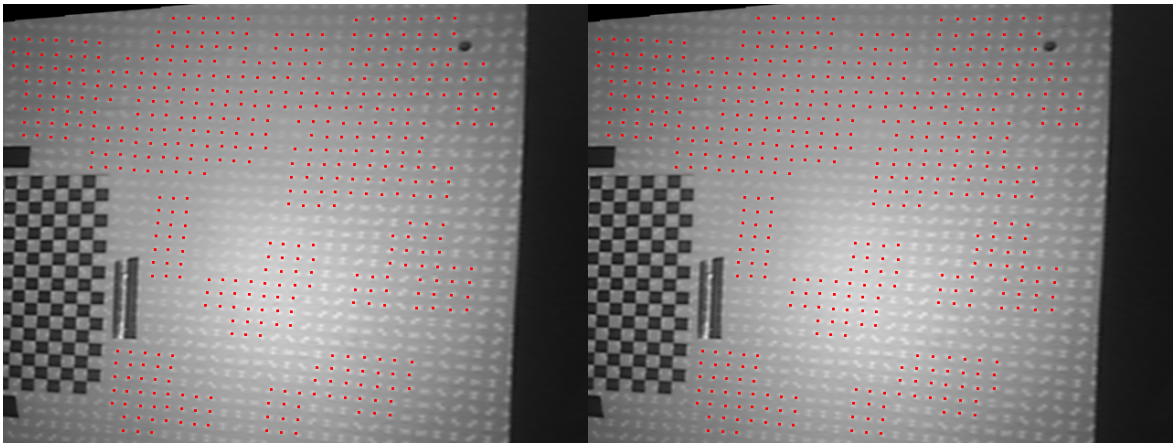
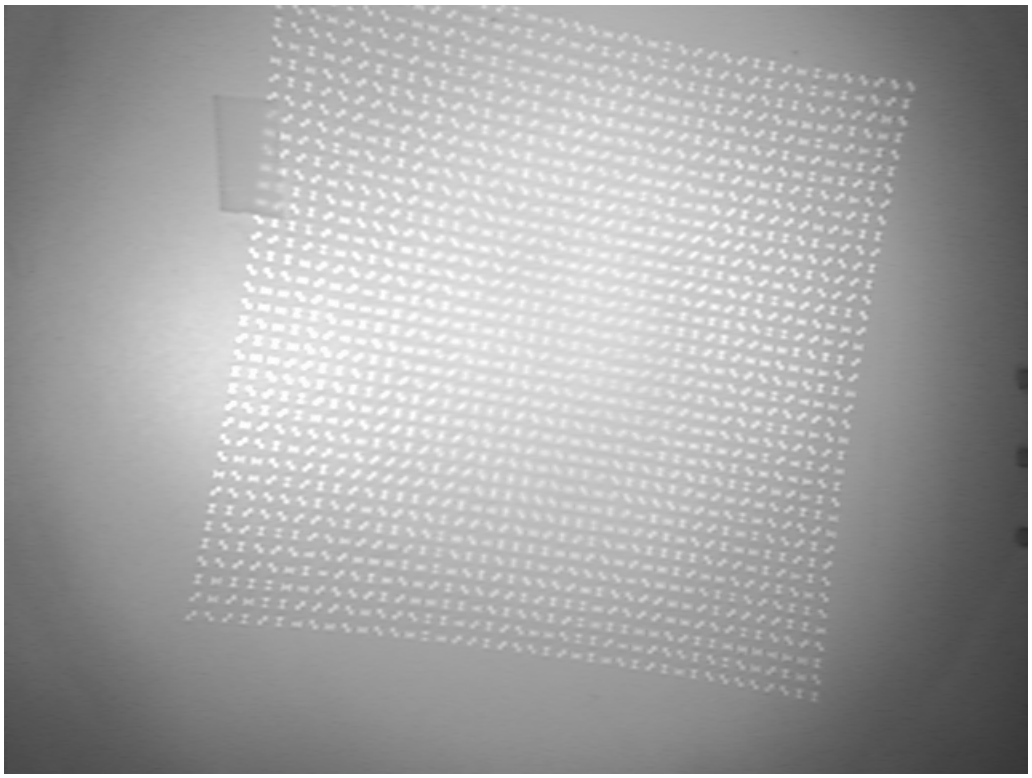


Figure 4.4.11 Input Image #11, Output image comparison



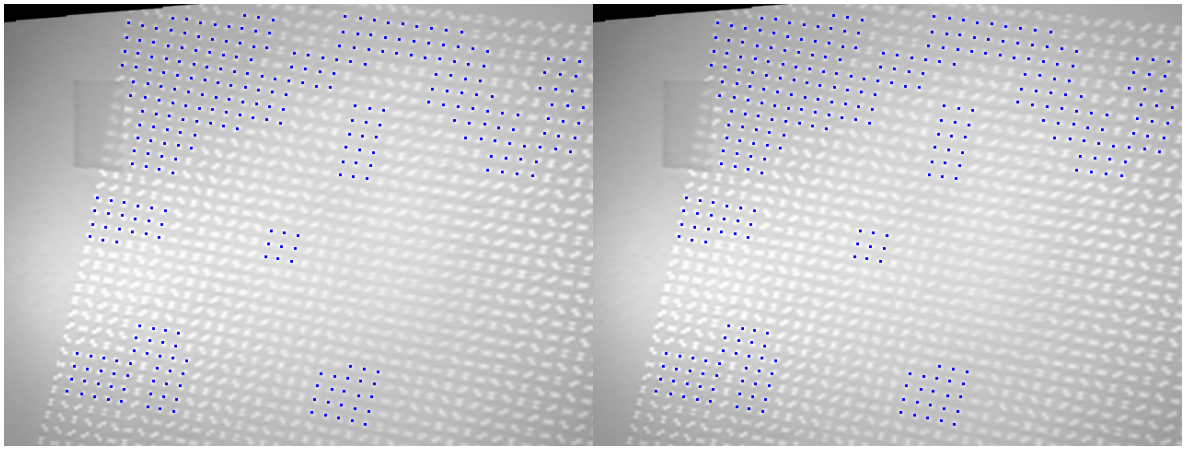
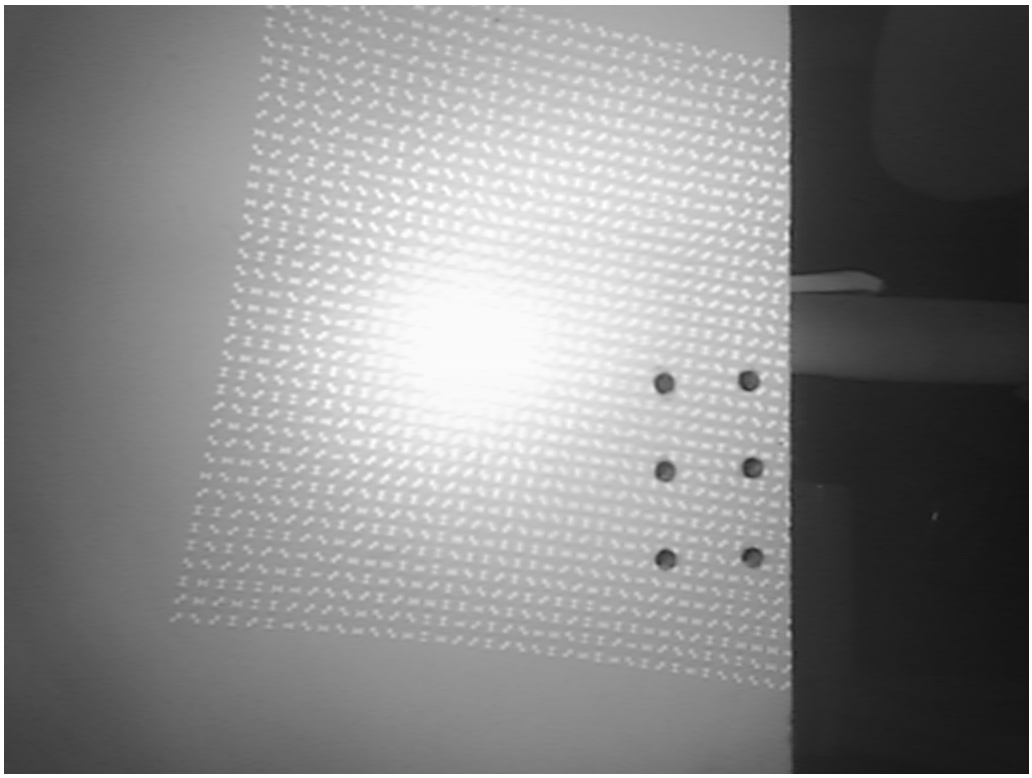


