Tampere University

Kianoush Jafari

MULTI-VIEW DEPTH ESTIMATION AND PLANE-SWEEPING REDNERING ON GRAPHIC PROCESSOR UNIT

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Atanas Gotchev
Pekka Jääskeläinen
August 2022

ABSTRACT

Kianoush Jafari: MULTI-VIEW DEPTH ESTIMATION AND PLANE-SWEEPING RENDERING ON GRAPHICAL PROCESSOR UNIT
Master of Science Thesis
Tampere University
Master's Degree Program in Information Technology
August 2022

Depth estimation and image-based rendering are two of the essential tasks in computer vision and key enablers of many modern-day technologies such as autonomous navigation, robot-assisted surgery, and 2-D to 3-D image conversion in the movie industry, to name but a few. Depth estimation can be defined as the problem of estimating the distance of each pixel within an image to the camera. Closely related to that, image-based rendering (IBR) is concerned with generating novel views from existing images. While the former is often considered one of the fundamental tasks in computer vision with a wide range of applications, the latter is mainly associated with virtual reality, immersive technologies, and 3-D reconstruction of the scene.

Some of these applications' significant drawbacks are their large data throughput volume, large memory bandwidth requirement, and high processing time. Considering this, they often require hardware acceleration to work in real-time. In this thesis work, we investigate the parallel computing power of graphical processor unit (GPU) and also multi-core CPU by implementing a multi-view stereo algorithm for sparse light field depth estimation and an IBR algorithm based on plane-sweeping rendering on these platforms. We use Open Computing Language (OpenCL) as our programming framework of choice for GPU computing and OpenMP API for multi-core CPU implementation. We have shown that our GPU implementation can achieve up to hundreds of times of speed-up once it is compared against both single-core and multi-core CPU implementations.

Keywords: Graphic Processor Unit, Multi-view Stereo, Image-based Rendering, OpenCL, Plane-sweeping

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

CONTENTS

1. INTRODUCTION 1
2. BACKGROUND 4
2.1 Stereo Vision for Depth Estimation 4
2.1.1 Pinhole Camera Model 4
Camera Intrinsic Parameters 5
Camera Extrinsic Parameters 5
2.1.2 Stereo Matching 6
2.1.3 Multi-view Stereo 12
2.2 Image-based Rendering 13
2.3 Parallel Computing 15
2.3.1 Heterogeneous Computing 16
2.3.2 Heterogeneous Computing for parallel processors 18
2.3.3 GPU Architecture 18
2.3.4 Open Computing Language 22
3. METHODS 30
3.1 Multi-view stereo on Sparse Light Field Data 30
3.1.1 Simple Linear Iterative Clustering 32
3.1.2 Depth Initialization 34
3.1.3 Depth Refinement 35
3.1.4 Implementation methodology 38
3.2 View-interpolation Rendering 40
3.2.1 Plane-sweep Rendering 40
3.2.2 GPU Implementation 42
4. EXPERIMENTAL RESULTS AND ANALYSIS 45
4.1 Experimental Setting 45
4.2 Multi-view Depth Estimation 45
4.2.1 Dataset Specifications 45
4.2.2 Superpixel Segmentation Results 45
4.2.3 Initialization and Refinement Results 47
4.2.4 Quality Analysis 49
4.2.5 Performance Evaluation 50
4.3 Plane-sweep Rendering 51
4.3.1 Stereo Dataset Specification 51
4.3.2 Execution Parameters 51
4.3.3 Plane-sweep Results 52
4.3.4 Rendering Results 52
4.3.5 Execution Time 53
5. CONCLUSIONS 55
REFERENCES 56

LIST OF FIGURES

FIGURE 2.1 ONLY ONE RAY OF LIGHT FROM EACH 3D POINT CAN HIT THE SENSOR PLANE 4
FIGURE 2.2. TYPICAL STRUCTURE OF PINHOLE CAMERA MODEL 5
FIGURE 2.3. IMAGE PLANE OF PINHOLE CAMERA MODEL... 5
FIGURE 2.4. ROTATION AND TRANSLATION CHANGES COORDINATE SYSTEM 6
FIGURE 2.5 A TYPICAL STEREO MATCHING PIPELINE 8
FIGURE 2.6. THE SIMILARITY FUNCTION BASED ON SIAMESE NETWORK 9
FIGURE 2.7 GENERAL SETUP OF EPIPOLAR GEOMETRY 9
FIGURE 2.8 EPIPOLAR SETUP FOR STEREO MATCHING SYSTEM 10
FIGURE 2.9. DISPARITY MAP WITHOUT AGGREGATION (LEFT) GROUND TRUTH (RIGHT)......................... 11
FIGURE 2.10. VISIBLE AND OCCLUDED CAMERA IN MVS CAMERA SETUP 13
FIGURE 2.11. THE NOMENCLATURE OF IBR TECHNIQUES [52]. 15
FIGURE 2.12 DISTRIBUTION OF SEQUENTIAL AND PARALLEL PORTION OF AN APPLICATION [3]. PEACH IS THE TOTAL EXECUTION TIME OF APPLICATION. MEAT OF THE PEACH (ORANGE) IS THE DATA-PARALLEL PART AND PIT OF THE PEACH (RED) IS THE SERIAL PART OF THE CORE. 19
FIGURE 2.13. THE DIFFERENCE BETWEEN CPU AND GPU ARCHITECTURE [3]. 20
FIGURE 2.14. COMPUTING ORGANIZATION OF AN SM [20]. 22
FIGURE 2.15. OPENCL ABSTRACT HARDWARE MODEL [7]... 23
FIGURE 2.16. VECTOR ADDITION DIAGRAM.. 25
FIGURE 2.17. THE GENERAL DIAGRAM OF OPENCL MEMORY REGIONS [7]. 28
FIGURE 3.1. CAMERA ARRAY SYSTEM (LEFT) AND PLENOPTIC CAMERA (RIGHT). 30
FIGURE 3.2. AN EXAMPLE OF SUPER-PIXEL PIXEL SEGMENTATION [82].. 31
FIGURE 3.3. THREE STAGES OF ALGORITHMS FROM LEFT TO RIGHT: SUPERPIXEL SEGMENTATION, DEPTH INITIALIZATION, DEPTH REFINEMENT. 32
FIGURE 3.4. THE SLIC PIPELINE ON GPU [82] 33
FIGURE 3.5. DIFFERENT PROJECTIONS OF A SINGLE PIXEL FOR DIFFERENT DEPTH HYPOTHESIS. 34
FIGURE 3.6. PROPAGATION KERNEL (LEFT) AND REFINEMENT PROCEDURE (RIGHT) 36
FIGURE 3.7. EMBARRASSINGLY PARALLEL PATTERNS: MAP (LEFT) AND STENCIL (RIGHT). 39
FIGURE 3.8. REDUCTION PATTERN.. 40
FIGURE 3.9. A GENERAL VIEW (CAMERA) SETUP OF THE ALGORITHM... 41
FIGURE 3.10. HIGH DATA ACCESS OVERLAPPING BETWEEN RED, GREEN, AND YELLOW WORK-ITEMS IN THE NAÏVE GPU KERNEL. ALL THREE WORK-ITEMS BELONG TO THE SAME GROUP (BLUE BOX) 43
FIGURE 3.11. MEMORY ACCESS PATTERN OF THE ALGORITHM 2. 44
FIGURE 4.1. THE DEFAULT SLIC OUTPUT WITH SUPERPIXEL SIZE 8 46
FIGURE 4.2 EFFECT OF THE ENFORCE CONNECTIVITY.. 47
FIGURE 4.3 SEGMENTATION WITH SUPER-PIXEL SIZE 16... 47
FIGURE 4.4 INITIAL DEPTH ESTIMATION WITH SUPER-PIXEL SIZE 8.. 48
FIGURE 4.5. OUR REFINED DISPARITY MAP. THE SUPER-PIXEL SIZE IS 8... 48
FIGURE 4.6. REFERENCE PAPER OUTPUT USING SUPERPIXEL SIZE 8 49
FIGURE 4.7. THE IMPROVED RESULT ON BAR DATASET USING SUPERPIXEL SIZE 16................................ 49
FIGURE 4.8. ALGORITHM'S RESULT ON BIERGARTEN DATASET USING SUPERPIXEL 8 50
FIGURE 4.9. PLANE-SWEEPING DEPTH MAP FOR CHAIR DATASET (LEFT) AND PIANO DATASET (RIGHT). 52
FIGURE 4.10. RENDERING VIEW FOR LIVING ROOM DATASET WITH DELTA EQUALS TO 0.55 53
FIGURE 4.11. RENDERED VIEW FOR PIANO DATASET WITH DELTA EQUAL TO 0.55................................. 53

LIST OF SYMBOLS AND ABBREVIATIONS

ALU	Arithmetic Logic Unit
CU	Control Unit
CUDA	Compute Unified Device Architecture
DLP	Data Level Parallelism
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
FIFO	First In First Out
GPU	Graphic Processor Unit
HDL	Hardware Description Language
HSL	High Level Synthesis
MVS	Multi-view Stereo
OpenCL	Open Computing Language
PC	Processor Cluster
SLIC	Simple Linear Iterative Clustering
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SPMD	Single Program Multiple Data
SVM	Shared Virtual Memory
SID	Squared Intensity Difference
SM	Streaming Multiprocessor
SP	Streaming Processor
SSD	Sum Square Difference
TSSD	Truncated Squared Sum Difference
ToF	Time of Flight

a	position between red and green colors in CIELAB color space
α	normalization coefficient for color distance
b	position between blue and yellow colors in CIELAB color space
C	superpixel color
d	disparity / depth
d_{lab}	Euclidian-based color distance in CIELAB
d_{xy}	Euclidian-based spatial distance
D_s	SLIC distance function
E	energy function
E_c	consistency term of energy function
E_s	smoothness term of energy function
K	intrinsic matrix
L	lightness term in CIELAB / left stereo image
Lab	CIELAB
n	normal vector
O	occlusion term
P	pixel
R	rotation matrix / right stereo image
S	depth similarity function
σ	spatial distance normalization coefficient
T	threshold / translation vector
u	pixel index on x-axis
v	pixel index on y-axis
V	visibility term
Ω	reference superpixel

ω	color similarity function
Z	third dimension

1. INTRODUCTION

Retrieving the 3D geometry of the scene from the raw sensory data is one of the most fundamental problems in computer vision. The methods which perform such tasks, depending on the type of sensors they use, can be categorized into two groups: active and passive. The active methods tend to measure the depth by physical interaction with the environment. Methods based on time of flight (ToF) and Ultrasound are examples of this type of approach [21][22]. Passive methods, on the other hand, are all computation-based methods that analyze the optical features of the scene by capturing images via cameras. Methods based on stereo correspondences are a prominent example of such methods that estimate the depth value for each pixel within an image (generating a depth map) by analyzing the scene from different views captured by a number of cameras. Consequently, these methods can be further categorized by the number of cameras they use, namely two-view stereo vision (uses two cameras only) and multi-view stereo (uses more than two cameras).

Stereo vision has a broad spectrum of applications ranging from robotics to 3-D and augmented reality. Most of these applications require real-time execution and highly accurate depth maps. However, satisfying these constraints is difficult due to the data-intensive nature of algorithms and high memory and bandwidth requirements. Moreover, the ever-increasing resolution of images (HD, 4k, 6k, and 8k), increasing number of cameras (light field cameras, for example), and increase in depth value range are other factors that have made stereo algorithms increasingly time-consuming.

Closely related to stereo vision and yet different are image-based rendering (IBR) algorithms, which aim to create virtual views from already existing ones. Their applications are in virtual reality, where they create images from angles of the scene that cameras do not cover. In practical cases, IBR algorithms need to be executed in real-time to create the immersive feeling that the users desire. Many IBR techniques require geometric information of the scene, which they use stereo vision techniques to achieve. As a result, they suffer from high computational cost for the same reasons as stereo algorithms, such as high resolution of the images and extensive range of depth hypothesis. In order to address the computational issues of both problems, we can use a wide

range of hardware accelerators to reduce the execution time. In this thesis work, we use Graphic Processor Unit (GPU), which is optimized for data-parallel processing, as the platform of choice.

GPUs are massively parallel arithmetic-oriented processors with high-throughput memory bandwidth, which allow for performing a large number of arithmetic operations in real-time. Originally designed exclusively for graphic computing, they later evolved to perform general purpose computing tasks, which have widely expanded the range of their applicability beyond graphic computing domain: something which is called general purpose computing with GPU or **GPGPU**. Today, GPUs are being used as hardware accelerators in many high-performance and parallel computing systems to gain high speed-up for different applications. Alongside FPGAs and multi-core CPUs, they are often considered one of the major pillars of high-performance computing domain [11]. Compute Unified Device Architecture (CUDA), and Open Computing Language (OpenCL) are two of the most prominent GPGPU programming frameworks, while OpenGL and Directx3D are exclusively used for graphics computing.

A GPU device can fetch a big chunk of data from memory all at once and process it through a large number of data-parallel computing units it possesses to produce the final result. Optimized for performing matrix-like operations, they tend to perform very efficiently in processing image data types, making them a suitable choice for many computer vision and image processing applications. In terms of programmability, GPUs are known to be far easier to program than FPGAs if the latter is programmed by hardware description languages (HDL). In fact, it is possible to reach a real-time or near real-time performance with a simple GPU implementation for a massively parallel data-intensive application such as those of computer vision and image processing.

The objective of this thesis is to implement two different computer vision applications on GPU and report the speed-up by comparing it with the single-thread CPU implementation. The first application is a multi-view stereo vision pipeline that aims to produce dense depth map estimation for each of its input images which are obtained by a light field camera. The pipeline starts with an image segmentation engine to shrink the size of the images and proceeds with a depth initialization and later an optimization scheme to estimate and refine the depth values, respectively. The second application is an image-based rendering task in which a view interpolation algorithm is used to create non-real images captured by a virtual camera that moves between two real stereo cameras. Both applications are extremely time-consuming when executed on a traditional single-core CPU mostly due to high image resolutions and large range of depth

quantization levels or depth hypothesis. We will show that a GPU implementation can significantly reduce the execution time of both applications.

The rest of this thesis is organized as follows: Section 2 explains the necessary background for stereo vision, image-based rendering, and understanding the principles of GPU computing. In Section 3, we explain algorithms and our implementations. The results are represented in Section 4, and finally, we conclude the thesis in Section 5.

2. BACKGROUND

2.1 Stereo Vision for Depth Estimation

2.1.1 Pinhole Camera Model

In its simplest form, the camera can be shown as a barrier with an ideally infinitely small aperture (abstract point) in the middle, which is placed between the 3D object and the image sensor. The reason for such a system is that each point within the 3D world emits several rays of light in different directions. If there is no barrier in between, each point on the 2D sensor plane will receive light rays from all the points within the 3D scene. However, if there is a barrier with an ideal aperture in the middle, then only one ray of light from each 3D point can hit the sensor plane, creating a one-to-one mapping from the 3D world to the 2D sensor plane. The process has been demonstrated in Figure 2.1.

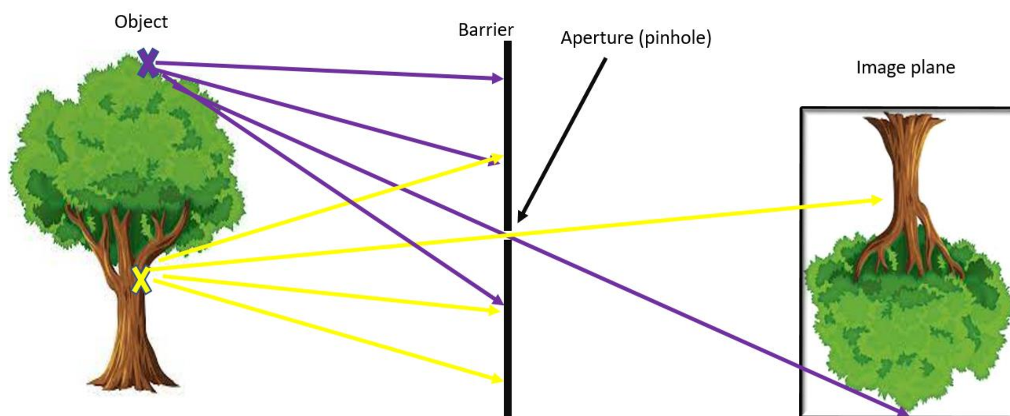

Figure 2.1 *Only one ray of light from each 3D point can hit the sensor plane [86].*

This simple system is referred to as **pinhole camera model**. Figure 2.2 shows a more detailed structure of an ideal pinhole camera model. In a pinhole camera, the distance between the pinhole and sensor plane is called the focal length. The line that passes through the pinhole and is perpendicular to the sensor plane is called optical axis and the intersection of this line with the sensor plane is called the principle point of the camera, which has the coordinate of $(0, 0)$.