Figure 4.4.12 Input Image #12, Output image comparison



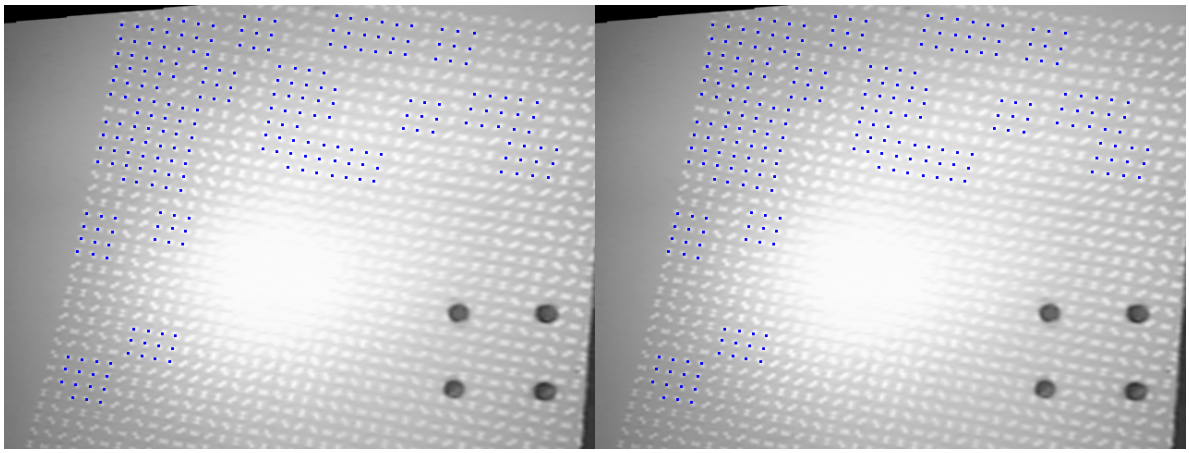
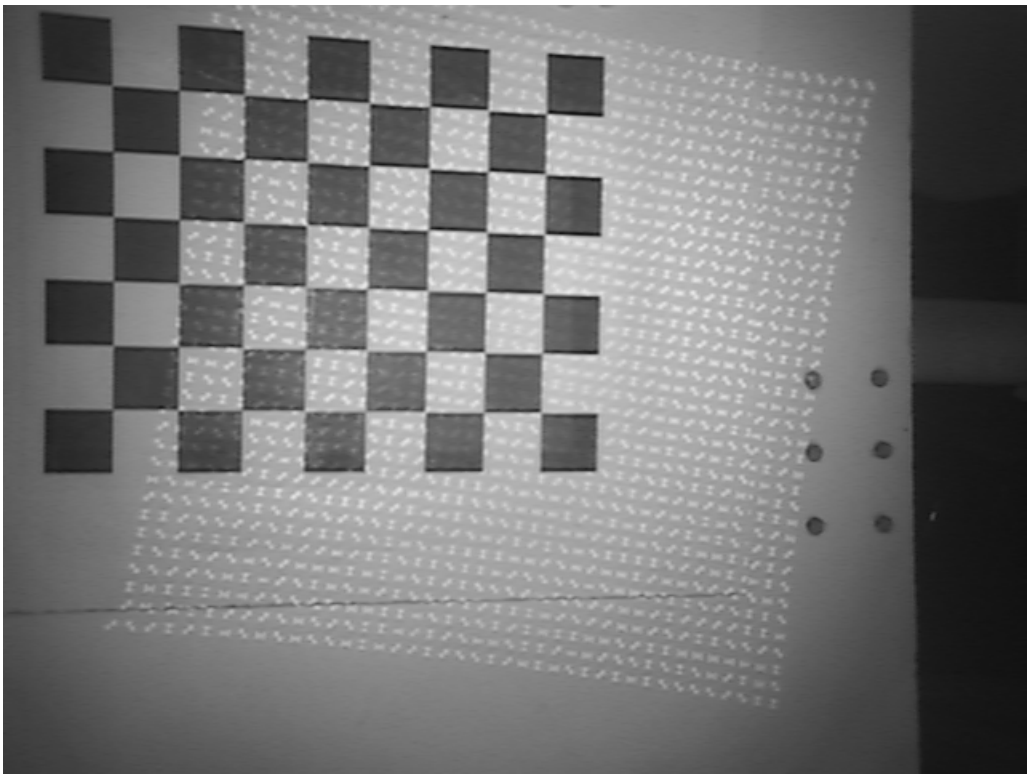


Figure 4.4.13 Input Image #13, Output image comparison



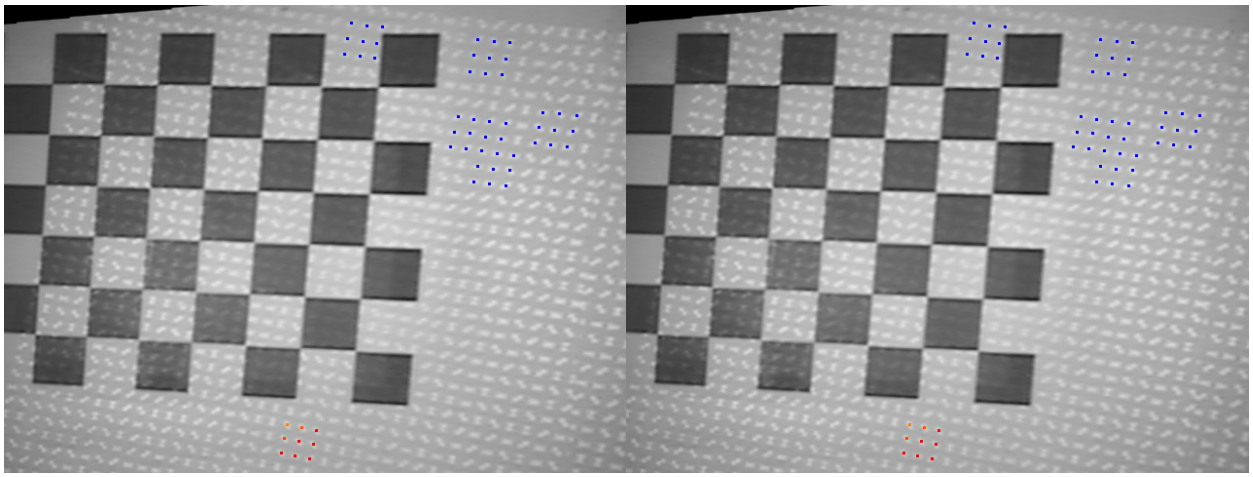
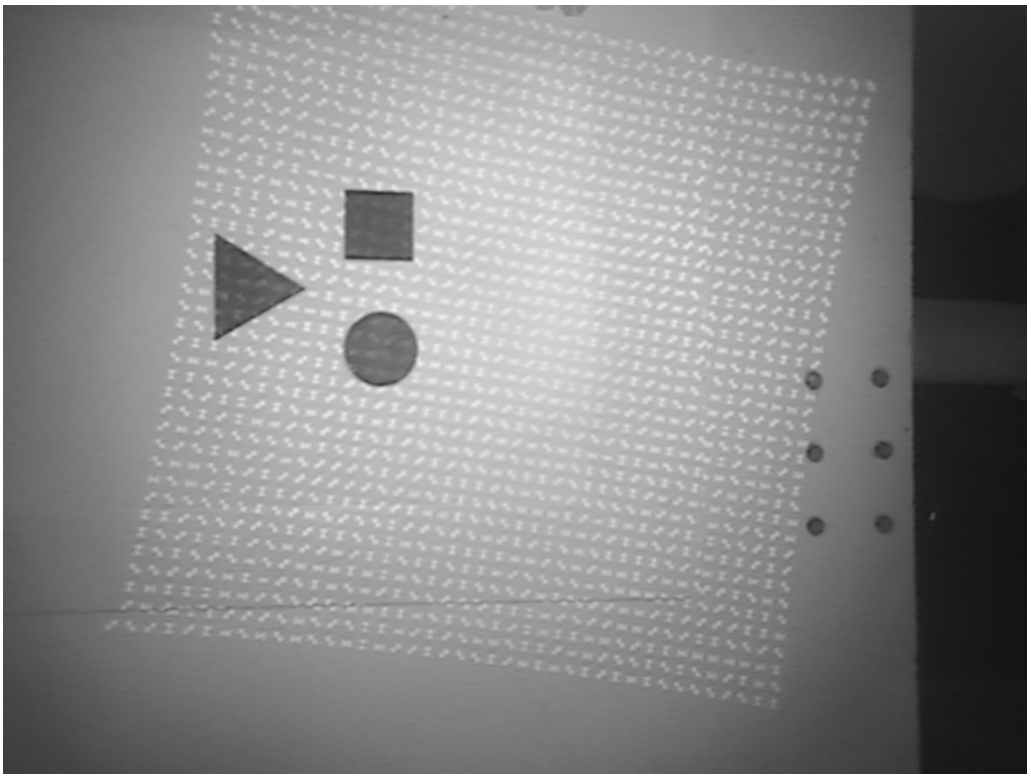


Figure 4.4.14 Input Image #14, Output image comparison



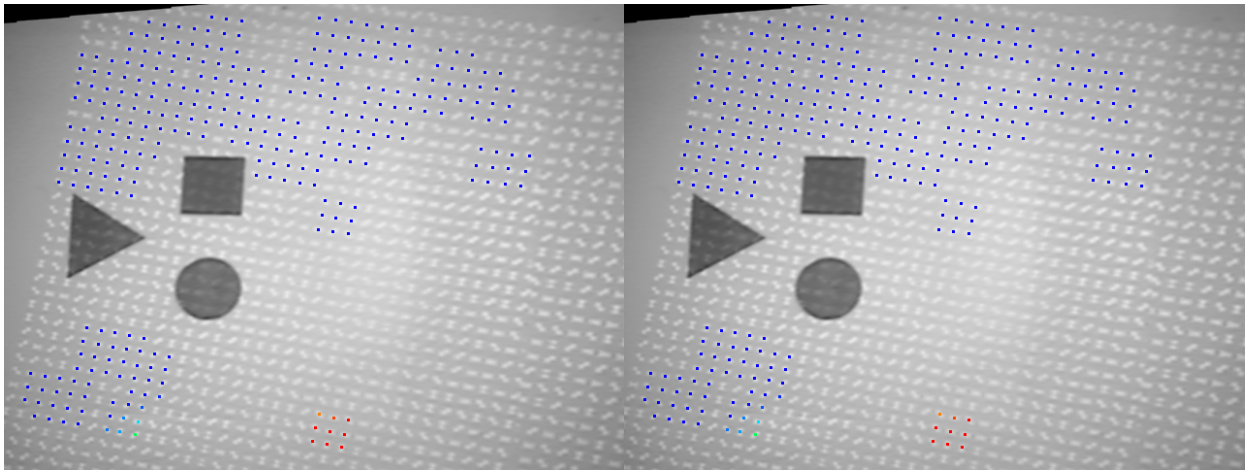
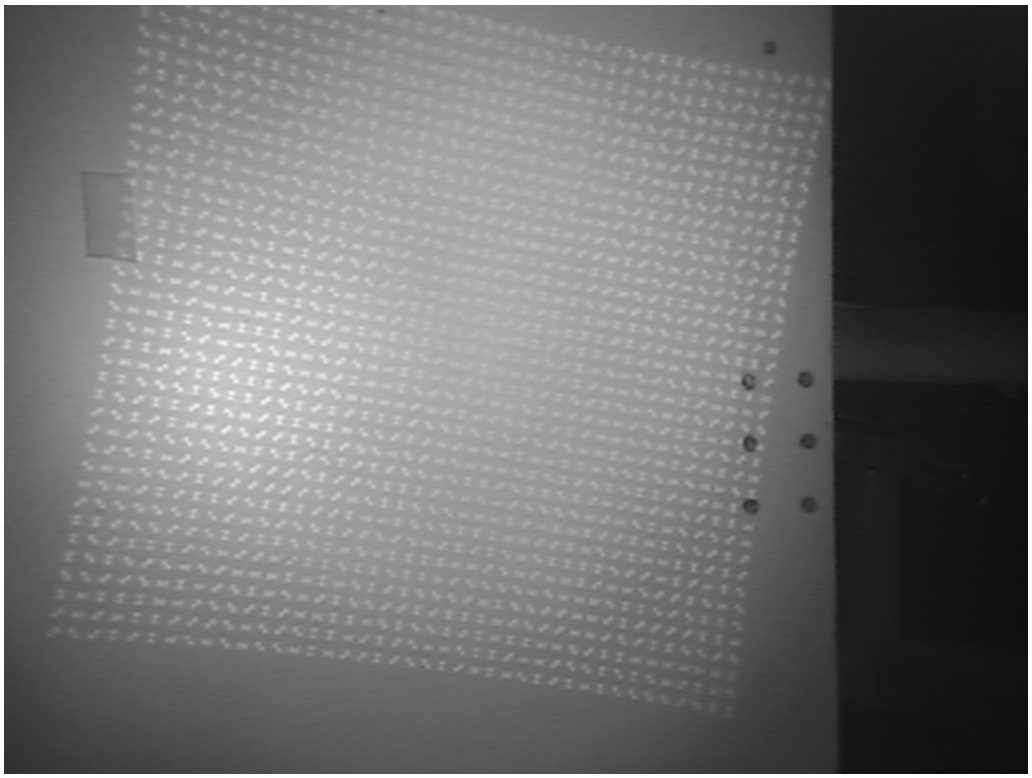