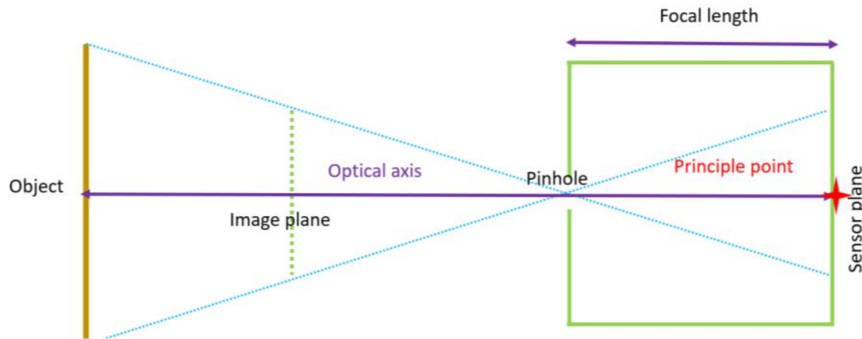

Figure 2.2. *Typical structure of pinhole camera model [87].*

Camera Intrinsic Parameters

The camera model is a mathematical description of how a camera captures images. Considering this, camera intrinsic are types of parameters that determine how the 3D points in the world are projected to 2D points on the image plane. Given the depicted scheme in Figure 2.3 and by using the triangles technique, the mapping formula for converting the 3-D point $P(X, Y, Z)$ to the 2-D point $p(x, y)$ can be done as follows:

Equation 1: $\boldsymbol{x = f\frac{X}{Z},\ \ y = f\frac{Y}{Z}}$

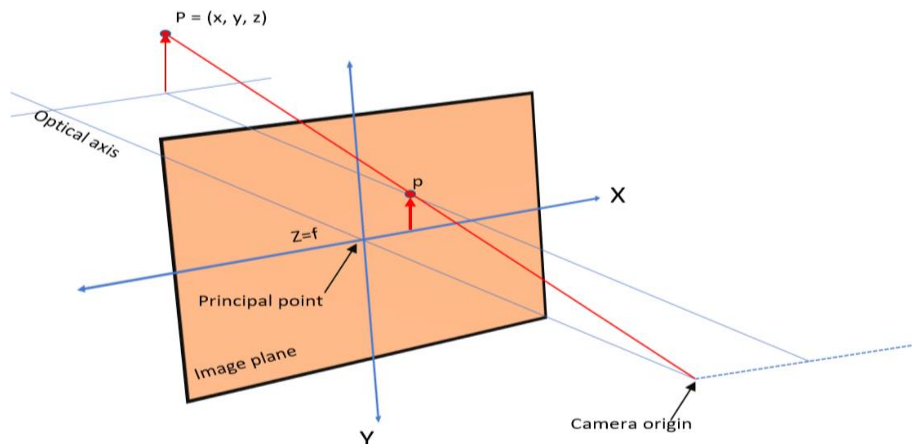

Figure 2.3. *Image plane of pinhole camera model [88].*

Camera Extrinsic Parameters

Camera intrinsics explain how the 3D coordinates are projected on the image plane, assuming that the camera and word coordinate systems are the same. The extrinsic parameters come into play when 3D points in the world are in the different coordinate system than that of the camera. A transformation is needed, as shown in Figure 2.4, to transfer these points from the world coordinate system to the camera coordinate system before projecting them on the image plane. These transformations come in the

form of rotation matrix $R_{3\times3}$ and translation vector $T_{3\times1}$, both of which can be concatenated to form what is known as extrinsic matrix Equation 2.

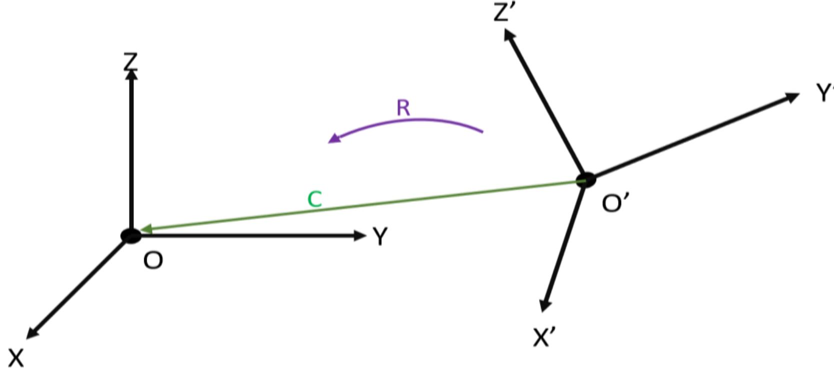

Figure 2.4. *Rotation and translation changes coordinate system*

Equation 2: $$[\boldsymbol{R}_{3\times3}|\boldsymbol{T}_{3\times1}] = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}, \text{where } \boldsymbol{T}_{3\times1} = -\boldsymbol{R}_{3\times3} \times \boldsymbol{C}_{3\times1}$$

The Equation 3 can be further updated with camera intrinsic and extrinsic matrices to form a new matrix $P_{3\times4}$ which is called projection matrix.

Equation 3: $$Z\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$= K[R_{3\times3}|T_{3\times1}]\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$= P_{3\times4}\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

2.1.2 Stereo Matching

Stereo matching is the most traditional form of depth estimating with cameras. In a stereo camera system, two identical cameras are put in slightly, horizontally or vertically, different positions from each other. The photos of the scene must be taken synchronously by the two cameras to make sure that the configuration of the scene does not change during the image acquisition process. The images which are obtained in this way are referred to as stereo pair images or just stereo images.

Taking the images from the same scene from two different perspectives allows the stereo matching algorithms to extract the depth information by analyzing the similarities

between the two views. The key idea enabling these algorithms comes from a physical phenomenon called the parallax effect, based on which distant objects tend to move slower than closer ones [18]. This principle was originally used in astronomy to estimate the distance of galactic objects such as stars and planets, but here it is being used to estimate the distance of each pixel from the camera [23].

According to the parallax, pixels belonging to the objects which are further away from the camera have more displacement (disparity) within the stereo images than the pixels of the closer objects. This implies that there is a reverse relationship between the depth of a pixel and its disparity in the stereo pair. However, finding the exact disparity for every single pixel has been shown to be an NP-hard problem for many stereo matching solutions [24]. Hence, many stereo matching algorithms are approximation solutions that aim at estimating the disparity of each pixel by finding the similar parts of two images. In this regard, it is often called the stereo correspondence problem. It can further be proved that by obtaining the disparity of a pixel, its depth can be calculated using the intrinsic parameters of the camera, which will be explained in more details in the next sections.

The traditional methods of stereo matching are usually categorized into two main groups: local methods and global methods. The local methods use a supporting window and a similarity function to do a similarity search across the stereo images to find the corresponding pixel for each pixel in the corresponding stereo image [30][31][32]. Global methods, on the other hand, optimize a global cost function for the entire image to obtain an approximately good disparity label for each pixel [33][34][35][36][37][38][39]. Generally speaking, local methods are faster but produce lower quality depth maps, while global methods are slower (more computationally intensive) but tend to produce more accurate results. There is also a third approach called semi-global matching, which tries to find a trade-off between local and global methods [40][41].

Modern day stereo matching algorithms, however, are mostly based on neural networks which either directly learn a mapping from the stereo pair to the depth map [25][26][27] or indirectly by first computing a cost-volume from the stereo images, then using a neural network to find a mapping from the cost-volume to the disparity/depth map [28][29].

In the rest of this section, we introduce a general taxonomy of stereo matching algorithms which can be shown as a four-stage pipeline. Figure 2.5 shows a diagram of a

typical stereo pipeline. All the traditional stereo algorithms tend to implement all or a subset of this pipeline [24].

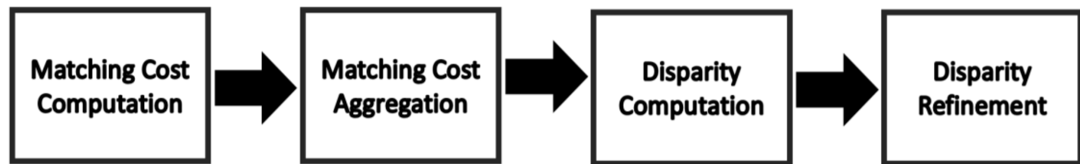

Figure 2.5 *A typical stereo matching pipeline [90].*

In the first stage of the pipeline, a similarity search is performed over two images to form an initial estimate of similarities among horizontal (or vertical) pixels of images. This initial estimate is called cost-volume. In the second stage, extra post processing and filtering operations are performed on cost-volume to further improve the accuracy of the initial estimate. In the third stage, the best disparity found so far is picked as the most potential candidate for each pixel in the volume to form an initial disparity map. In the fourth and final stage, we try to further improve the initial disparity map using optimization algorithms. In the rest of this section, we explain each stage of the pipeline in greater detail.

Computing the matching cost

The first step in computing matching cost is to select a similarity function, which is used to measure the degree of similarity between two pixels in the left and right images, respectively. There are a number of similarity functions with their own specific properties. One of the most commonly used functions is squared intensity difference (SID). The SID formula is as follows, where L and R are left and right images, respectively:

Equation 4: $\boldsymbol{C_{SID}(x, y) = \left(L(x, y) - R(x, y)\right)^2},$

Another common option is absolute intensity difference:

Equation 5: $\boldsymbol{C_{AID}(x, y) = |L(x, y) - R(x, y)|},$

In some stereo matching algorithms, matching cost computation and cost aggregation steps are merged into a single step. This allows for employing similarity measures that work over a range of pixels rather individual pixels. Similarity measures based on correlation coefficient [42] and more recently proposed functions based on Siamese networks [43][44] are such examples. *Figure 2.6* shows an example of Siamese network which applies a series of convolutions on local patches of left and right images before using a dot product to produce the similarity score.

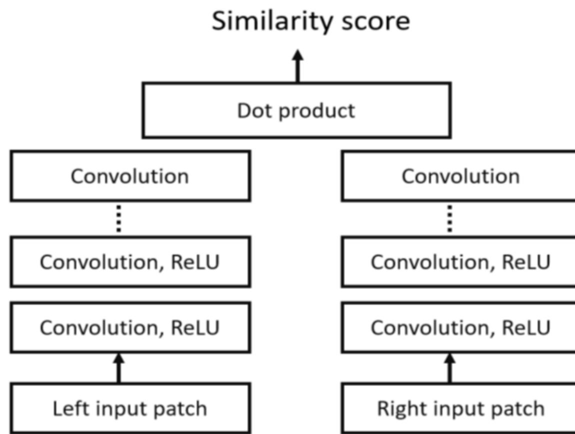

Figure 2.6. The similarity function based on Siamese network [44].

It is important to note that in the context of multi-view stereo (MVS) the similarity function is referred to as the **photo-consistency** function.

After selecting the similarity function, in the next step, we need to find the matching points between two stereo images to compute the initial cost-volume. The matching point can be found using the constraint induced by the epipolar geometry between two images. Figure 2.7 shows the general setup of epipolar geometry.

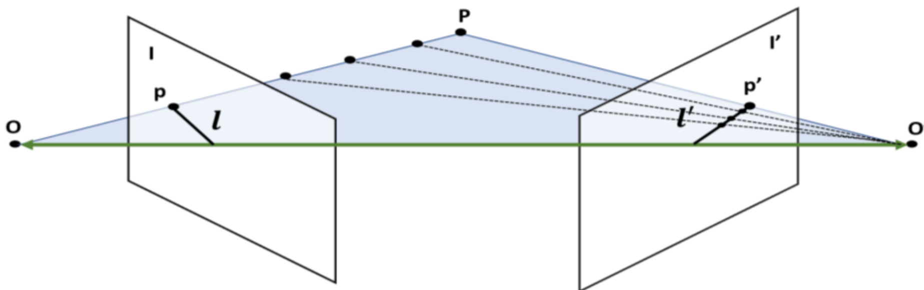

Figure 2.7 *general setup of epipolar geometry [45][89].*

In a more general case, two cameras can be placed in any arbitrary position to each other. Here, the image plane of each camera is placed in front of its principal point, and two principal points (O and O') are connected through a green line called the baseline. The 3D point P, which is observed by both cameras, is projected on each image plane I and I' to form points p and p', respectively. By passing plane from the baseline and the point P and intersecting it with each image plane (I and I') we will obtain two lines (l and l'), which are being referred to as epipolar lines. By computing the essential matrix, it can be shown that points on one epipolar line correspond to the points on another epipolar line [45]. In other words, given the point p, in order to find the corresponding point p', we only need to search line l' instead of the whole image.

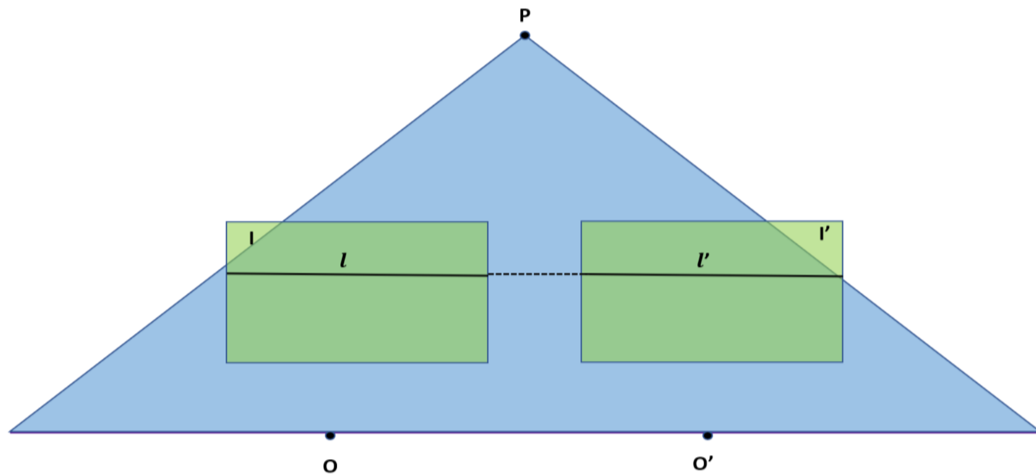

Figure 2.8 *epipolar setup for stereo matching system [89].*

In the case of stereo matching with two horizontal cameras, since baseline is parallel to the both image-plane, Figure 2.8, the epipolar lines l and l' would become the corresponding rows of the two image planes. Now by taking one image as the reference (usually the left image L), we can generate a three-dimensional matrix called cost-volume with the same height and width as the input image and the same depth as the number of the disparity levels. Considering this, each (x, y, d) entry of this matrix contains the similarity value at the disparity level d for the pixel (x, y) in the reference image. The process is better shown in the following equation:

Equation 6: $\boldsymbol{C(x,y,d) = similarity_function\big(L(x,y) - R(x-d,y)\big)}$

Computing Cost Aggregation

If we use the initial cost-volume created in the previous stage directly to compute the disparity, it will only generate a distorted (noisy) disparity map similar to the one in Figure 2.9-left, whereas the ground truth of the same scene is smooth and noiseless (Figure 2.9-right).

We aim to remove the noise at the cost aggregation stage and improve the result by applying an aggregation function on the cost-volume produced in the previous stage. The aggregation function is defined based on the assumption that disparity/depth values change smoothly across the image and only change sharply near the boundary of the objects. This implies that there is a high correlation among the disparity values of neighbouring pixels. As a result, the aggregation for each pixel is performed over a local region captured by a sliding support window which filters the entire cost-volume.

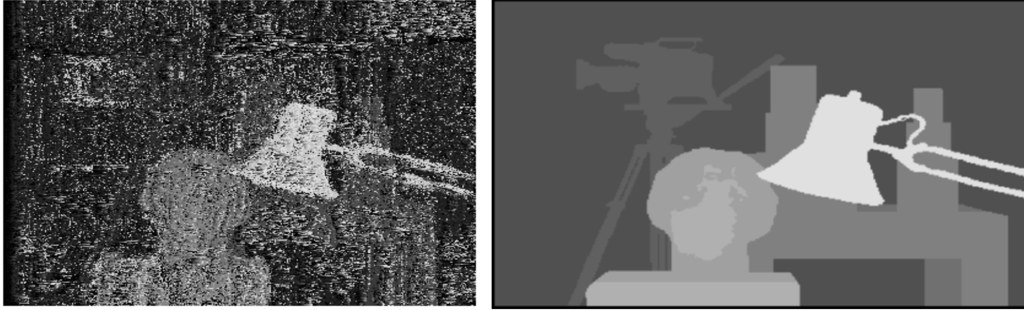

Figure 2.9. *disparity map without aggregation (left) ground truth (right) [90].*

The local window can come in different sizes (e.g., 5 x 5 or 11 x 11), different shapes (one to three dimensions), and different weights (uniform, Gaussian) [46][47]. In some cases, we may use color proximity as a weighting factor for disparities or adjust the size of the local window to generate more robust results [48].

A simple example of an aggregation function is the box method with a N x N window, where N = 2r. The formula is as follows:

Equation 7: $\boldsymbol{C_A(x,y) = \frac{1}{N^2}\sum_{n=-r}^{r}\sum_{m=-r}^{r} C(x+n, y+m)}$

Treating the cost-aggregation as a filtering operation, where each slice of the cost-volume across the disparity axis is considered as a depth map, could quickly lead to a computational inefficiency once the range of the disparity levels increases. The higher the disparity range, the more depth map slices needed to be filtered. For the simple box method, this issue has been addressed using the well-known SAT (summed area table) method, which is proposed in [49].

The algorithm works in two stages. First, for each pixel location, the following summation will be done, and the results are stored in the table S, which has the same size as the image:

Equation 8: $\boldsymbol{S(x,y) = \sum_{n=0}^{x}\sum_{m}^{y} C(n,m)}$

In the second stage, the aggregation cost at each point is calculated using the following formula:

Equation 9: $\boldsymbol{C_A(x,y) = \frac{1}{N^2}\big(S(x+r, y+r) - S(x-r-1, y+r) - S(x+r, y-r-1) + S(x-r-1, y-r-1)\big)}$

Disparity Computation

In local methods, most of the focus is on the first two stages (matching cost computation and cost aggregation), and the disparity computation is simply done by choosing

the disparity with the minimum cost at each pixel location (x, y): the so-called WINNER-TAKES-ALL approach [24].