Figure 4.4.15 Input Image #15, Output image comparison



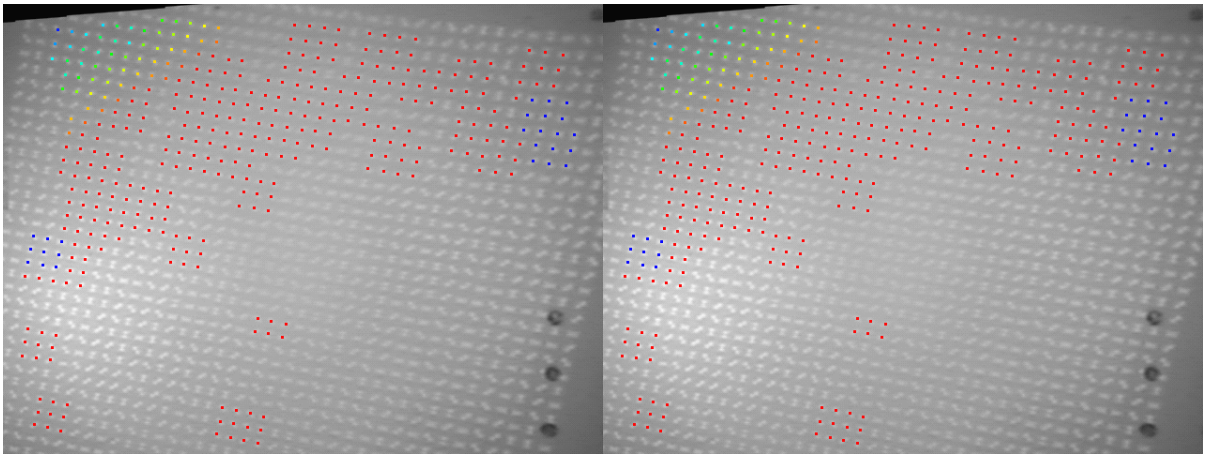
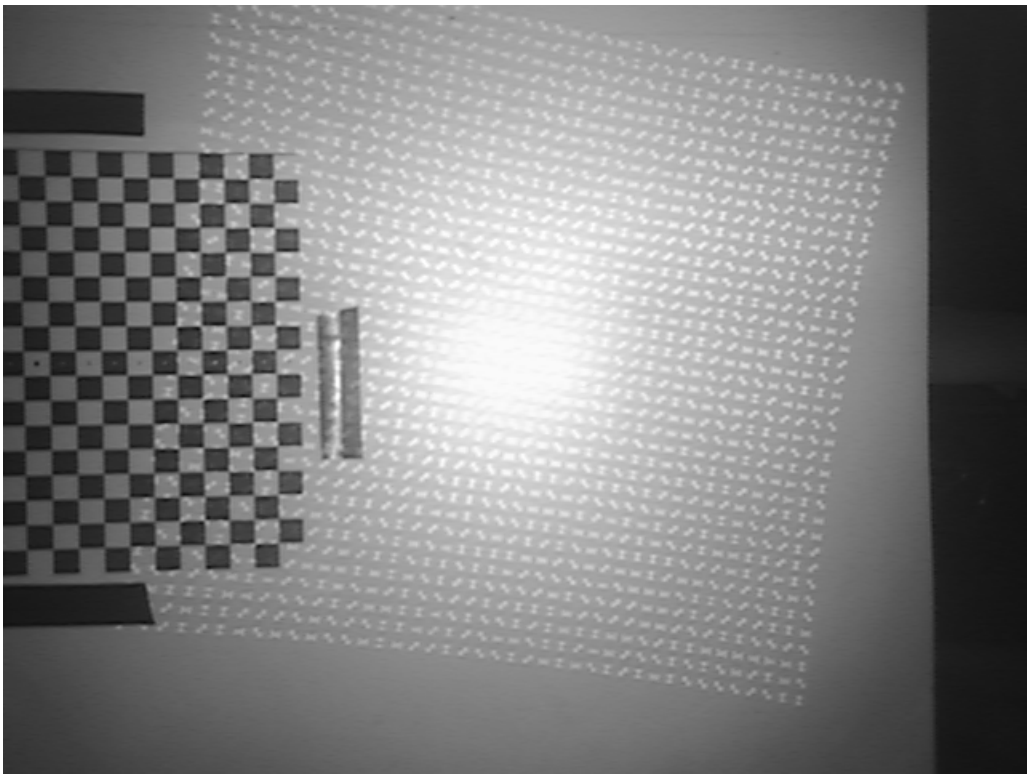


Figure 4.4.16 Input Image #16, Output image comparison



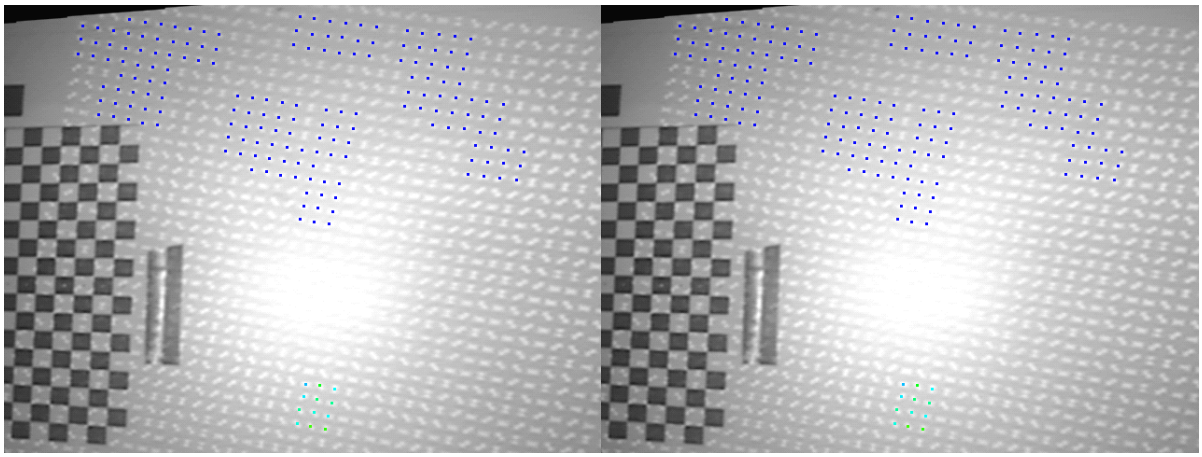
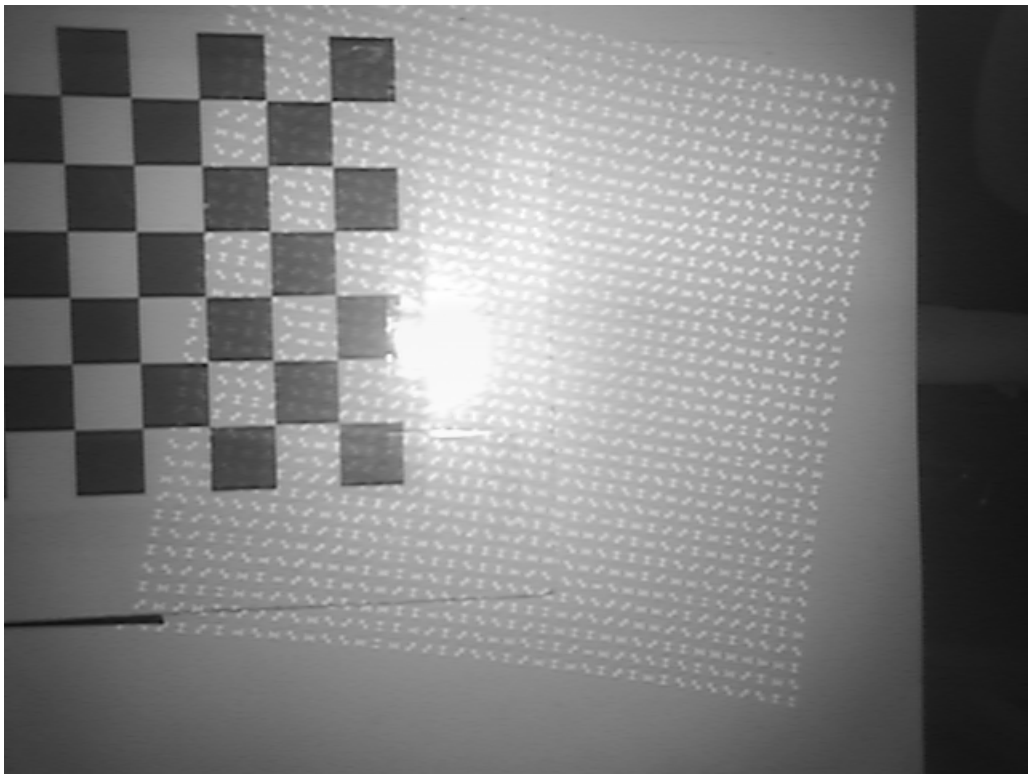


Figure 4.4.17 Input Image #17, Output image comparison



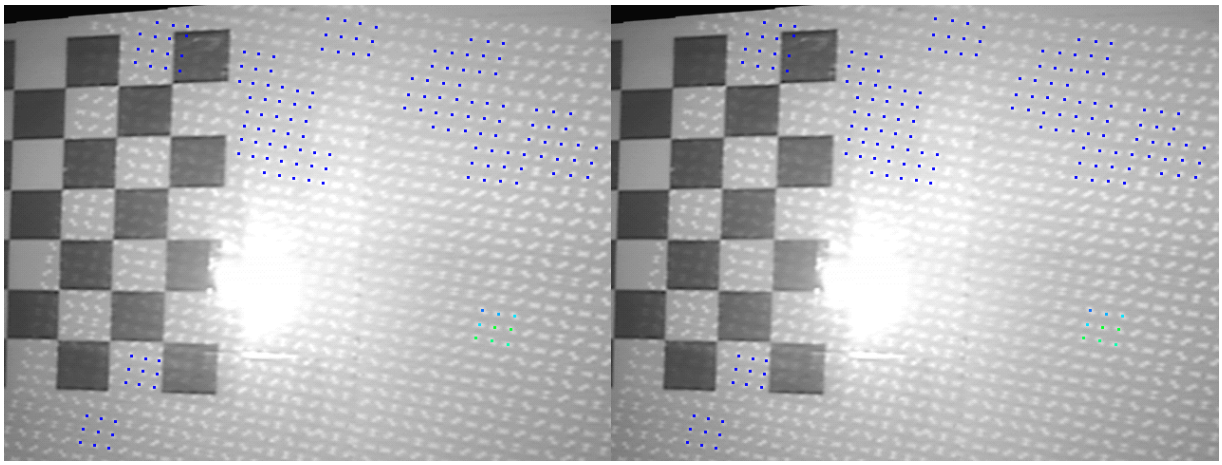
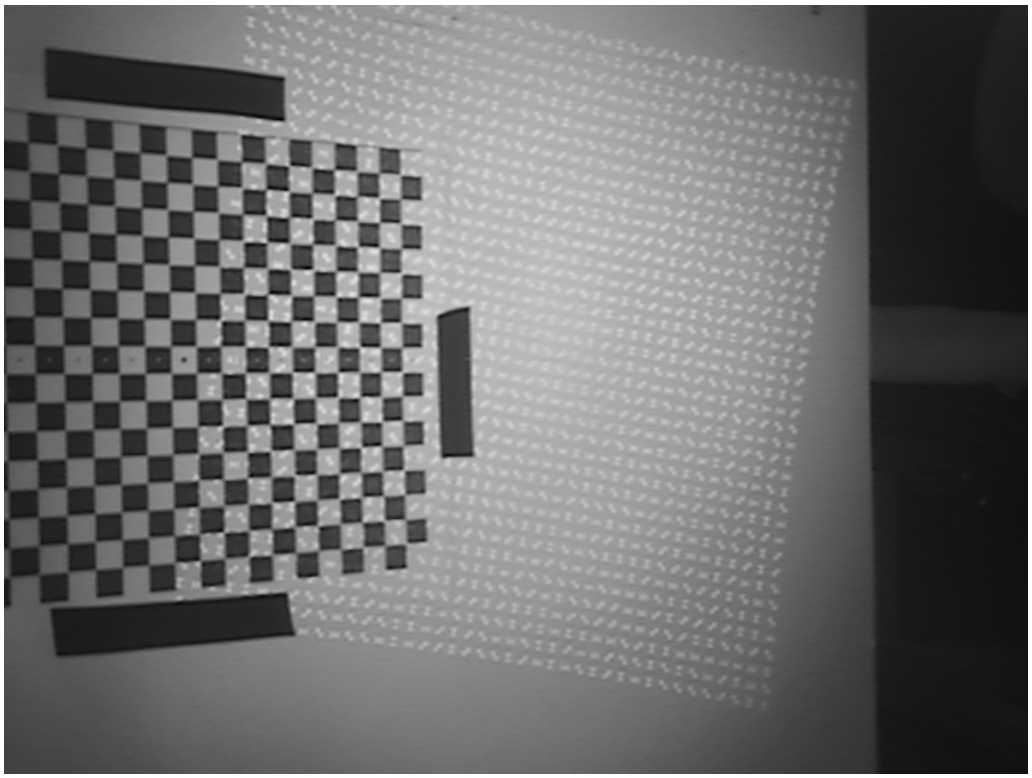