Global methods, on the other hand, tend to ignore cost aggregation or use it as a simple initialization process [24][45]. The reason for this is those global methods, unlike local methods, enforce a global smoothness constraint, which removes any need for enforcing local smoothness by the cost aggregation step.

Contrary to local methods, global methods do most of the job during the disparity computation stage, where a global energy (cost) function is minimized to enhance the quality of the disparity map to a high level. The objective is to assign a disparity label to each pixel in such a way that minimizes an energy function with the following form:

Equation 10: $\boldsymbol{E(d) = E_{data}(d) + \lambda E_{Smooth}(d)},$

where $E_{data}(d)$ measures how well the current disparity d does fit in and $E_{Smooth}(d)$ measures the cost of assigning two different labels (i.e., $d_p$and d_q) to two adjacent pixels.

2.1.3 Multi-view Stereo

Multi-view depth estimation is a general term used for describing a set of algorithms and methods which solve the stereo correspondence with more than two images. Although one might consider MVS a natural improvement to stereo methods, as the additional number of views would lead to the generation of a more accurate depth map, the main goal of the MVS is to reconstruct the 3-D geometry of the scene [45]. As a result, the MVS algorithms follow a slightly different pipeline than that of the stereo, which we have introduced so far. In this section, we limit ourselves to the depth estimation aspect of the MVS and highlight some of its key features as well as similarities and differences with respect to stereo methods.

The first difference between MVS and stereo vision is the way they represent the scene's geometry. While the standard binocular stereo generates a single depth map by choosing one of the views as a reference view, an MVS approach would generate a depth map for each one of the views. In addition to depth maps, MVS methods would also use different ways to represent a scene's geometry such as voxels, level sets, and polygon meshes. In this thesis, we mainly focused on the type of MVS algorithms that works with depth map [50].

Another critical difference between MVS and stereo matching is the camera setup. In the MVS, several cameras can be placed at any distance and angle from each other to capture different parts of an object. One of the consequences of such camera setup is

that the input images, unlike the stereo matching, are no longer rectified. For this reason, it is no longer beneficial to use disparity as a substitution for depth since we need to perform a 2D search on all other views to find the corresponding points. Therefore, for MVS algorithms, it is more common to work directly with depth rather than disparity for which we only have to perform a 1-D search on a range of depth hypotheses to find the corresponding point on each view [51].

One advantages of the MVS method over stereo vison methods is in the photo-consistency (similarity function), where adding extra views creates a better cost-volume. Given the disparity/depth d, the cost-volume for the MVS method is obtained by iteratively performing the following equation over different values of x, y, and k.

Equation 11: $\boldsymbol{C(x,y,d) = \sum_k similarity_function(I_k(x,y), I_r(x,y), d)}$

Here, $C(x,y,d)$ is obtained by summation of photo-consistency values of all the views.

As MVS methods look at a scene (object) from a different perspective, they have to deal with the issue of visibility and occlusion (Figure 2.10). Photo-consistency computation cannot lead to a correct result if it is performed over an occluded region. As a result, occlusion modeling needs to be done to determine which views are visible and can be used for photo-consistency [50].

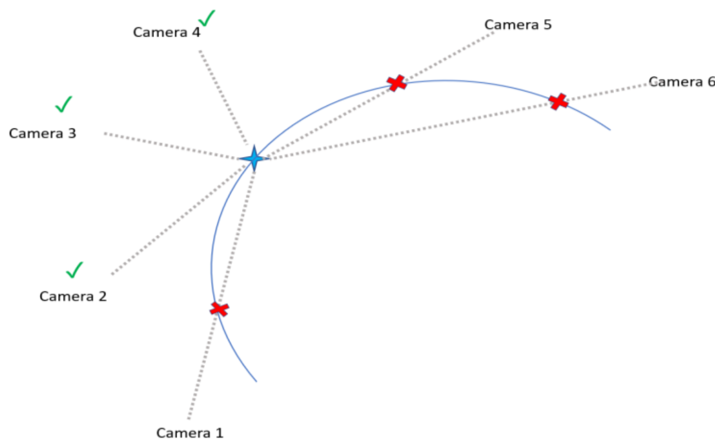

Figure 2.10. *visible and occluded camera in MVS camera setup [51]*

2.2 Image-based Rendering

Creating a computer-generated virtual environment that simulates the real world and gives users an immersive experience is one of the long-standing goals of virtual reality. Achieving such a goal requires modeling and rendering an entire environment from any arbitrary point of view. In this regard, constructing and representing 3D objects as well as free-viewpoint navigation within the environment are considered as two of the most fundamental challenges within this field [45].

Modeling and rendering of a 3D scene have been traditionally done using computer graphics algorithms through 3-D geometry modelling followed by model-based rendering. While modeling is concerned with extracting the geometry and illumination characteristics of the scene to form a 3D model, the rendering is about performing computation on this model to generate a desired photo-realistic view [52]. The rendering of the view can be done based on several variables such as user's position, illumination, and color. Such rendering techniques include illumination/shading computation, geometric cropping, and simplification of the reconstructed scene [53].

The strength of model-based techniques lies in the accuracy of their model and flexibility of the rendering. A 3D model is expected to represent the complete scene geometry. Rendering also can be applied with different specifications on the same model to create different images. The downside, however, is the high computation cost. As the size of the scene and/or complexity of the objects grow, the complexity of the 3-D model increases, which makes it more time-consuming to obtain. Because of this, traditional graphics-based techniques are not preferred for rendering real-world scenes where the 3D model is hard to obtain. These techniques have applications in computer-aided design, game development, and other engineering disciplines which employ synthetic scenes, where the geometrical models are pre-defined.

Contrary to model-based rendering methods, there are image-based rendering methods that use multi-perspective images as the fundamental building blocks to achieve the same goal. One of the direct consequences of using such approaches is reducing the computation time. The reason is that, first, traditional methods use complex geometry modeling and rendering, whereas in IBR we perform only simple operations and analysis on a set of images. Second, in IBR, computational complexity does not increase with the increase in the complexity of the objects and their relationships. And finally, since we directly render the scene from the images, the generated views are already photo-realistic, and there is no need for additional computation on that. All these advantages together make IBR a very suitable alternative approach to traditional computer graphics-based techniques.

The IBR can be defined as the problem of reconstructing a non-existing view of the scene from a set of already existing views captured by a system of cameras. Generally, IBR techniques can be categorized in two groups, depending on the amount of the geometry information available, namely rendering without geometry and rendering with geometry [53]. In some references, the rendering with geometry is further broken down into rendering with implicit or explicit geometry [52][54]. Figure 2.11 shows the whole range of different IBR techniques.

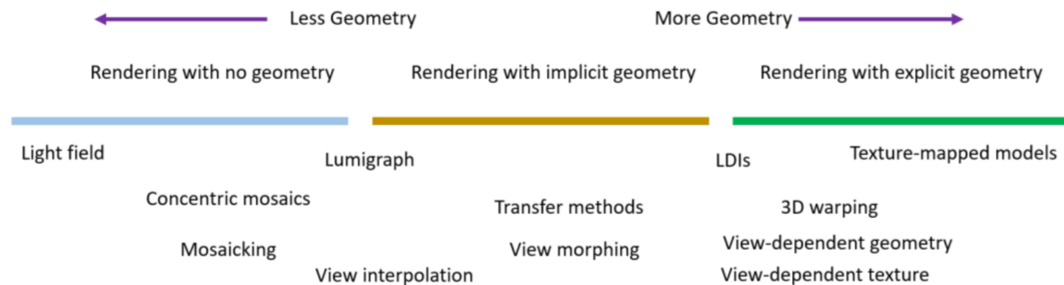

Figure 2.11. *The nomenclature of IBR techniques [52]*

Regarding rendering without geometry, a new scene is entirely reconstructed from multi-view images and videos using a special formalism based on the plenoptic function. Originally proposed in [55], a plenoptic function is a 7-dimension function that can describe the intensity of light at any given point, direction, and wavelength in the scene [56]. A new scene view can be rendered from the plenoptic function, which has to be reconstructed first from its sampled version (i.e., a set of multi-view images). Prominent examples of this type of rendering are techniques based on light field rendering and lumigraph [57][58], as well as Concentric Mosaics [59].

Geometry-based methods, on the other hand, incorporate the geometry of the scene to reduce the number of required images. Implicit geometry methods, for example, would use feature correspondences, e.g., point-to-point correspondences, to generate a new view without directly working with the depth information. Examples of such methods are view interpolation [60] and view morphing [61]. The explicit geometry methods, however, require receiving the whole geometry information of the scene, usually in the form of a depth map. Layered Depth Images [62] and 3D Warping techniques [63] are prominent examples of this class of rendering.

Image-based rendering, especially geometry-based methods, is often considered as a direct application of stereo vision. In this thesis, we target a view-interpolation method based on plane-sweeping principles to generate novel views from stereo images. Generally speaking, these algorithms heavily rely on point projection among different views as well as on camera calibration.

2.3 Parallel Computing

The early computers were originally designed based on the sequential model of machine languages to make them easy to program, even though the hardware itself is

inherently parallel [1]: the so-called serial illusion [2]. However, as time passed, the increasing demand for more computational power led to the unleashing of more and more parallel processing potential of the computers. The early parallelism was done at three different levels: bit-level parallelism, instruction-level parallelism, and data-level parallelism. These types of parallelism, however, were implemented implicitly by the processor itself without impacting the software development process at all [7]. In addition to implicit parallelism, the software community also heavily relied on rapid improvements in the clock frequency of microprocessors to consistently enhance the performance of their software applications. The same serial version of the code could now enjoy faster execution on the next generation of hardware without programmers being needed to do anything regarding the performance of the software. As a result, the development of new software applications became completely dependent on the recent advancements in hardware, not better software engineering.

Things for the software community, however, started to change in 2003 when computer vendors could no longer increase the clock frequency of their processors easily: mostly due to issues like power consumption and heat dissipation [3]. To fill the performance gap, computer architects developed new models based on multiple computing units, known as processor cores, and created a new form of parallelism called thread-level or task-level parallelism. Unlike the previous types of parallelism, thread-level/task-level parallelism requires direct intervention of programmers, affecting many aspects of software development: algorithm design, implementation, and debugging, to name but a few. From that time onward, only software written in parallel could enjoy the benefit of newer generations of hardware. It was a start of a paradigm shift for the software development community, which is known today as the concurrency revolution [4]. One of the consequences of this shift was to make programmers become more aware of the architecture specifications of the processor under the hood to develop their applications based on that accordingly. Developing a good software application based on hardware requires better resource allocation [5] and assign each task to the best type of processor or processor core [6]. The latter is a key component in the embedded systems programming and will be explained in more detail in the following sections.

2.3.1 Heterogeneous Computing

Different types of algorithms present a different range of computational behavior. Some algorithms, such as searching or parsing, are very control-intensive. This means that these algorithms use a high number of if-then-else statements within their coding structure. Unlike arithmetic operations like addition and multiplication, control operations are

handled by the control unit (CU) of the processor rather than the arithmetic logical unit (ALU). This class of algorithms is known to be very hard to parallelize. In contrast, data-intensive algorithms, which process a vast amount of data, tend to be more arithmetic oriented and more parallel-friendly [7]. Most of the algorithms in computer graphics, image/signal processing, computer vision, and deep learning [8] belong to this category of computation. There is also another branch of the algorithms, which is known as compute-intensive. In this class of algorithms, lots of processing is being done on a limited amount of data. Many iterative algorithms, such as those in numerical methods and financial modeling, belong to this category of computation.

In order to optimally cover this wide range of computing domain, different computer architectures were developed with each of which being excelled at only one or few aspects of computation. Superscalar CPUs [7][11] with a large control unit and high clock frequency were designed to handle-control intensive tasks, while GPUs and vector processors, where a single instruction is executed for multiple data (SIMD), tend to work better with data-intensive applications [9]. Field programmable gate arrays (FPGAs) are another type of platform that can be programmed either by HLS (high level synthesis) using instruction-based languages such as C/C++, system C, and OpenCL or in traditional ways using Hardware Description Languages (HDL) like Verilog and VHDL. FPGA would allow developers to design and optimize an architecture of their choice, the one that perfectly fits into the computational structure of their application. Another advantage of FPGA is their low energy consumption compared to GPUs, while their downside is their low clock frequency [10]. Digital signal processors (DSPs) are another worth mentioning spectrum of microprocessors, which are commonly used in embedded systems such as smartphones to perform tasks such as audio processing, image encoding/decoding, and voice recognition [7].

The problem, however, comes from the fact that each application is a mixture of different types of computation, and there is no single best device to address them all perfectly. For example, an application might possess some areas which are control-intensive, therefore more suitable to be executed on a CPU, while it has some other areas which have lots of bit-wise operations and are more suitable for an FPGA platform [11][12][13]. The solution is to use multiple processors, and the challenge is to find the right combination of devices to solve the problem optimally. Considering this, the programmer's task is to map their applications on a wide array of architectures to find the best performance: execution time, power consumption, or both depending on the goal. This type of computation where comprising tasks of an application are divided among

processors with different architecture within the same framework is called **heterogeneous computing**.

2.3.2 Heterogeneous Computing for parallel processors

Heterogenous computing comes with an inherent opportunity to exploit parallelism in applications as it can combine parallel and serial processors in one framework. Since 2003 and the beginning of the concurrency revolution, computer architects came up with two main trajectories for their newly designed parallel processors: multi-core and many-thread. Multi-core approach tends to utilize parallelism within the application while maintaining good performance on the sequential parts of the code. Examples of such are the latest series of intel core-i family for personal computers and Xeon family for servers [14]. In these processors, each core has its own program counter and executes a full x86 instruction set [15]. Many-thread approach [3], on the other hand, tends to sacrifice the sequential performance of the code entirely in the favour of parallelism by dedicating a huge portion of silicon area (die) to arithmetic cores. Processors of this type are often referred to as massively parallel processors. The prime example of such is graphic processor units (GPUs), which are originally developed for the video game industry, but later found their way to high-performance computing community. Today modern GPUs like Nvidia RTX 2080Ti possess approximately 4000 cores with floating point performance of 13.45 teraflops. This number would become even more stunning when it is compared with the performance of the latest intel core i9 series which is no more than 1.3 teraflops [16].

A real-world application, however, is neither purely parallel nor sequential but a mixture of both. In fact, a typical program consists of both sequential and parallel parts, with the sequential parts taking the most volume of the code, while the parallel parts take the most execution time, as illustrated in Figure 2.12. GPUs tend to perform very poorly in the face of the sequential portion of the code, whereas CPUs, with their high working frequency and sophisticated control logic tend to work the best. This implies that neither GPU nor CPU alone is not enough to address the heterogeneous nature of the parallelism, while a joint CPU-GPU framework is a natural option to address this problem.

2.3.3 GPU Architecture

GPU’s architecture finds its roots in demand in computing 3-D graphics, where a large number of arithmetic operations had to be done in real-time. To gain a deeper insight

into the architecture of GPU, in this section, we contrast it against the CPU's architecture, which in many respects plays a complementary role to GPU.

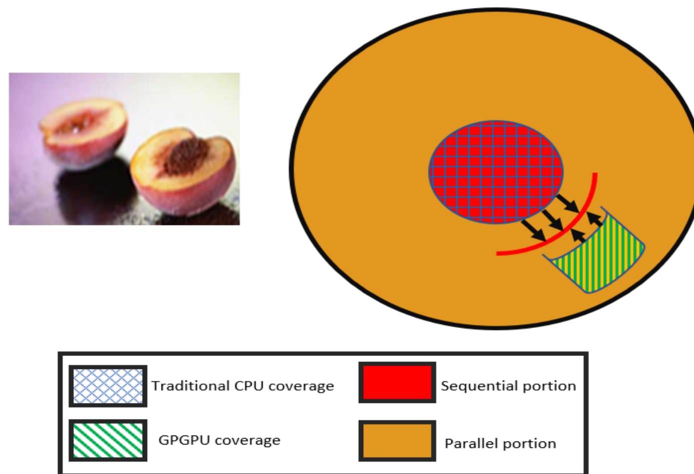

Figure 2.12 Distribution of sequential and parallel portion of an application [3]. Peach is the total execution time of application. Meat of the peach (orange) is the data-parallel part and pit of the peach (red) is the serial part of the core.

CPU and GPU follow two fundamentally different philosophies of design. The former focuses on minimizing the program's latency, defined as the time delay to perform each task [2]. This requires the employment of strong cores which are optimized for single thread performance of the code. As a result, a CPU core tends to possess a very high clock frequency and a more sophisticated control unit for strong ILP support, namely superscalar, branch prediction, and out-of-order execution. Moreover, a large cache size is being employed to cover up high-latency memory access operations which have the highest clock cycle count among all instructions. All of this, however, comes at the cost of consuming an increasingly bigger portion of the chip area and more power supply, which, in turn, dramatically reduces the number of the cores that can be used on the chip. As a matter of fact, most of the modern-day CPUs do not possess more than thirty cores on their chip [19].