Figure 4.4.17 Input Image #17, Output image comparison



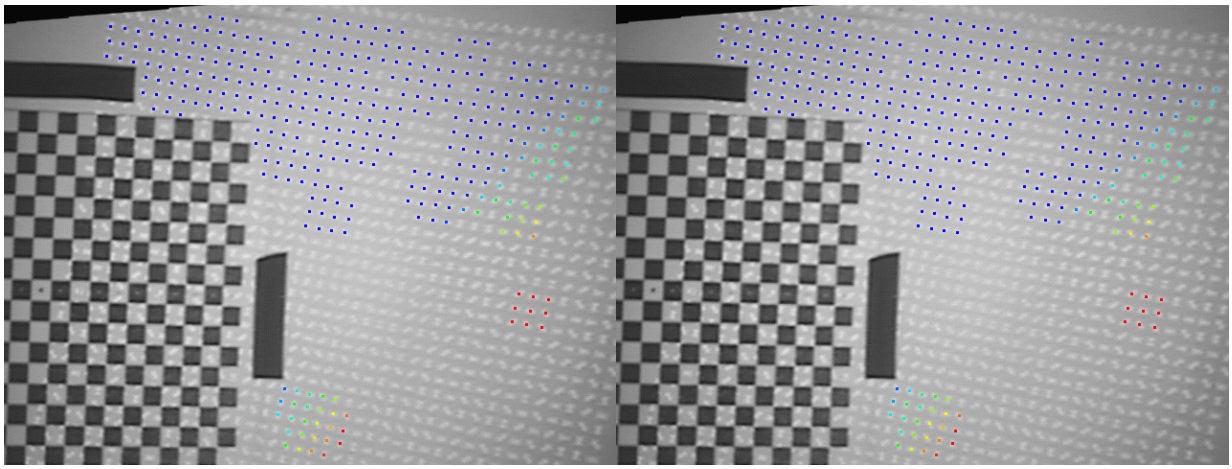
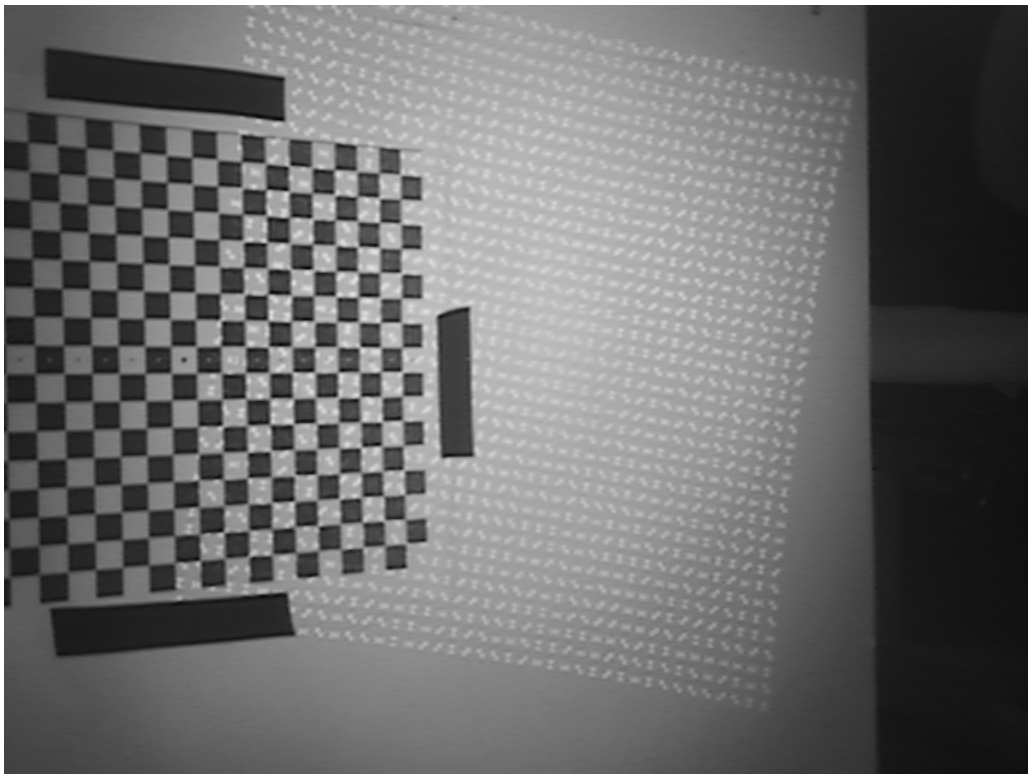


Figure 4.4.18 Input Image #18, Output image comparison



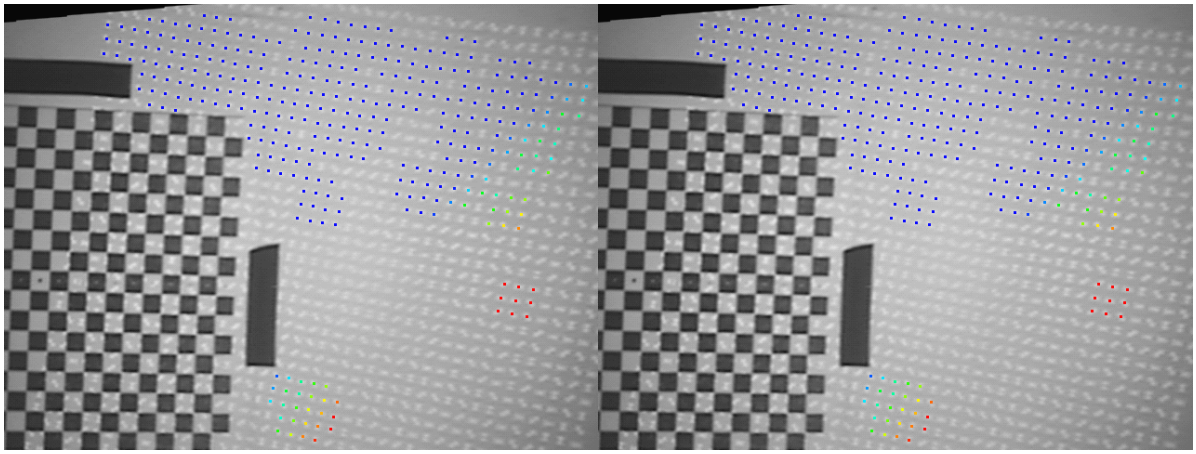
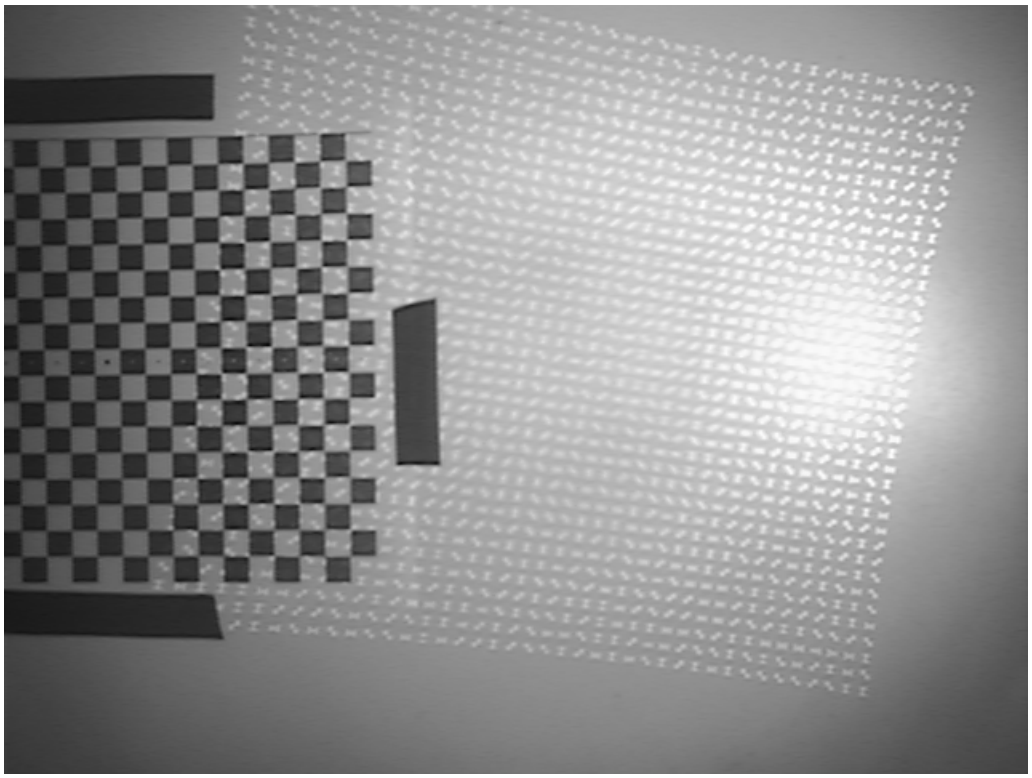