GPUs' architecture, on the other hand, has been optimized to maximize the throughput of the program. The throughput is defined as the number of tasks being done in a given amount of time. Unlike latency, which is the unit of time per unit of the task, throughput measures the unit of work (number of tasks) per unit of time. The consequence of such an approach would lead to a significantly different design where the number of the cores is prioritized over the computing power of each core. This, however, can come at the cost of increasing the latency of each individual thread, causing the processor to perform poorly in the presence of a low amount of parallelism, unlike a latency-based design which tends to work poorly in the presence of a high amount of parallelism.

The schematic difference between CPU's and GPU's architectures has been demonstrated inFigure 2.13. As it can be seen, a GPU is almost entirely made up of arithmetic cores that share control logic and cache memory among themselves. The GPU cores, though, are not nearly as powerful as their CPU counterpart. They are smaller in size (made up of fewer transistors), work with less clock frequency, and do not have their own independent control unit. As a result, they do not do very well in the face of complex tasks where lots of control flow is involved. However, what they lack in speed and complexity, they compensate with numbers. Due to their huge number of cores, GPUs are capable of computing a massive number of arithmetic operations at the same time, maximizing the throughput of the program. As a result, GPUs tend to be extremely good at matrix and vector computation where lots of arithmetic operations are being done.

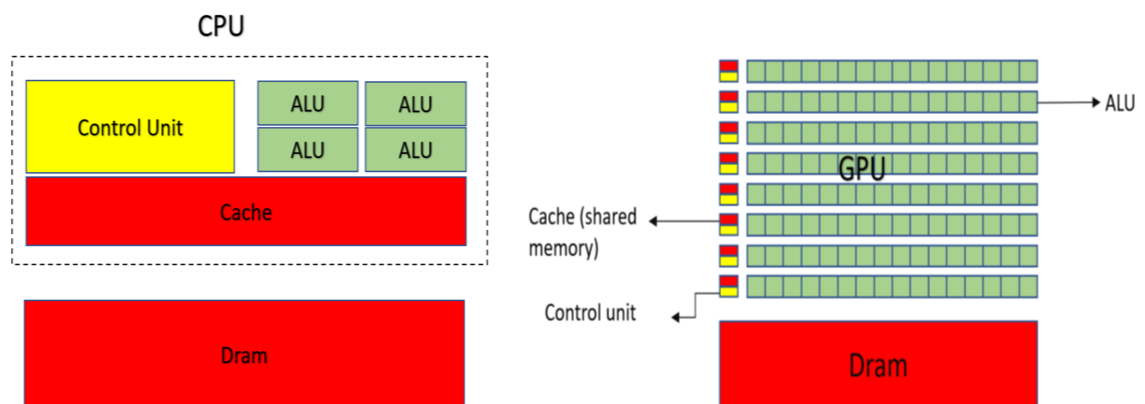

Figure 2.13. the difference between CPU and GPU architecture [3].

Another major difference between CPU and GPU is the cache memory. Cache memories are great tools for decreasing the instruction and data latencies by fetching them before they are needed. However, GPUs tend to dedicate most of their die to arithmetic cores; hence, they do not possess a big cache size. In GPU, long latency operations, such as memory accesses, are being addressed in two other ways. Firstly, by taking advantage of a large number of available tasks, GPU can keep itself occupied doing the rest of the tasks in the queue while data for other groups of tasks is being fetched from memory: the so-called **latency hiding**. Secondly, GPUs tend to compensate for the long-latency memory operations by employing a very high memory bandwidth that exclusively focuses on the throughput. The GPU memory bandwidth can fetch a big chunk of data all at once. It might not fetch each data as fast as the CPU does, but it loads a big chunk of data in consecutive memory locations much faster. As a result, if applications can use consecutive data memory elements, the data can be loaded into the device much faster. In GPGPU, the techniques which allows programmers to do so are called **memory coalescing** techniques [3]. A limited cache memory is only

provided to control the data access congestion on the bandwidth, so that multiple threads do not access the same memory location.

Computer Organization of GPU

GPUs from different manufacturers, e.g., Nvidia and AMD, can have different computer organizations. A typical Nvidia GPU processor is organized as an array of computing units called streaming multiprocessor (SM). Each SM is a complete and independent processor. It has its own cache, control, and arithmetic logic unit and can work in any order to other SMs, allowing the GPU to support thread-level parallelism. SMs are further organized into building blocks called processor clusters (PC); Each PC can contain one or more SMs depending on the architecture of the device [20].

SMs are independent of each other but not isolated. They share a slow high-bandwidth off-chip global memory and fast on-chip L2-level cache memory, which can be accessed by all the cores within the device. These two memories are a place for SMs to collaborate and share data with each other. The global memory is separated from the CPU's RAM, which is located in the computer's motherboard. Both global and RAM memories use a PCI-Express bus to transfer data between each other.

Figure 2.14 shows a typical architecture of a SM. Each SM has a dedicated cache memory, which is partitioned into two different areas: The L1 level cache for fast instruction fetch and shared memory for low latency data access and thread-level collaboration. Furthermore, each SM has a large register bank to accelerate thread-level computation. SM is a heavily multithreaded processor. It contains an array of SIMD cores called streaming processor (SP). SP is a small vector processor capable of DLP (Data Level Parallelism) support [2].

In Nvidia GPUs, each consecutive 32 threads form a computing unit called **warp,** and each SM can handle at least one warp per cycle. All the threads of a warp are executed in parallel by the SM. The unit called warp scheduler is provided to implement latency hiding by switching between stalled and eligible warps [93]. A stalled warp is a warp whose instruction cannot be issued (mostly due to the high memory access latency), and eligible warp is a warp whose instruction is ready to be executed. The key point here is that all the threads within a warp share the same control path (same warp scheduler and program counter register, for example), while each of them having their own arithmetic core. This means that they load and execute the same instruction with different operands at the same time. The terminology used by NVIDA to describe this model of execution is called **SIMT** or Single Instruction Multiple Thread. It is a similar

concept to **SIMD** with this difference that a single instruction is being executed for multiple thread not multiple data.

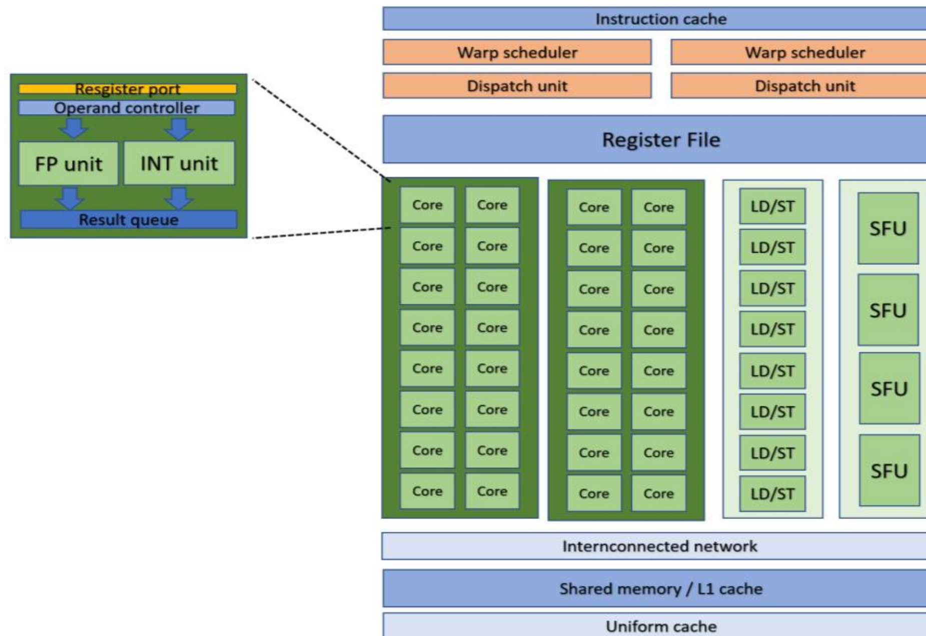

Figure 2.14. computing organization of an SM [20].

2.3.4 Open Computing Language

Open computing language (OpenCL) is a programming framework for parallel computing in heterogeneous environment. Initially, a project supported by Apple, it was soon joined by other major microprocessor vendors such as NVIDIA, Intel, and AMD. As a result, OpenCL is platform independent. It supports a wide range of devices, and its code can be executed on any combination of these devices.

OpenCL introduces a number of API functions to interact with these devices. A general OpenCL program is made-up-of four parts, which are being referred to as models: The Platform model, the execution model, the kernel programming model, and the memory model. In the rest of this section, we explain each of these models and introduce their API functions

Platform model

The platform model determines the topology of the computing system. A typical heterogeneous system consists of a host processor, usually a CPU, and a number of devices. The host is the main processor of the system and is responsible for launching the execution of the program. Devices are co-processors, which are working in accordance with the host. OpenCL platform model has been designed to model such a framework. As a result, an OpenCL program has a host side of a code, which contains the main program and runs by the host processor and kernels. Kernels are OpenCL functions

that are executed on the device. In a typical CPU-GPU platform, the CPU is the host, and GPU is the device. CPU starts and finishes the execution of the program and can decide when and when not to use the GPU.

In addition to defining the relationship between host and device, the platform defines an abstract hardware model for the devices (**Figure 2.15**). In this abstract model, each device consists of an independent block called compute unit, and each compute unit contains at least one processing element. In the case of an NVIDIA GPU, we discussed earlier, each SM is a compute unit, and each SP is a processing element.

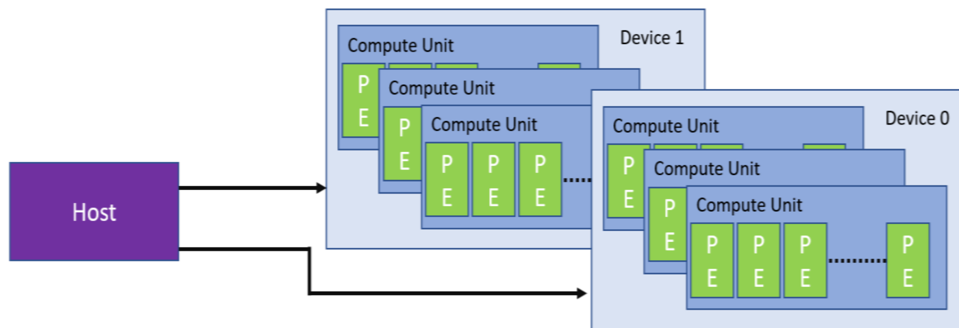

Figure 2.15. OpenCL Abstract hardware model [7].

The platform is an implementation of OpenCL. This means that we have a number of devices on our system, and a specific platform may consider only a subset of the devices to work with. For example, an NVIDIA-made platform may not recognize the Intel CPU installed on our system because it is from a different vendor, and its instruction set is not known by NVIDIA.

Execution model

After determining platform and devices within it, OpenCL program proceeds with an execution model to run kernel codes on the set of devices. In order to do that, an object named context is created. Context is an abstract framework for defining and managing other OpenCL objects, which have something to do with the execution of the kernels such as memory and kernel objects. The API function for creating context is as follows:

```
cl_context
clCreateContext (
const cl_context_properties *properties,
cl_uint num_devices,
const cl_device_id   *devices,
void (CL_CALLBACK *pfn_notify)(
    const charr *errinfo,
    const void *private_info,
    size_t cb,
    void  *user_data),
    void *user_data,
    cl_int *errcode_ret)
```

Context is platform specific and take devices of the platform as an input argument. This means that it can be defined for any number of devices available within the platform, but it cannot be defined for devices associated to other platforms. The API function clCreateContextFromType() is also provided to create context for all the devices of the same type (all GPU or CPU devices) within the platform [7].

After defining the context, we can use it to define several other objects to initiate, configure and execute our kernel codes. For example, we can use memory objects like buffers (cl_mem) to allocate memory on devices' memories or kernel objects (cl_kernel) to call our kernel functions. OpenCL kernel codes usually written in form of string. An object with the type of Program can be used to read the entire string code, compile it, and store it to be used by other objects later.

Another important aspect of execution model is a communication mechanism called **command-queue**. A command-queue object is created to enable the host to directly command a specific device within the context to do a certain action. For this reason, a unique command-queue object has to be created for each single device within the context. There are three types of command which can be done by a command-queue object: memory commands, kernel execution commands, and synchronization commands. Memory commands are being used to transfer data between host and device memories. For example, the API function enqueueReadBuffer() is being used to transfer data from a certain memory location in device to a memory location in host. Kernel execution command can call the kernel function from the host side of the code. The API command for doing this is enqueueNDRange(). The synchronization commands, however, are not submitted on the queue. They are basically barrier operations designed to synchronize the activity among different command-queue objects and host code. The API functions are Finish() and Flush(). The Finish() function, for example, halts the execution of the host code till all of the commands within the command-queue is completed.

Kernel Programming Model

Kernels are parts of the OpenCL application which are executed on the device. For this reason, they are often called device side of the code in contrast to host side, which contains all the API calls and runs on CPU. From the syntax point of view, kernels are very similar to a standard programming language such as C/C++ with some additional keywords which enables the OpenCL features to execute a code on device. For example, the term global and local might be used to define a memory pointer on either global or local memories of the device respectively; or the keyword "kernel" can be used right

before the definition of the function to distinguish it from a standard C function. The most important difference, however, is that the kernel execution happens in parallel, while normal C/C++ code does not. In fact, kernel execution follows an abstract concurrency model defined by the Kernel Programming Model.

The OpenCL concurrency model can be explained as a two-level coarse-to-grain hierarchy of units, namely work-items and work-groups. In the first level, there is a grid of work-items. A work-item (also called thread in CUDA literature) is smallest unit in the concurrency model, which represents an independent task in our kernel function. In the broader picture, all the compromising tasks of an application must be mapped on an array of work-items, generated by NDRange(), in host side of the code before kernel launch. Consequently, a copy of kernel function is being executed by each one of the work-items within the grid in a single parallel phase: a well-known style of programming called SPMD (Single Program Multiple Data) [2].

In the kernel function, each work-item is identified by three indexes provided by the OpenCL intrinsic function get_global_id(). The get_global_id(0), for example, would output the index in the x-dimension of the work-item that calls and it the same applies for get_global_id(1) and get_global_id(2) which output the y and z dimensions of the same work-item respectively. Here, we show a simple example of an array addition with OpenCL and compare it a with a standard C implementation to demonstrate how the indexing works. Figure 2.16 shows a diagram of a simple vector addition algorithm and Program 1 is a standard C code of the same algorithm.

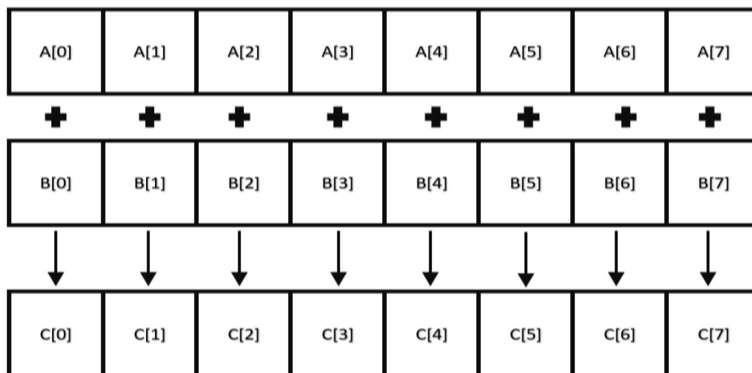

Figure 2.16. vector addition diagram.

```
void vector_addition_sequential(float *A, float *B, float *C)
{
for (int i = 0 ; i < 7 ; i++)
   C[i] = A[i] + B[i]
}
```

Program 1. *A serial vector addition*

By considering each iteration of *for* loop as a work-item, we reimplement the algorithm on OpenCL in Program 2. As it can be seen the for loop is omitted.

```
void vector_addition_parallel(float *A, float *B, float *C)
{
    int i = get_global_id(0)
    C[i] = A[i] + B[i]
}
```

Program 2. *A parallel vector addition*

Work-items are isolated units which cannot collaborate with each other on their own. For this reason, they are further organized into units called work-groups or simply just groups, which enables the collaboration among member work-items. One of the most important ways of collaboration in a group is accessing the same local memory address by work-items of that group. The local memory, as we explain later, is a shared memory space dedicated for group collaboration through which work-items can share the preliminary results of their computation with each other.

In addition to local memory, groups also provide a robust runtime management mechanism called barrier synchronization, through which work-items within a group can coordinate their execution with each other using OpenCL barrier functions, e.g., barrier(). When a work-item reaches a barrier point within the code, it halts its execution until all the other work-items within a group reach the same point. After that they resume their execution at the same time.

Both the work-item and work-group are not actual hardware, but rather abstract units which model the parallel workload of our program. As a result, their number can well exceed above the number of available resources in our hardware. Therefore, resource management is being performed by Kernel Programming model to map these units on our device hardware. In fact, each work-item is assigned to a processing element and each work-group is handled by a compute unit. In the case of a Nvidia GPU, for example, a work-item is assigned to an SP (our device processing element) and a work-group (our device compute unit) is assigned to a SM.

Memory Model

The CPU (host) and GPU (device) have two physically distinct memories from each other. The CPU's RAM memory is located on the motherboard and can only be accessed by CPU, while GPU's global memory is on the graphic card and is dedicated for GPU alone. Before the kernel can be executed on the device, the input data has to be transferred from the host to the device memory. In OpenCL, this usually happens in two steps. First, a memory space is encapsulated (allocated) as what we refer to

memory object on device memory. Then, the data is copied from a host array to that allocated memory area on the device.

OpenCL has three types of memory objects which has different properties: buffers, images, and pipes. Buffer is equivalent to C/C++ array where data elements are stored in consecutive memory spaces. Image objects can store one, two, or three dimensional images with the same format as graphics applications. Pipe object is basically a queue data structure with FIFO (First in First Out) protocol with write and read endpoints. At the time, only one kernel can write and one kernel can read from two endpoints of the pipe. Here, we limit our explanation to the buffers since we do not use the other type of memory objects in our works.

The API call that creates the buffer objects is as follows:

```
cl_mem
clCreateBuffer(
    cl_context context,
    cl_mem_flags flags,
    size_t size,
    void *host_ptr,
    cl_int *errcode_ret)
```

The API call would take the associated context, memory flags (READONLY, WRITEONLY, AND ERADWRITE), size of data in bytes and a host pointer, which refers to the starting element of an input array on host memory, and returns a memory object of the type buffer as output. The final argument, *errcode_ret, is optional and can be used to return the type of error if any happens during the function call. The buffer object is visible to all devices associated with the given context and can be treated as pointer on the device side of the code.

The clCreateBuffer() function can also be used to transfer the data from the host pointer to the buffer object it creates. This is an implicit method to transfer data. However, there are also an explicit memory transfer API functions for transferring data between host and device. The advantage of using the explicit commands over the implicit one is merely the performance factor, since they transfers the data at faster rate. You can see an example of explicit memory transfer commands as follows:

```
cl_int
clEnqueueWriteBuffer (
    cl_command_queue command_queue,
    cl_mem buffer,
    cl_bool blocking_write,
    size_t offset,
    size_t cb,
    const void *ptr,
```

```
cl_uint  num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)
```

Memory Regions

The structure of the device memory in OpenCL (and also GPU) is divided into four different regions: global memory, local memory, private memory, and constant memory. ***Figure 2.17*** demonstrate a general overview of all memory regions. All regions are logically disjointed from each, and they are handled by the programmer. In this section, we briefly explain each region.

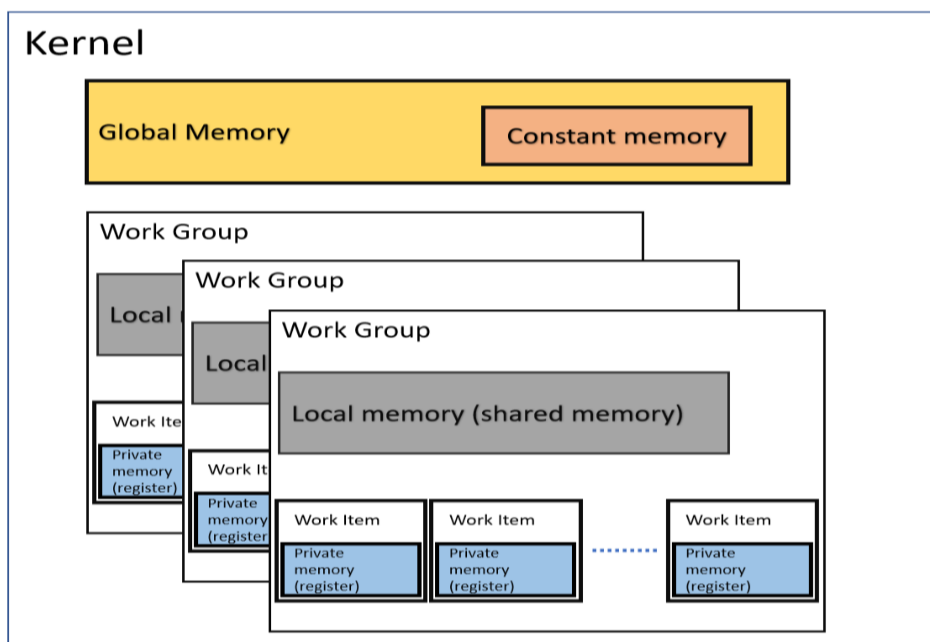

Figure 2.17. *The general diagram of OpenCL memory regions [7].*

Global memory is a type of a memory which being used to transfer the data between host and device. For example, if the data need to be transferred from the host memory to device memory, it will be loaded into the global memory. An important point about the global memory is that all the work-items (threads) within the kernel will see the same version of a global variable. Moreover, the lifetime of a global variable is entire application, which means that the data within the memory will not be deleted after the current kernel is terminated. Global memory is the main memory of the GPU. It has the largest size, usually in order of gigabyte, but longest latency due to the fact that is a slow cheap off-chip memory.

Constant memory is an area of the global memory whose values are constant throughout the kernel execution. Similar to global memory, it is allocated and initialized by host.

private memory, which is mapped to register memory in GPU, is the fastest type of memory with a very limited size. The scope of the private memory is each individual work-item. In other words, each work-item sees a unique private value. The lifetime of the private memory is during the kernel execution.

In addition to the mentioned memory regions, OpenCL since version 2.0 allows for expansion of the device global memory region to host memory through a technology called **shared virtual memory (SVM)**. SVM is particularly useful for pointer-based data structures (linked-list for example) which are defined on the host memory since there would be no way to transfer them to device memory. In addition to that, pointers defined on the host are only valid on the host memory, so transferring them to the device memory is meaningless. However, thanks to the SVM technology, it is possible to simply pass data structure pointers as an argument to the device memory.

OpenCL can use SVM in three different ways: Coarse-grained buffer SVM, Fine-grained buffer SVM, Fine-grained system SVM. Coarse-grained buffer SVM means that host and device share same virtual pointer at granularity of buffers. Here, the mapped and unmapped processes need to be done to update the host memory regarding the latest changes on the device. Fine-grained buffer SVM is done on buffer at byte-level granularity. It does not require mapped and unmapped processes. Fine-grained system SVM expands the fine-grained SVM to the entire host memory region, making the buffer object effectively useless since any pointer allocated by simple malloc() can now be accessed by OpenCL kernel.

3. METHODS

In this thesis, we introduce two different algorithms that we have implemented on GPU using the OpenCL. The first algorithm is a multi-view stereo depth estimation method, which is being used to estimate depth maps for images captured by camera array system and the second one is an Image-based rendering method based on plane-sweeping technique in [65][66][67].

3.1 Multi-view stereo on Sparse Light Field Data

The ideal goal of light-field technology is to capture the entire visual information of the scene by capturing every single ray of light emitted from the scene. This, however, is not practically possible due to the existence of almost infinite number of rays of light [68]. For this reason, we use technologies that perform sampling for light field acquisition. Two of these technologies are plenoptic cameras [69][70] and camera array systems [71][72][73][74][75]. An example of these two types of cameras are shown in Figure 3.1.

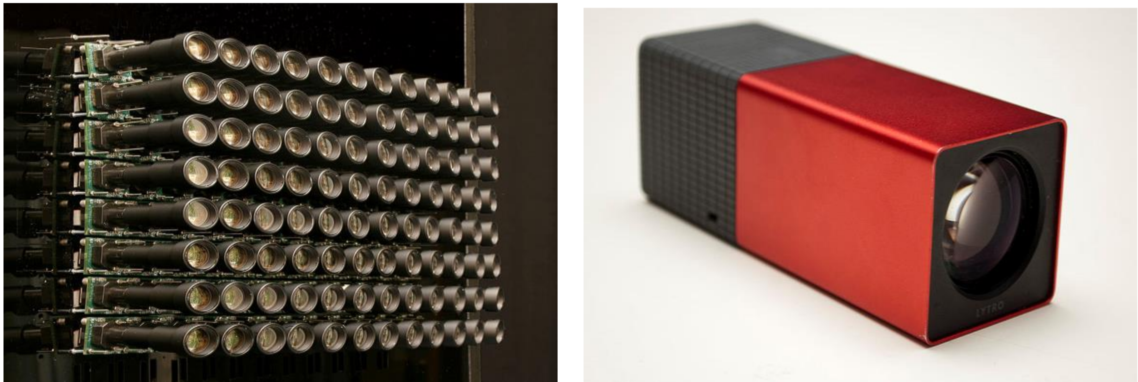

Figure 3.1. *Camera array system (left) [91] and plenoptic camera (right) [92].*

For depth estimation problem, since the input is a matrix of images, we can use MVS algorithms to solve the problem. The main MVS algorithm is based on plane-sweeping, which uses camera projection equation to estimate the initial depth values. Plane-sweeping has shown to have a good performance for simple datasets [76][77] but it tends to work poorly in the face of occlusion and textureless regions [78]. More complex but computationally intensive algorithms such as those based on patch-match methods and global optimizations (belief propagation and graph cuts) can be used to handle occlusion and textureless regions. In this section, we introduce a GPU-friendly MVS algorithm originally proposed in [78] to estimate the depth values for each pixel in

real-time. To reach this end, the proposed method uses superpixel instead of pixel as the main building block for depth estimation. Super-pixel is a compacted set of neighboring pixels of an image, which have a homogenous color [79]. An example of superpixels segmentation is shown in the Figure 3.2.

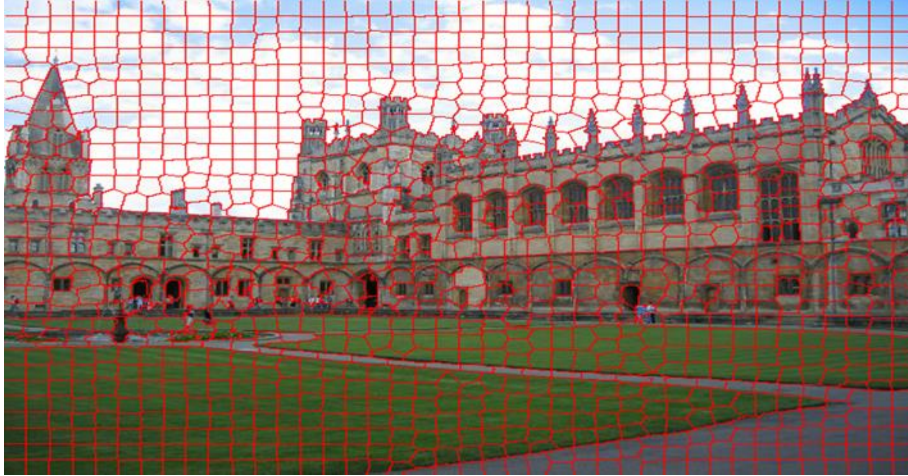

Figure 3.2. *an example of super-pixel pixel segmentation [82].*

There are several benefits to choose super-pixels over pixels. First, by using super-pixels, the total number of the computing threads in our GPU kernel is reduced dramatically, causing the program to consume less memory and run significantly faster. Second, since super-pixel is formed by grouping pixels with similar colors, the negative effect related to occlusion, noise, and presence of textureless region is reduced. Finally, by shrinking the size of input image, information can propagate faster across the grid, reducing the overall chance of converging to local optima [78].

The general diagram of the algorithms has been shown in the Figure 3.3. The input to the algorithm is a set of images taken by camera array system, and the output is a set of dense depth maps for each input image. In the first stage of the algorithm, superpixel segmentation is performed on the images to create a regular grid of superpixels called superpixel map. In the second stage, a plane-sweeping strategy is applied to estimate an initial depth value for each superpixel. In the third stage, we use an iterative optimization algorithm to refine the initial depth values. The key idea behind this algorithm is to model each superpixel as a plane, using plane equation. As a result, each superpixel can be shown with $(d, \bar{n})$, where d is the depth value at superpixel centroid and $\bar{n}$ is the normal vector. The goal of the optimizing refinement algorithm is to iteratively update the values of d and $\bar{n}$ for each superpixel to satisfy a certain cost-function. In the fourth and final stage, we use the refined planes to estimate the depth value for all the pixels in the image.

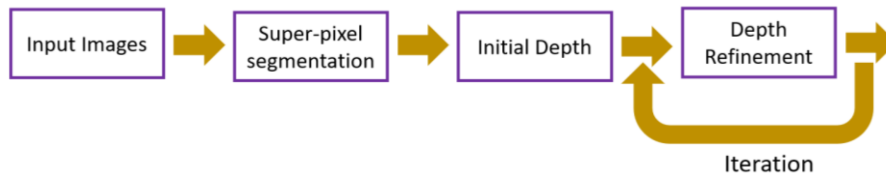

Figure 3.3. *Three stages of algorithms from left to right: superpixel segmentation, depth initialization, depth refinement.*

3.1.1 Simple Linear Iterative Clustering

The main job of the super-pixel segmentation is to create a more compact and sophisticated version of the image, while preserving its spatial content. Such segmentation method could effectively lead to a major performance increase, if it is added as a pre-process to the begging of an image processing pipeline. In this regard, several different super-pixel generation algorithms have been proposed with each of which having its own specification and performance requirements [79][80][81].

One of the most well-known algorithms for super-pixel segmentation is Simple Linear Iterative Clustering (SLIC). Originally proposed in [79], it uses an array of local k-means clustering algorithms across the image to assign each pixel to the closest and most similar cluster. In order to do that, SLIC defines a five-dimensional similarity function, which consists of two different components: the color and spatial components. The color component measures the color similarity between two clusters, while using **CIELAB** (L*a*b) as the system of color [45]. The reason for such a choice though is that L*a*b color space tends to differentiate pixels with similar colors better from each other. The spatial component, on the other hand, implicitly restrict the superpixel spatial range by comparing it only to the eight neighboring superpixels. The equation describing the distance function is described as follows:

Equation 12: $$d_{lab} = \sqrt{(l_k - l_i)^2 + (a_k - a_i)^2 + (b_k - b_i)^2}$$

$$d_{xy} = \sqrt{(x_k - x_i)^2 + (y_k - y_i)^2}$$

$$D_s = d_{lab} + \frac{m}{s} d_{xy}$$

The above equations compute the final distance D_s between two pixels k and i, using the color distance d_{lab} and the spatial distance d_{xy}. The variable m is being used as balancing factor between two distance components and basically determines the compactness of the superpixel. For example, if we increase the value of m, the weight of the spatial distances increases and more distant pixels receives more penalty, which in turn makes our superpixels more spatially isolated and compact.

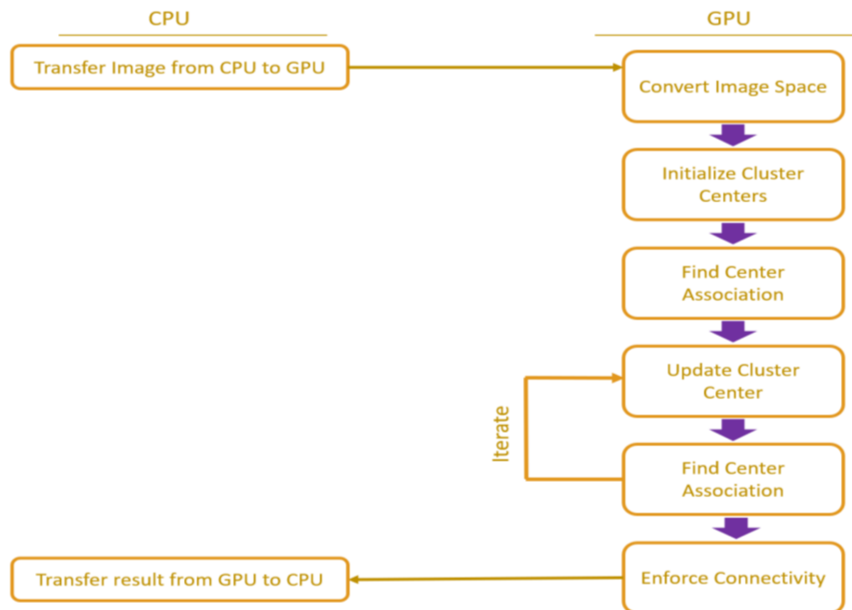

Figure 3.4. *The SLIC pipeline on GPU [82]*

The Figure 3.4 shows a general diagram of SLIC algorithm. The demonstrated pipeline has six different stages with the last stage, Enforce Connectivity, being an optional one. The pipeline outputs two two-dimensional matrices: Index image and superpixel map. The former contains the superpixel (cluster) index of all pixels within the input image and latter models the superpixel grid and hold the information of all superpixels such as color or location of center.

The algorithm starts by converting the RGB color space to CIELAB color space at the first stage and proceed with initializing the superpixel map and index image at second (Init_Cluster_Center) and third (Find_Center_Association) stage respectively. After that, the Update_Cluster_Center and Find_Center_Association procedures are iteratively called within a for loop to update the map and index matrices respectively.

The final stage, Enforce_Connectivity, although optional, would contribute greatly to the quality of the segmentation by eliminating the small isolated clusters which have only one or two members (pixels).

3.2 Depth Initialization

After partitioning our input stereo images into units of superpixels and organizing them in the form of regular 2-D grids which we named superpixel maps, in the next step, we assign a single depth value to each superpixel. This initial depth value is obtained by implementing a plane-sweeping strategy kernel function [76].

Plane-sweeping is the main algorithm for multi-view depth estimation, which unlike the traditional stereo matching methods works directly with the depth values rather than disparity. For this reason, it requires to receive camera parameters as input to perform the projection of pixels between different views. See Figure 3.5 for more details.