Figure 4.4.19 Input Image #19, Output image comparison



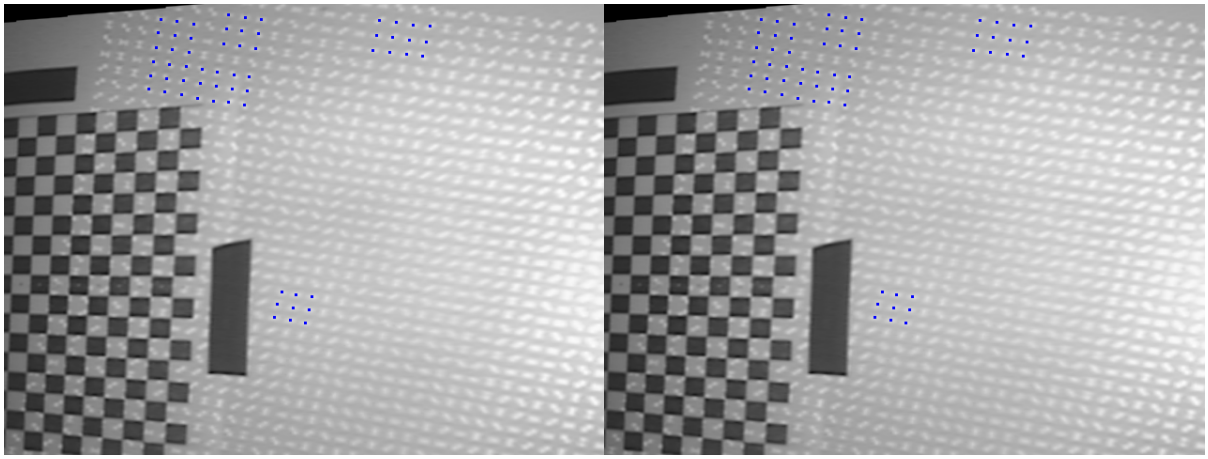
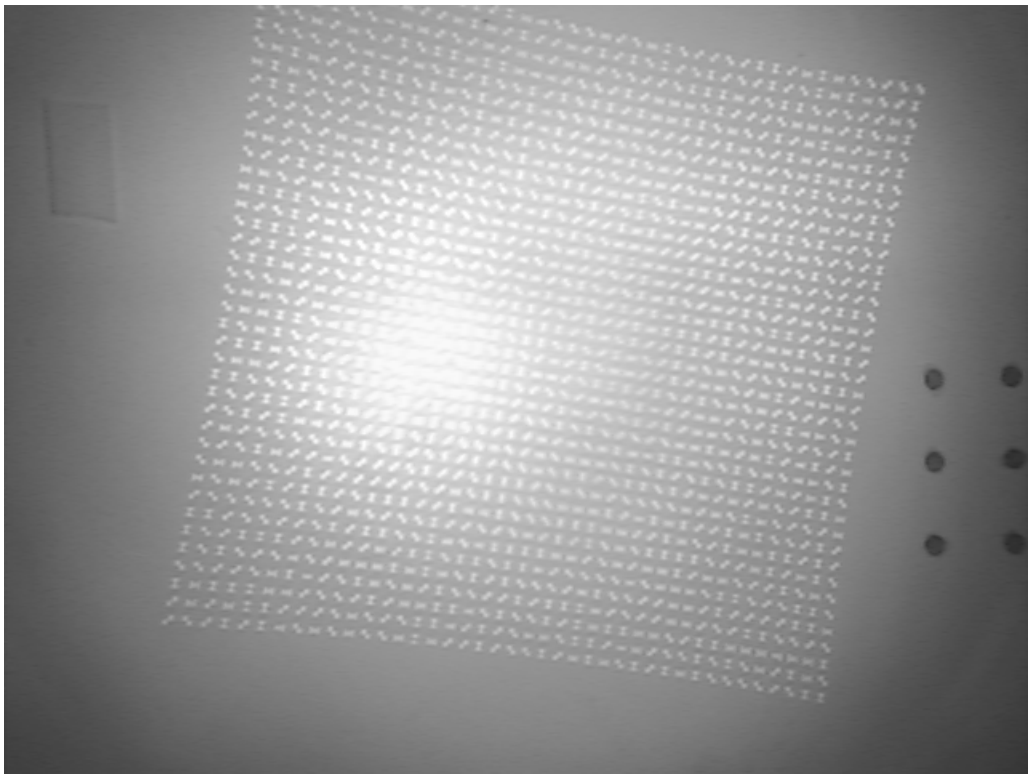


Figure 4.4.20 Input Image #20, Output image comparison



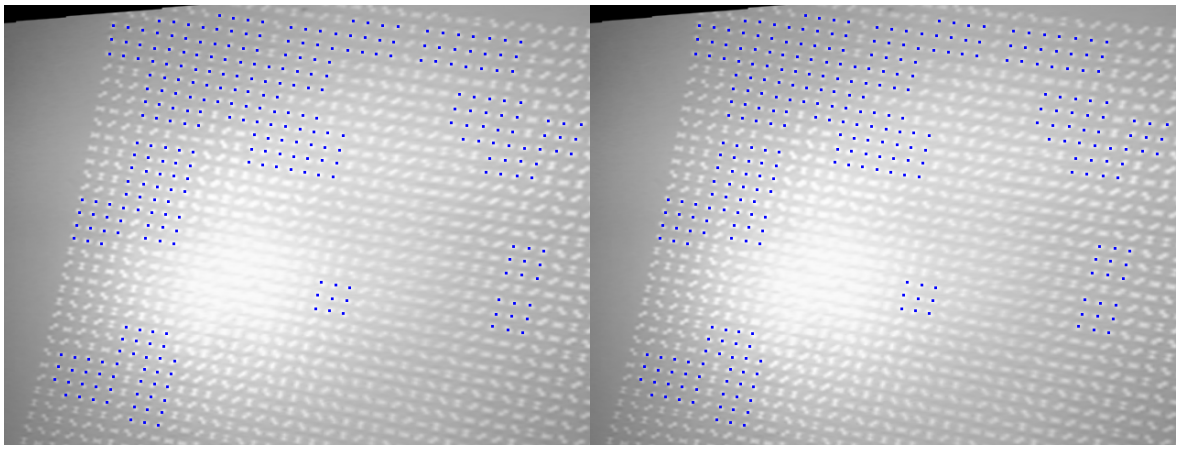


Figure 4.4.21 Input Image #21, Output image comparison

Chapter 5

Conclusion and Future Work

In this paper, I presented the parallel decoding algorithm using CUDA based GPU of three dimensional data streams in real-time. The calculation steps are involved in this algorithm are distance computation, k-NN sorting and angle position sorting. Point clouds in these kernels are implemented under CUDA-based GPU parallel computation architecture.

Many experiments are being used to test the stability of this program. The parallel decoding algorithm reduces the computation complexity from six to two and accelerates the serial decoding algorithm with the speedup times up to 2149X. Summarizing the experiments results and analyze them, I have successfully implemented a efficient and stable parallel decoding algorithm with CUDA-based GPU for the single-directional Camera system. Furthermore, the k-NN parallel sorting algorithm can also be widely applied to any other application which needs to use the distance as a reference to order the large scale of data. Also, as shown in Figure 4.1 and 4.2, it is easy to see that the implementation speed also determines by how to allocate the number of the blocks and threads. As illustrated in table 4.1 and 4.2, implementing decoding with 128 threads/block is faster than implementing decoding using 512 threads/block. This is because, in GPU, each thread performs parallel computing from the others, so does blocks, and between blocks there is no communication involved. Therefore increasing the number of blocks

and decrease the number of threads will decrease the number of data points needed to compute in each block and increase the number of data parallelism for each frame which will increase parallel computing power. And for large scaled data points, using more threads for each block can result less processing time than putting all data in one block.