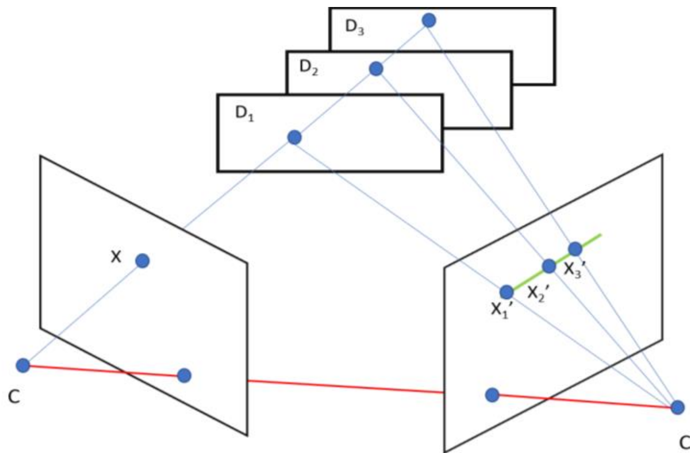

Figure 3.5. *Different projections of a single pixel for different depth hypothesis.*

For generating the depth map, plane-sweeping first assume a set of fronto-parallel planes in front of each image plane (view). Each plane is considered as a depth hypothesis. Then one at a time, it picks one of the views as a reference view and use all the available depth hypotheses, one by one, to project each pixel of the reference view to their equivalent pixels in the neighboring views, using the camera projection matrix. After that, it computes the accumulated photo-consistency between the current pixel of reference view and the projected pixels of the neighboring views. Eventually, it returns the depth hypothesis which gives the maximum (or minimum depending on type of the function) photo-consistency as the initial depth value for the current pixel of the reference view. For computing photo-consistency value, we chose **Truncated Square Sum Difference** (TSSD) as the cost function. The formula is as follows:

Equation 13: $$\boldsymbol{TSSD(p, p(d, \bar{n})) = min\left(T, SSD(p, p(d, \bar{n}))\right)},$$

Here, p is the current pixel in reference view, $p(d, \bar{n})$is its projection on a neighboring view, T is the threshold used to reduce the effect of the outliers, and SSD is the **sum square difference** [78].

However, in our case, since our algorithm works with superpixel instead of the pixel, we sample each superpixel in nine different points as a representation of whole area of the superpixel. The nine representative points are the centroid of superpixel as well as furthest away pixels in eight main directions: north, south, east, west, northeast, northwest, southeast, and southwest. Therefore, for each superpixel, we compute nine different projections on each of the neighboring views in our camera arrays system and compute the accumulate photo-consistency. This can be shown more accurately as the following equation:

Equation 14: $$\boldsymbol{d}_\Omega = \underset{\boldsymbol{d}}{\boldsymbol{argmin}}\left(\sum_{\boldsymbol{i}=\boldsymbol{1}}^{\boldsymbol{N}} \sum_{\boldsymbol{p}\in\Omega} \boldsymbol{TSSD}\left(\boldsymbol{p}, \boldsymbol{p_i}(\boldsymbol{d}, \overline{\boldsymbol{n}})\right)\right)$$

In the above equation, the d_Ω is the estimated depth for the superpixel Ω, p is the current representative point of the superpixel Ω, $p_i(d, \bar{n})$ is projection of p in the ith view induced by plane $(d, \bar{n})$.

3.2.1 Depth Refinement

Superpixels are small compact regions of image in which the depth value is expected to change smoothly. For this reason, each superpixel can be approximated as a plane where depth is changing linearly. Consequently, each superpixel is shown as plane equation $(d, \bar{n})$, where d is the depth value at the centroid of the superpixel and $\bar{n}$ is the normal vector perpendicular to the plane.

The goal is to refine the initial depth values obtained in the previous stage. To achieve that, the problem is formulated as an optimization algorithm to maximize the consistency among different views, while imposing smoothness constraints within each view. Our energy function has the following form:

Equation 15: $$\boldsymbol{E}(\boldsymbol{d}, \overline{\boldsymbol{n}}) = \boldsymbol{E_c}(\boldsymbol{d}, \overline{\boldsymbol{n}})\boldsymbol{E_s}(\boldsymbol{d}, \overline{\boldsymbol{n}})$$

Here, the $E_c(d, \bar{n})$ and $E_s(d, \bar{n})$ are consistency and smoothness terms respectively and $d$and $\bar{n}$ are the plane parameters that model the superpixel. The refinement algorithm updates these parameters by iteratively performing two procedures: **plane propagation** and **plane refinement**.

In plane propagation, the superpixel centroid is interpolated in the plane equation of the neighboring superpixels. If the energy function using the new parameters is improved, then the current superpixel is updated with the parameters of its neighboring superpixel.

Relying only on the immediate neighbors, however, can easily halts the progress of algorithm within a local optimum [78]. To avoid that, we expand the range of the

propagation beyond the immediate neighbours by defining a propagation kernel as follows. First, we consider a square-like propagation window around the current superpixel. The size of the window is defined by the parameter $Size$. We check additional superpixels within the range of this window by sampling new ones. The frequency of candidate sampling is defined by parameter $Steps$. The $Steps$ and $Size$ together are called propagation parameters. This has been better shown in the Figure 3.6-left.

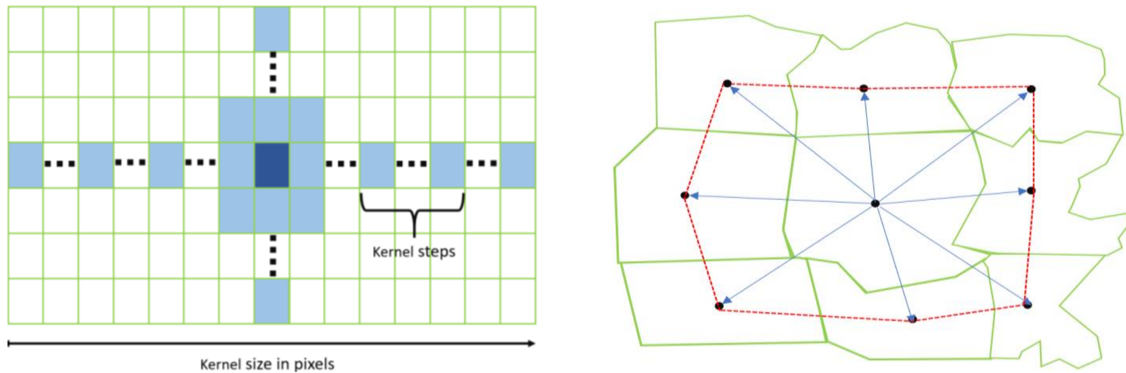

Figure 3.6. *Propagation kernel (left) and refinement procedure (right) [78].*

It has been experimentally shown large kernel size is more suitable for large textureless regions while scenes with clutters and lots of fine details tends work better with smaller kernel size [66]. To take a more balance approach, we decrease the $Size$ and $Steps$ parameters by the iteration number I in each round of iteration.

The propagation of initial plane information alone does not converge to optimal solution as long as new planes are not introduced to the current state of the map. This is achieved by performing plane refinement alongside plane propagation in which new slanted planes are added to the superpixel map. This is done by letting a new plane pass through the centroid of the reference superpixel and the centroid of its two adjacent neighbours as it is illustrated in Figure 3.6 (right). After that, a new normal vector is computed using the cross-product principle. Since the reference superpixel has eight adjacent pairs of neighbours, this process can be repeated eight times. If each newly generated plane improves the energy function of the current parameters, then the current normal vector is replaced as the new normal vector of the reference superpixel.

Plane propagation and refinement are consistently using smoothness and consistency as well as occlusion terms as their main procedure. In the rest of this section, we explain each mentioned term with more specific details.

The smoothness term is about enforcing spatial smoothness within each superpixel map. The basic assumption is that the neighbouring superpixels are expected to have

similar depth values rather than different ones. This assumption would become especially strong when two neighbouring superpixels have similar color. With this in mind, we tend to penalize the amount of difference on depth based on color similarity between the reference superpixel and its eight immediate neighbours. The neighbours with highest color similarity and lowest depth similarity tends to give the most penalty and the ones with highest color and depth similarity produces the least.

The penalty between a superpixel and its neighbor is computed by extrapolating the plane equation of the reference superpixel in the centroid of its neighbor to obtain the extrapolated depth. After that, the difference between extrapolated and current depth of the neighbouring superpixel is normalized (by using a Gaussian function) to produce the penalty of the current superpixel. Eventually, the final penalty is computed by summation of all neighbour's penalties.

Here we use the color similarity between two superpixels as a weight factor which multiplies by the difference in depth between two superpixels. The mathematical expression of the smoothness energy function is described as follow:

Equation 16: $$\boldsymbol{E_s(d,\bar{n}) = \frac{1}{\sum_{i=1}^{M} \omega(C_\Omega, C_i)} \sum_{i=1}^{M} \omega(C_\Omega, C_i) S_i\big(d_i, d_i(d,\bar{n})\big)}$$

$$\mathbf{S_i\big(d_i, d_i(d,\bar{n})\big) = e^{-\left(d_i - d_i(d,\bar{n})\right)^2/2\sigma^2}}$$
$$\omega(\mathbf{C_\Omega}, \mathbf{C_i}) = \mathbf{e^{-(C_i - C_\Omega)^2/2\alpha^2}}$$

In the Equation 1, the M is the number is of neighbors (eight), $\omega(C_\Omega, C_i)$ computes a normalized color similarity between reference superpixel Ω and its ith neighbour and d_i and $d_i(d,\bar{n})$ are the current and extrapolated depth of the neighbouring superpixel respectively. Consequently, $S_i\big(d_i, d_i(d,\bar{n})\big)$ computes the normalize depth difference between d_i and $d_i(d,\bar{n})$.

Consistency term measures the degree of visibility of each superpixel in other views. This can be done by computing the sum of photo-consistency values between the reference superpixel and its corresponding area in other views obtained by projecting reference superpixel in those views using the plane parameters $(d,\bar{n})$. If the current parameters are good enough, then we should expect to receive a relatively high photo-consistency since the depth of each pixel in all the views is the same.

This assumption, however, does not always hold true due to the presence of occlusions. To solve this issue, we break our consistency term into different terms: the visibility term and occlusion term. Occlusion term is responsible for measuring the correctness of the depth in the presence of occlusion. The general formula is as follows:

Equation 17: $\boldsymbol{E_c(d,\bar{n}) = \frac{1}{N}\sum_{i=1}^{N}\left(V_{\Omega}^{i}(d,\bar{n}) + O_{\Omega}^{i}(d,\bar{n})\right)}$

In the above equation, N is the number of the superpixel's representative pixels, $V_{\Omega}^{i}(d,\bar{n})$ is visibility term and $O_{\Omega}^{i}(d,\bar{n})$ is the occlusion term for the ith view. Here, if the depth candidate, which was used to do the projection between views, is the correct one, then both the color and the depth of the two regions should be the same. For this reason, the visibility term uses the color similarity as a weight factor to penalize the difference in depth between two corresponding superpixels. In other words, regions with similar depth and color would receive less penalty than regions with similar depth but different color. The formula for computing the color weight is as follows:

Equation 18: $\boldsymbol{S_i(d,\bar{n}) = \frac{1}{|\Omega|}\sum_{p\in\Omega} \omega\left(C_{\Omega}, C_{\Omega}^{i}(p_i(d,\bar{n}))\right)}$

In the Equation 18, $|\Omega|$ is superpixel area, C_{Ω} is the color of the reference super-pixel, C_{Ω}^{i} is the color of the corresponding super-pixel in the ith view, and $\omega(C_{\Omega}, C_i)$ is the color similarity function explained in Equation 16.

To compute the visibility term, assume all the representative pixels of the reference super-pixel p who has smaller depth and their correspond pixels in the ith view p_i to define the X as $\{P \mid D(p) \leq D(p_i)\}$. X is the set of all pixels which are closer to the ith camera. The visibility term is computed as follows:

Equation 19: $\boldsymbol{V_{\Omega}^{i}(d,\bar{n}) = S_i(d,\bar{n})\frac{1}{|X|}\sum_{p\in X} e^{-\left(D(p)-D(p^i)\right)^2/2\sigma^2}}$

For the occlusion term, we further define Yas $\{P \mid D(p) > D(p_i)\}$. Y is a set of all representative pixels which are closer to the reference camera and hence are either inconsistent or occluded. The occlusion term is defined as:

Equation 20: $\boldsymbol{O_{\Omega}^{i}(d,\bar{n}) = \begin{cases} \mu\left(1 - \min_{0\leq i\leq M} \omega(C_{\Omega}, C_i)\right), & Y \neq 0 \\ 0, & Y = 0 \end{cases}}$

where μ is a constant regularizer (typically set to 0.5).

3.2.2 Implementation methodology

Our depth estimation pipeline consists of several highly parallelizable functions which are serially connected to each other. Therefore, the natural implementation strategy for this problem is to execute each of these functions one at a time on a single GPU device.

In order to implement our functions on a GPU, it is better to first recognize the parallel pattern of each of them. In this pipeline, except the Update Cluster Center function,

which is used as part of the SLIC segmentation, the rest of our function possess an embarrassingly parallel pattern, which can easily be implemented on a GPU platform.

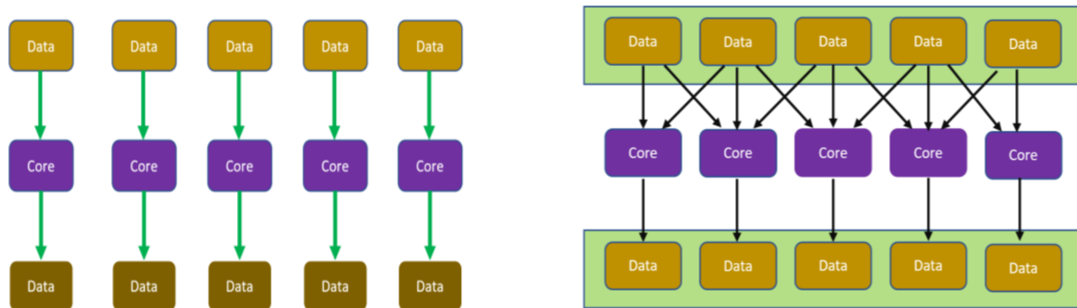

Figure 3.7. *Embarrassingly parallel patterns: map (left) and stencil (right) [2].*

Embarrassingly parallel patterns are types of a parallel workloads which can easily be broken down into independent parallel tasks. As a result, a serial program can be written for a single thread using global indexing parameters (get_global_id(0) and get_global_id(1)), and then, a private version of that program is generated for and executed by each thread (work-item) within the grid.

Embarrassingly parallel patterns can be further divided into two main groups: map and stencil patterns. The map pattern is the one-to-one connection between memory element and execution thread (core) of the parallel device. There is no memory overlapping between threads in either reading or writing phase (Figure 3.7-left). As a result, there is little to no point in using the shared (local) memory, as the nearby work-items do not share any data with each other. Convert_Color_Space and Find_Center_Association functions are examples of map pattern where one thread is dedicated for each pixel, and all the computation is done for each pixel independently.

Similar to map pattern, in the stencil pattern, computation for each work-item is perform independently. The difference, however, is in the memory read phase where neighboring work-items share memory (data) elements (Figure 3.7-right) with each other, providing an opportunity to use shared memory.

In addition to embarrassingly parallel patterns, our pipeline also uses reduction pattern as part of the Update_Cluster_Center function, where each cluster's information gets updated by aggregating information from the most recent update on member pixels. Reduction technique is applied for summation of information, for example color average, for each cluster, which is routinely done serially. The process is better shown on array in the Figure 3.8.

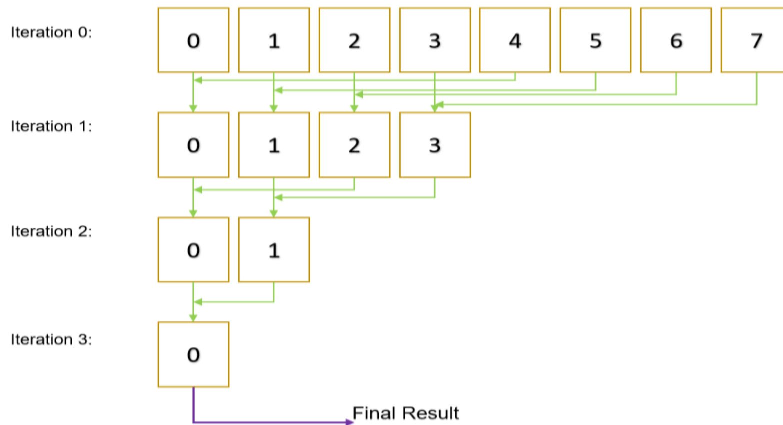

Figure 3.8. Reduction pattern

In reduction, the array is portioned into two equal parts. Each work-item being assigned to array elements (pixel in our case) of the first half is responsible for aggregating (summation in our case) and storing the information in its own respective element and its corresponding mirror element in the second half of the array. The work-items belonged to the second half are ignored. For example, if array has 100 elements, then the work-item index 0 is responsible for aggregating information of elements 0 and 50 (0 + 100/2) and work-item number 1 is responsible for elements 1 and 51 (1 + 100/2). Then this process is performed iteratively to break the size of the current array at each iteration. The process stops eventually when the size of the current array is equal to 1, and that single element holds all the aggregation (summation) of all the array.

3.3 View-interpolation Rendering

Our second application for harnessing the power of massively parallel GPU is a view-interpolation rendering algorithm based on plane-sweeping, where an entire view is generated from already existing images using correspondences between two stereo images. In the previous sections, we explained the plane-sweeping in detail. Here we would use the same principle to combine two existing stereo images to render a new image.

3.3.1 Plane-sweep Rendering

Given two stereo cameras, the goal is to move a virtual camera alongside the baseline and render a series of virtual views. This has better been depicted in Figure 3.9.

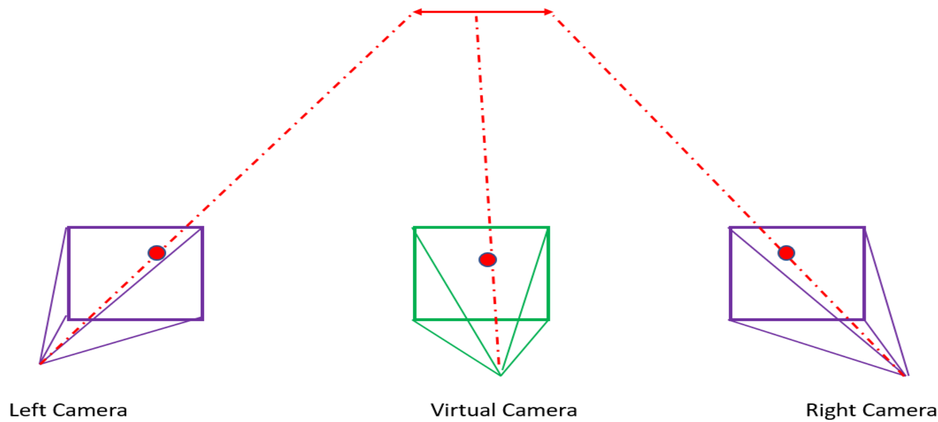

Figure 3.9. *A general view (camera) setup of the algorithm*

The general form of the algorithm has been depicted in Algorithm 1. It first visits each pixel p of virtual view one at a time and then it iterates through the whole range of the depth hypothesis D to perform four different steps on each (p, d) pair where d is the current depth hypothesis.

Algorithm 1: *Serial Plane-sweep Rendering*

Input: *left and right stereo images*
Output: *virtual view I*

1 ***for*** *each p ∈I:*
2 *for each d ∈D:*
3 ***step 1:*** *Project p from 2-D virtual image to 3-D space, using current d*
4 ***step 2:*** *Apply rotation and translation to change the coordinate system to each real camera.*
5 ***step 3:*** *Project the new 3-D points back into their corresponding 2-D camera plane*
6 ***step 4:*** *use photo-consistency to update the best match so far*
7 *end*
8 *update I by averaging the color of best matches for p in left and right images*
9 *end*

Algorithm 1: the general overview of the algorithm

In the first step, the pixel p(x, y) *is projected* from a 2D plane to point *P(X, Y, Z)* on 3D plane using the current depth hypothesis *d*. we implement the following lines of codes to do that:

$$X = \frac{(x - c_x)}{f} \times d, \; Y = \frac{(y - c_y)}{f} \times d, \quad Z = d$$

The second step transfers the origin of the coordinate system from the virtual camera to the left and right cameras using the rotation (R) and translation (T) vectors. Here, T_r and T_l are translation vector's elements alongside X-axis in right and left directions respectively. Since we are using horizontal images, we do not have any translation in *y* or *z* directions. Moreover, since the stereo images are already rectified, the rotation matrix

is an identity matrix with no effect on coordinate system. We implement the following equations to transform the coordinate system:

$$X_r = X + T_r, \qquad X_l = X + T_l$$

In the third step, the newly generated 3-D coordinate is projected back to its corresponding left and right camera planes.

$$x_{rproj} = Z \times \frac{X_r}{f} \qquad x_{lproj} = Z \times \frac{X_l}{f}$$
$$y_{rproj} = Z \times \frac{Y_r}{f} \qquad y_{lproj} = Z \times \frac{Y_r}{f}$$

In the final step, the photo-consistency term is computed to pick the best matching d for the pixel p. We use absolute some difference (L1 norm) as the cost function. At the end, the color of the best projected pixels (on left and right images), are averaged to make the color of the pixel p in the virtual image.

3.3.2 GPU Implementation

We developed two GPU-based implementations for this algorithm. The first one is a simple embarrassingly parallel (stencil pattern) implementation where each pixel of the virtual image is considered as a parallel thread [2]. In this way, a general code using thread indexing of OpenCL is written and then a private version of that code is generated for every thread within the grid. As a result, the first for loop, which iterates over all the pixels of the virtual camera, is omitted. Algorithm 2 shows the pseudo code of our first GPU implementation.

Since GPU's global memory, the main memory of GPU, is very slow, one of the best ways to reduce the running time of our GPU algorithm is to decrease the number of accesses to the global memory. This can be achieved by using shared memory to cache the data once they have been accessed. In this way, the same data can be accessed multiple times on fast on-chip shared memory without accessing the global memory multiple times. For our second implementation, we utilize shared memory to improve the data efficiency of our code.

Algorithm 2: *GPU Plane-sweep Rendering*

Input: *left and right stereo images*
Output: *virtual view I*

```
1   x = get_global_id(0), y = get_global_id(1)
2       for each d ∈ D:
3           step 1:   Project (x, y) from 2-D space to (X, Y, Z) in 3-D space, using current d
```

4   ***step 2:*** *Apply rotation and translation to change the coordinate system to each real camera.*
5   ***step 3:*** *Project the new 3-D points back into their corresponding 2-D camera plane*
6   ***step 4:*** *use photo-consistency to update the best match so far*
7   ***end***
8   *update **I** by averaging the color of best matches for p in left and right images*
9   *end*

Algorithm 2: Naïve GPU implementation

In our algorithm, once the equivalent location of virtual pixel p is found on left(P_{lproj}) and right (P_{rproj}) images, using the current d, by current work-item (x, y) , the photo-consistency cost is computed between two projections. The computation is done by placing a local support window around P_{rproj} and $P_{lproj}$and compute the Sum of Absolute Difference (SAD) between those two local regions. The problem with this approach, however, is that all the neighboring pixels will be accessed several times by the neighboring work-items, i.e.,$(x-1, y)$, during executions. As a result, the number of the global memory access would exponentially increase throughout the grid.

The high overlapping data access pattern of Algorithm 1 has been better illustrated in Figure 3.10. As it is readily apparent, the three different work-items (red, green, yellow) from the same work-group (blue region) and their corresponding local windows (dashed lines) are sharing a significant area of the image with each other. The goal of our second implementation is to cache these areas on shared memory for efficient use.

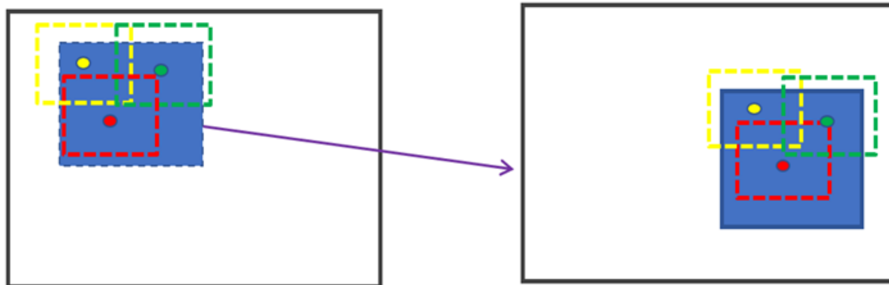

Figure 3.10. High data access overlapping between red, green, and yellow work-items in the naïve GPU kernel. All three work-items belong to the same group (blue box)

The improved memory pattern access, for our second implementation, has been shown in Figure 3.11. The blue region (separated with solid blue line) is the location of an arbitrary 6x4 work-group on one of the input images, and each work-item of this group is responsible of loading a portion of that image from the global memory to the shared memory. In this figure, all pixels of a color group (i.e., all the red pixels) are loaded to the shared memory all at once using their corresponding work-item in the group.

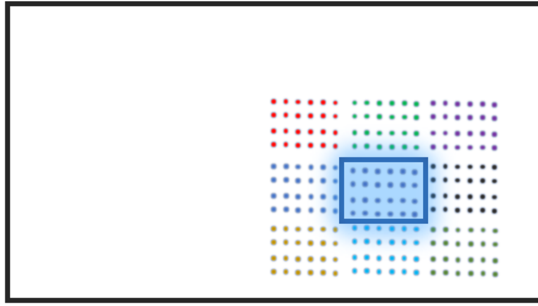

Figure 3.11. *Memory access pattern of the algorithm 2.*

Since the data overlapping comes at the photo-consistency computation, the only part of the code that changes from Algorithm 2 is the function that computes the photo-consistency. The rest of the algorithm remain the same. Therefore, the only thing we need to do is to replace the function call of old photo-consistency function with the new one.

4. EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Experimental Setting

All our experiments are performed on a desktop computer equipped with a Nvidia GTX-1080 GPU and Intel Core i7 CPU. We use OpenCL version 1.2 to implement our algorithms on GPU and OpenMP version 5.0 for multi-core CPU.

As previously explained, our multi-view depth estimation algorithm consists of three main stages: super-pixel segmentation, depth initialization, and depth refinement. In this section, we demonstrate the experimental results of each stage, try to provide a deeper insight into the performance of our methods, and finally report the speed up results.

4.2 Multi-view Depth Estimation

4.2.1 Dataset Specifications

We use the Max Plank Institute's light field dataset available at [85]. The dataset specification is described in Table 1. Dataset specification for MVS algorithm.

Table 1. Dataset specification for MVS algorithm

Name	No views	Resolution	Disparity Levels	Baseline Ratio(y/x)
Bar	3x5	1920x1080	45	0.625
Biergarten	3x3	1920x1080	30	1.03590

Here, baseline ratio is the ratio of y and x elements of two diagonal camera's baseline in the camera array system.

4.2.2 Superpixel Segmentation Results

As described in the section 3.1.1, we implement the simple linear iterative clustering (SLIC) as our method of choice for superpixel segmentation. Here, we have implemented the Oxford's gSLICr library, which was originally developed in CUDA [71], in OpenCL.

Our implementation possesses several parameters that determines the execution flow of the algorithm. Based on this, we have designed and executed a number of experiments to demonstrate the whole range of SLIC's behaviour under different parameter settings. Our experiments are described in Table 1.

Table 2. *SLIC experiments with different set of parameters*

No	Super-pixel Size	SLIC Color Weight	Number of Iterations	Enforce Connectivity
1	8	0.6	5	False
2	8	0.6	5	True
3	16	0.6	5	False

In this table, superpixel size determines the initial size of the superpixel which iteratively get updated throughout the execution. SLIC is an iterative process, and it approximately takes four or five iterations for the algorithms to converge to a stable result. So, we consider five as the default number of iterations for all our experiments. SLIC's distance function, which measures the closeness of a pixel to the center of cluster, has two terms: spatial term and color term. The Color Weight parameter is a coefficient which emphasis the degree of importance of color term. Finally, in experiment 3, we enable the enforce connectivity to remove small isolated super-pixels to generate a cleaner result. We show our experiments output in Figure 4.1, Figure 4.2, and Figure 4.3.

Figure 4.1. *The default SLIC output with superpixel size 8*

The reason for such result is that super-pixels are defined to be spatially restricted in a small neighborhood . The regular squares are, in fact, the default position of all super-pixels before any update happening. Once the iterative update starts, each super-pixel falls within a race with its eight immediate neighbours for taking more pixels. This competition especially is high near the bordering areas where color similarity plays a major role. However, as we move further and further away from the border area towards the center, the role of spatial term becomes increasingly more important. With this in mind, once the color weight is set to zero, super-pixels no longer have any leverage against each other and as a result they all become regular square with the same size.

Figure 4.2 *Effect of the enforce connectivity.*

Figure 4.3 *segmentation with super-pixel size 16*

4.2.3 Initialization and Refinement Results

Our depth estimation parameters are listed in Table 3. We use superpixel size of 8 and 16 for our simulations. We also generate the output without enforce connectivity and show the results.

Table 3. *depth estimation parameters for bar dataset*

No	Name	Value	Description
1	Kernel_Size	1080	range of propagation kernel around superpixel
2	Kernel_Steps	13	sampling step within a propagation kernel
3	alpha	6	normalization coefficient for color distance
4	gamma	2	spatial distance normalization coefficient in the smoothness and consistency coset function
5	no_iteration	5	number of the times the propagation function is called to produce stable results

The result of our depth initialization part has been depicted in Figure 4.4 Initial depth estimation with super-pixel size 8. Since we are not using any optimization, the result is very noisy.

Figure 4.4 *Initial depth estimation with super-pixel size 8*

The result of depth refinement of our implementation as well as the result of reference paper are shown in the Figure 4.5 and Figure 4.6 respectively.

Figure 4.5. *Our refined disparity map. The super-pixel size is 8*

Figure 4.6. *Reference paper output using superpixel size 8*

4.2.4 Quality Analysis

As it can be seen, the reference result is much smoother and less noisy compared to our own results. We tried to reduce the gap by utilizing larger superpixel size to reduce the amount of the noise. The previous experiments were done by using superpixel size 8. For the new experiment we use superpixel size 16. The result has been shown in the Figure 4.7.

Figure 4.7. *The improved result on Bar dataset using superpixel size 16*

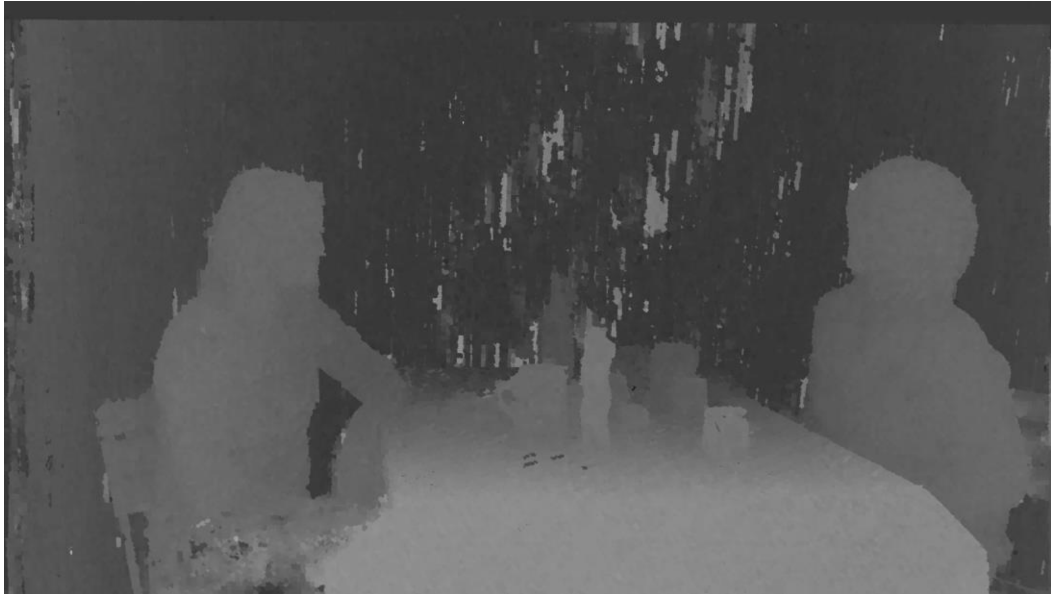

Figure 4.8. Algorithm's result on Biergarten dataset using superpixel 8

For the Biergarten dataset, however, with superpixel pixel size 8, we can still have a decent result (Figure 4.8). The reported experiments were all done using three different platforms: single-core CPU, multi-core CPU, and GPU. This section reports the execution time and speed up gain of all our methods under different platforms. Moreover, Table 4 and Table 5, show the speed up gain of different depth initialization and depth refinement stages for Bar and Biergarten dataset respectively.

4.2.5 Performance Evaluation

We evaluate the running time performance of our implementation on Bar and Biergarten dataset

Table 4 *different platforms's performance on Bar dataset for superpixel size 16*

platform name	***initialization time (msec)***	***Refinement time (msec)***	***Initialization Speed- up***	***Refinement Speed- up***
single-core CPU	238905	954566	1.0x	1.0x
multi-core CPU	182391	337808	1.3x	2.8x
GPU	1415	605	168.83x	1577.8x

Table 5 *different platforms' performance on Biergarten dataset for superpixel size 16*

platform name	***initialization time (msec)***	***Refinement time (msec)***	***Initialization Speed Up***	***Refinement Speed Up***
single-core CPU	106266	366279	1.0x	1.0x
multi-core CPU	42398	131901	2.5x	2.77x
GPU	421	167	252.4x	2193.2x

4.3 Plane-sweep Rendering

4.3.1 Stereo Dataset Specification

We use the 2014 Middlebury stereo dataset to develop our algorithm. The dataset can be found and downloaded in the Middlebury website at [84]. Each dataset contains several files which depict and describe two views captured under different illuminations and exposures. These files include calibration information of each camera, different versions of rectified stereo pairs of the same scene, and the disparity ground truth of left and right image. The dataset specification is described in the Table 6.