In order to accelerating the single-direction camera system by GPU, design an algorithm which efficiently utilizes the CUDA-based GPU parallel program structure is very important. These includes pre-define the data structure, design the shape of the grid and blocks, modification of existing sequential arithmetic, and optimized allocate memory on host and device. As seen in Figure 4.3, I can conclude that using different number of blocks and different number of threads per block makes a big difference on processing time.

The final goal is to achieve video rate for the single-directional 3-D camera system to fast and accurate navigate the robot arm to correctly and quickly grab the object in front of the robot. The parallel decoding algorithm optimizes the entire program to a significant speed improvements, 2000×. However, this is not enough to improve the whole system to have video rate response. Future implementation of other eight functions in the software architecture is necessary. Therefore, future work will be focused on parallel implementing the other eight functions in the system using CUDA based GPU.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] David B.Kirk, Wen-mei W.Hwu (2010). *Programming Massively Parallel Processors*. London: Morgan Kaufmann. 1-279.

[2] Nvidia, “Compute Unified Device Architecture Programming Guide Version 4.0," http://developer.download.nvidia.com/compute/cuda/4.0/docs/NVIDIACUDAProgrammingGuide 4.0.pdf

[3]Nvidia, “NVIDIA GeForce GTX 470 GPU Architectural Overview,” http://www.nvidia.com/docs/IO/55506/GeForce GTX 470 GPU Technical Brief.pdf, May 2009

[4] ——, “The CUDA Compiler Driver NVCC,” http://www.nvidia.com/object/io 1213955090354.html.

[5] W. J. van der Laan, “Decuda,” http://wiki.github.com/laanwj/decuda/.

[6] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “NVIDIA Tesla: A Unified Graphics and Computing Architecture,” IEEE Micro, vol. 28, no. 2, pp. 39–55, 2008.

[7] V. Volkov and J. W. Demmel, “Benchmarking GPUs to Tune Dense Linear Algebra,” in SC ’08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[8] Z. S. Hakura and A. Gupta, “The Design and Analysis of a Cache Architecture for Texture Mapping,” SIGARCH Comput. Archit. News, vol. 25, no. 2, pp. 108–120, 1997. [12] H. Goto, “Gt200 over view,” http://pc.watch.impress.co.jp/docs/2008/0617/kaigai 10.pdf, 2008.

[9] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, “Analyzing CUDA Workloads Using a Detailed GPU Simulator,” in Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, April 2009, pp. 163–174.

[10] D. Kirk and W. W. Hwu, “ECE 489AL Lectures 8-9: The CUDA Hardware Model,” http://courses.ece.illinois.edu/ece498/al/Archive/Spring2007/lectures/lecture8-9-hardware.ppt, 2007.

[11]H. J′egou, M. Douze, and C. Schmid, “Hamming embedding and weak geometric

consistency for large scale image search," in European Conference on Computer Vision, Marseille, France, 2008.

[12] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk,and W. W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA," in PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA: ACM, 2008, pp. 73–82..

[13] J, Xu, N. Xi, C. Zhang and Q. Shi,"Real-time 3-D shape measurement system based on single structure light pattern," IEEE International Conference in Robotics and Automation (ICRA), 2010

[14] MaCallum A, Nigam K. A comparison of event models for Navie Bayes text classification. In: Processings of AAAI-98 workshop on learning for text categorization, 41-48, 1998.

[15] Joachims T. Learning to classify text using support vector machines: methods, theory and algorithms. Springer, Berlin, 2002.

[16] D. Aha, D. Kibler, and M. K. Albert. Instance-based learning algorithms. Machine Learning, 6:37–66, 1991.

[17] Han, E.H., George, K., Vipin, K.: Text categorization using weight adjusted k-nearest neighbor classification: Technical Report. University of Minnesota (2000).

[18] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. iDistance: An adaptive B+-tree based indexing method for nearest neighbor search. ACM Transactions on Database Systems, 30(2):364-397, June 2005.

[19] Xiang Li, Daming Shi, Varoon Charastrakul and Junhong Zhou. Advanced P-Tree based K-Nearest neighbors for customer preference reasoning analysis. Journal of Intelligent Manufacturing, Volume 20, Number 5, Pages 569-579,2009.

[20] An Gong, Yanan Liu. Improved KNN Classification Algorithm by Dynamic Obtaining K. Communications in Computer and Information Science, Volume 143, 320-324, 2011.

[21] M. Kamber, J. Han, Data Mining: Concepts and Techniques, 2nd Edition, Morgan Kaufmann, 2005.

[22] B. Catanzaro, N. Sundaram, K. Keutzer, "Fast Support Vector Machine Training and Classification on Graphics Processors", Proc. ofInternational Conference on Machine Learning , 2008, pp. 104-111.

[23] G. Vasiliadis, S. Antonatos , M. Polychronakis, et ai, "Gnort : High Performance Network Intrusion Detection Using Graphics Processors", Recent Advances in Intrusion Detection (RAID), 2008, Vol. 5230, pp. 116-134.

[24] V. Garcia, ′ E. Debreuve, and M. Barlaud, “Fast k nearest neighbor search using gpu,” in CVPR Workshop on Computer Vision on GPU, Anchorage (AK), USA, 2008.

[25] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” Int. J. Comput. Vision, vol. 60, pp. 91–110, 2004.

[26] V. Garcia, E. Debreuve, M. Barlaud, "Fast K Nearest Neighbor Search using GPU", IEEE Conference on Computer Vision and Patter Recognition Workshops, 2008, Vol. 1-3, pp. 1107-1112.

[27] H. Zhang, A. C. Berg, M. Maire, and J. Malik, “SVM-KNN: Discriminative nearest neighbor classification for visual category recognition,” in International Conference on Computer Vision and Pattern Recognition, New York (NY), USA, 2006.

[28] M. N. Goria, N. N. Leonenko, V. V. Mergel, and P. L. Novi Inverardi, “A new class of random vector entropy estimators and its applications in testing statistical hypotheses,” J. Nonparametr. Stat., vol. 17, pp. 277–297, 2005.

[29] F. Pan, B. Wang, X. Hu, and W. Perrizo, “Comprehensive vertical sample-based knn/lsvm classification for gene expression analysis,” J. Biomed. Inform., vol. 37, pp. 240–248, 2004.

[30] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, “An optimal algorithm for approximate nearest neighbor searching fixed dimensions,” Journal of the ACM, vol. 45, pp. 891–923, 1998.

[31] H. J′egou, M. Douze, and C. Schmid, “Searching with quantization: approximate nearest neighbor search using short codes and distance estimators,” Tech. Rep. RR-7020, INRIA, 2009.

[32] D. Qiu, S. May, and A. N¨uchter, “GPU-accelerated nearest neighbor search for 3-D registration,” in International Conference on Computer Vision Systems, Li`ege, Belgium, 2009.

[33] Y. Zhuge, Y. Cao, and R.W. Miller, “GPU accelerated fuzzy connected image segmentation by using CUDA,” in Engineering in Medicine and Biology Conference, Minneapolis (MN), USA, 2009.