Table 6. *parameters for stereo datasets*

Name	***Parameter Description***
cam0, 1	calibration matrices for left/right camera. The format is *[f 0 c_x ; 0 f c_y ; 0 0 1]* where f is the focal length (in pixel) and c_x and c_y are the deviations from the principle point
d_{offs}	x-difference of principle point or $c_{x1} - c_{x2}$
baseline	the distance between two camera's principal point (in mm)
width, height	image resolution
ndisp	number of the disparities.
isint	whether ground truth disparities have integer precision
vmin, vmax	a tight bound on minimum and maximum amount of disparity
dyavg,dy-max	average and maximum absolute y-disparities, providing an indication of the calibration error present in the imperfect datasets

Since our algorithm works with depth instead of disparity, we, first, need to convert all the disparities to dept h using the following formula:

$$Z = baseline \times f/(d + d_{offs})$$

As a result, d_{min} (minimum depth) and d_{max} (maximum depth) can be calculated as follows:

$$d_{min} = baseline \times f/(v_{max} + d_{offs})$$
$$d_{max} = baseline \times f/(d_{min} + d_{offs})$$

We also convert the measurement of the focal point (f) from millimetre to meter to use more accurate floating-point operation.

4.3.2 Execution Parameters

Our implementation works with few parameters which determines the flow of the execution. The parameters are listed in Table 7.

Table 7. *execution parameters of our implementation*

Name	***Description***
wSize	Size of the local supporting window for plane sweeping and view rendering
dScale	Scaling coefficient for input images (default value is 1)
delta	The distance of the principle point of virtual camera from the left camera

These parameters are input to the rendering function. Since we want to generate a series of images moving from left image to the right image, we call the rendering function in a loop with increasing delta value.

4.3.3 Plane-sweep Results

Our implementation methodology is to first implement plane-sweeping on GPU and then develop a separate rendering function base on that GPU code. Figure 4.9 demonstrate the output of our plane-sweeping algorithms on chairs and piano datasets.

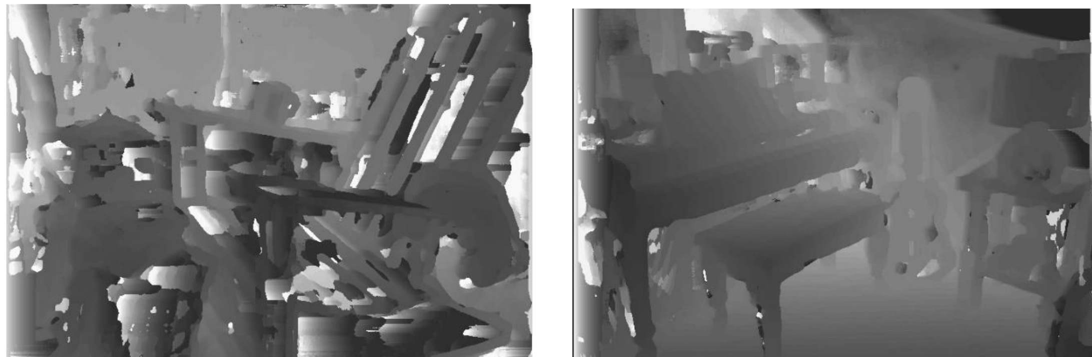

Figure 4.9. *Plane-sweeping depth map for Chair dataset (left) and Piano dataset (right).*

4.3.4 Rendering Results

The result of our viewport rendering is demonstrated in Figure 4.10 and Figure 4.11 for chair and piano dataset respectively. A series of images can be generated consecutively by moving the virtual camera from the left image toward the right image (increasing the delta parameter).

Figure 4.10. *Rendering view for living room dataset with delta equals to 0.55*

Figure 4.11. *Rendered view for piano dataset with delta equal to 0.55*

As it can be seen, the quality of the generated images is much better for the Piano dataset than the chair dataset. The reason is that plane-sweeping is a simple local window-based algorithm without any global optimization constraints. As a result, it tends to work poorly in the presence of fine details such as those in the chair dataset.

4.3.5 Execution Time

The execution time of our experiments is reported in this section. Table 8 shows the performance of four different implementations of rendering algorithms: single-core implementation, multi-core CPU implementation, simple GPU implementation, and advanced GPU implementation. In addition to, Table 9 shows the execution time of the plane-sweeping method on the same three platforms.

Table 8. *execution time and speed up for Living room dataset*

Platform Name	***Windows size (pixels)***	***Execution time (msec)***	***speed up***
Single-core CPU	13	123325	1.0x
Multi-core CPU	13	29069	4.24x
Naive GPU	13	154	800.8x
Shared memory GPU	13	60	2055.4x

Table 9. *execution time and speed up for piano dataset*

Platform Name	***Windows Size (pixels)***	***Execution time (msec)***	***speed up***
single-core CPU	13	126512	1.0x
multi-core CPU	13	28053	4.5x
GPU	13	144	878.5x
Shared memory GPU	13	57	2219x

5. CONCLUSIONS

In this thesis, we examine the massively parallel power of GPU on computer vision algorithms for depth estimation and image-based rendering applications. We implemented our algorithms on three different platforms (single-core and multi-core CPU as well as GPU) and chose single-core implementation as anchor for calculating the speed up. We have shown for both applications, which hold a significant amount of parallelism, a GPU-based implementation would achieve a very high speed up compared to a standard CPU implementation.

The first application, a multi-view depth estimation algorithm proposed in [78], is a three-stage pipeline process. First, it partitions the input images into compact homogenous regions called superpixels. Then, it initializes the depth for each superpixel using a plane-sweeping-based strategy. Finally, it refines depth value using an iterative optimization technique to fit a proper plane surface to each superpixel. In this thesis, we have shown that our GPU implementation has significantly improved compared to standard sequential and multi-core CPU implementations.

For the second application, the goal is to render a novel view from two calibrated stereo images. To this end, we have chosen a less complex but computationally efficient approach based on a plane-sweeping algorithm. In our algorithm, we avoid producing a plane-sweeping volume due to its large memory size and the fact that the off-chip global memory of the GPU is very slow. Instead, we kept all the computations on the GPU's register memory and update the best candidate for the current virtual pixel in real-time. As a result, our algorithm has gained a high amount of speed-up. Taking such approach, however, could come at the cost of dropping the quality of generated view in case of fine details and sharp edges in stereo images due to the lack of a cost-aggregation process.

REFERENCES

[1] Pacheco P, Malensek M. An introduction to parallel programming. Morgan Kaufmann; 2021 Aug 27.

[2] McCool M, Reinders J, Robison A. Structured parallel programming: patterns for efficient computation. Elsevier; 2012 Jul 9.

[3] Kirk D, Wen-Mei WH. Programming massively parallel processors: a hands-on approach. Morgan kaufmann; 2016 Nov 24.

[4] Sutter, H., & Larus, J. (2005). Software and the Concurrency Revolution. ACM Queue, 3(7), 5462.

[5] Tunc C, Kumbhare N, Akoglu A, Hariri S, Machovec D, Siegel HJ. Value of service based task scheduling for cloud computing systems. In 2016 International Conference on Cloud and Autonomic Computing (ICCAC) 2016 Sep 12 (pp. 1-11). IEEE.

[6] Khemka B, Machovec D, Blandin C, Siegel HJ, Hariri S, Louri A, Tunc C, Fargo F, Maciejewski AA. Resource management in heterogeneous parallel computing environments with soft and hard deadlines. In11th Metaheuristics International Conference (MIC 2015) 2015 Jun (p. 10).

[7] Kaeli DR, Mistry P, Schaa D, Zhang DP. Heterogeneous computing with OpenCL 2.0. Morgan Kaufmann; 2015 Jun 18.

[8] S. Likun Xi et al., “SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning Workloads,” arXiv e-prints, p. arXiv:1912.04481, Dec 2019.

[9] Nurvitadhi E, Venkatesh G, Sim J, Marr D, Huang R, Ong Gee Hock J, Liew YT, Srivatsan K, Moss D, Subhaschandra S, Boudoukh G. Can fpgas beat gpus in accelerating next-generation deep neural networks?. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays 2017 Feb 22 (pp. 5-14).

[10] Muslim FB, Ma L, Roozmeh M, Lavagno L. Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis. IEEE Access. 2017 Feb 20;5:2747-62.

[11] Mario Vestias and Horácio Neto. 2014. Trends of CPU, GPU and FPGA for high-performance computing. International Conference on Field Programmable Logic and Applications (FPL’14). IEEE, 1–6.

[12] Vestias M, Neto H. Trends of CPU, GPU and FPGA for high-performance computing. In2014 24th International Conference on Field Programmable Logic and Applications (FPL) 2014 Sep 2 (pp. 1-6). IEEE.

[13] Eskandari N, Tarafdar N, Ly-Ma D, Chow P. A modular heterogeneous stack for deploying fpgas and cpus in the data center. InProceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays 2019 Feb 20 (pp. 262-271).

[14] Intel corporation. Xeon Processor [Internet]. [place unknown]; Intel corporation. 2022. Available at: https://www.intel.com/content/www/us/en/products/details/processors/xeon/w.html

[15] Hennessy JL, Patterson DA. Computer architecture: a quantitative approach. Elsevier; 2011 Oct 7.

[16] Intel Corporation. Intel Product Information. [unknown place]:[unknown publisher]. [reviewed 2021 Oct 25; cited 2021 March 2021]. Available at: https://www.intel.com/content/www/us/en/developer/articles/news/raw-compute-power-of-new-intel-core-i9-processor-based-systems-enables-extreme-mega-tasking.html#:~:text=The%20new%20Intel%C2%AE%20Core,cores%2C%20performing%20at%20124.5%20petaflops.

[17] OpenMP ARB. OpenMP guide. [unknown place]: OpenMP ARB. 2021. Available at: https://www.openmp.org/resources/refguides/

[18] Mansour M, Davidson P, Stepanov O, Piché R. Relative importance of binocular disparity and motion parallax for depth estimation: a computer vision approach. Remote Sensing. 2019 Jan;11(17):1990.

[19] UC Brekely. CPU Performance Information [Internet]. [place unknown]:[university of California].2022. Available at: https://setiathome.berkeley.edu/cpu_list.php

[20] Nvidia Corporation. NVIDIA Fermi Architecture Whitepaper [Internet]. [place unknown]: Nvidia. Available at: NVIDIA Fermi Compute Architecture White Paper | Dell

[21] Sanz PR, Mezcua BR, Pena JM. Depth estimation-an introduction. IntechOpen; 2012 Jul 11.

[22] Khan F, Salahuddin S, Javidnia H. Deep learning-based monocular depth estimation methods—A state-of-the-art review. Sensors. 2020 Jan;20(8):2272.

[23] Jain P, Ralston JP. Direct determination of astronomical distances and proper motions by interferometric parallax. Astronomy & Astrophysics. 2008 Jun 1;484(3):887-95.

[24] Scharstein D, Szeliski R. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. International journal of computer vision. 2002 Apr;47(1):7-42.

[25] Feng D, Haase-Schütz C, Rosenbaum L, Hertlein H, Glaeser C, Timm F, Wiesbeck W, Dietmayer K. Deep multi-modal object detection and semantic segmentation for autonomous driving: Datasets, methods, and challenges. IEEE Transactions on Intelligent Transportation Systems. 2020 Feb 17;22(3):1341-60.

[26] Ummenhofer B, Zhou H, Uhrig J, Mayer N, Ilg E, Dosovitskiy A, Brox T. Demon: Depth and motion network for learning monocular stereo. InProceedings of the IEEE Conference on Computer Vision and Pattern Recognition 2017 (pp. 5038-5047).

[27] Wang K, Shen S. MVDepthNet: Real-time multiview depth estimation neural network. In 2018 International conference on 3d vision (3DV) 2018 Sep 5 (pp. 248-257). IEEE.

[28] Flynn J, Neulander I, Philbin J, Snavely N. Deepstereo: Learning to predict new views from the world's imagery. InProceedings of the IEEE conference on computer vision and pattern recognition 2016 (pp. 5515-5524).

[29] Huang PH, Matzen K, Kopf J, Ahuja N, Huang JB. Deepmvs: Learning multi-view stereopsis. InProceedings of the IEEE Conference on Computer Vision and Pattern Recognition 2018 (pp. 2821-2830).

[30] Bleyer M, Rhemann C, Rother C. PatchMatch Stereo-Stereo Matching with Slanted Support Windows. InBmvc 2011 Aug 29 (Vol. 11, pp. 1-11).

[31] Zhan Y, Gu Y, Huang K, Zhang C, Hu K. Accurate image-guided stereo matching with efficient matching cost and disparity refinement. IEEE Transactions on Circuits and Systems for Video Technology. 2015 Aug 26;26(9):1632-45.

[32] Kitagawa M, Shimizu I, Sara R. High accuracy local stereo matching using DoG scale map. In2017 Fifteenth IAPR International Conference on Machine Vision Applications (MVA) 2017 May 8 (pp. 258-261). IEEE.

[33] Felzenszwalb PF, Huttenlocher DP. Efficient belief propagation for early vision. International journal of computer vision. 2006 Oct;70(1):41-54.

[34] Tippetts B, Lee DJ, Lillywhite K, Archibald J. Review of stereo vision algorithms and their suitability for resource-limited systems. Journal of Real-Time Image Processing. 2016 Jan;11(1):5-25.

[35] Murphy K, Weiss Y, Jordan MI. Loopy belief propagation for approximate inference: An empirical study. arXiv preprint arXiv:1301.6725. 2013 Jan 23.

[36] Kolmogorov V, Zabih R. Multi-camera scene reconstruction via graph cuts. In European conference on computer vision 2002 May 28 (pp. 82-96). Springer, Berlin, Heidelberg.

[37] Zhang G, Jia J, Wong TT, Bao H. Consistent depth maps recovery from a video sequence. IEEE Transactions on pattern analysis and machine intelligence. 2009 Mar 6;31(6):974-88.

[38] Ulusoy AO, Geiger A, Black MJ. Towards probabilistic volumetric reconstruction using ray potentials. In 2015 International Conference on 3D Vision 2015 Oct 19 (pp. 10-18). IEEE.

[39] Besse FO. PatchMatch Belief Propagation for Correspondence Field Estimation and Its Applications (Doctoral dissertation, UCL (University College London)).

[40] Scharstein D, Taniai T, Sinha SN. Semi-global stereo matching with surface orientation priors. In2017 International Conference on 3D Vision (3DV) 2017 Oct 10 (pp. 215-224). IEEE.A Resource-Efficient Pipelined Architecture for Real-Time Semi-Global Stereo Matching Semi-Global Stereo Matching Algorithm Based on Minimum Spanning Tree

[41] Lu Z, Wang J, Li Z, Chen S, Wu F. A resource-efficient pipelined architecture for real-time semi-global stereo matching. IEEE Transactions on Circuits and Systems for Video Technology. 2021 Feb 26.

[42] Dinh VQ, Pham CC, Jeon JW. Robust adaptive normalized cross-correlation for stereo matching cost computation. IEEE Transactions on Circuits and Systems for Video Technology. 2016 Mar 8;27(7):1421-34.

[43] Zbontar J, LeCun Y. Computing the stereo matching cost with a convolutional neural network. InProceedings of the IEEE conference on computer vision and pattern recognition 2015 (pp. 1592-1599).

[44] Zbontar J, LeCun Y. Stereo matching by training a convolutional neural network to compare image patches. J. Mach. Learn. Res.. 2016 Jan 1;17(1):2287-318.

[45] Szeliski R. Computer vision: algorithms and applications. Springer Science & Business Media; 2010 Sep 30.

[46] Zhang K, Fang Y, Min D, Sun L, Yang S, Yan S, Tian Q. Cross-scale cost aggregation for stereo matching. InProceedings of the IEEE Conference on Computer Vision and Pattern Recognition 2014 (pp. 1590-1597).

[47] Tombari F, Mattoccia S, Di Stefano L, Addimanda E. Classification and evaluation of cost aggregation methods for stereo correspondence. In2008 IEEE Conference on Computer Vision and Pattern Recognition 2008 Jun 23 (pp. 1-8). IEEE.

[48] Yang Q, Wang L, Yang R, Stewénius H, Nistér D. Stereo matching with color-weighted correlation, hierarchical belief propagation, and occlusion handling. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2008 Apr 18;31(3):492-504.

[49] Crow FC. Summed-area tables for texture mapping. InProceedings of the 11th annual conference on Computer graphics and interactive techniques 1984 Jan 1 (pp. 207-212).

[50] Seitz SM, Curless B, Diebel J, Scharstein D, Szeliski R. A comparison and evaluation of multi-view stereo reconstruction algorithms. In 2006 IEEE computer society conference on computer vision and pattern recognition (CVPR'06) 2006 Jun 17 (Vol. 1, pp. 519-528). IEEE.

[51] Hernández C, Furukawa Y. Multi-View Stereo: A Tutorial. Comput. Graph. Vision2. 2013; 9:1-48.

[52] A Review of Image-based Rendering Techniques, Heung-Yeung Shum and Sing Bing Kang, Microsoft research

[53] Chang Y, Guo-Ping WA. A review on image-based rendering. Virtual Reality & Intelligent Hardware. 2019 Feb 1;1(1):39-54.

[54] Sun W, Xu L, Au OC, Chui SH, Kwok CW. An overview of free view-point depth-image-based rendering (DIBR). InAPSIPA Annual Summit and Conference 2010 Dec 14 (pp. 1023-1030).

[55] Adelson EH, Bergen JR. The plenoptic function and the elements of early vision. Vision and Modeling Group, Media Laboratory, Massachusetts Institute of Technology; 1991 Oct.

[56] Wong TT, Heng PA, Or SH, Ng WY. Image-based rendering with controllable illumination. InEurographics workshop on Rendering Techniques 1997 Jun 16 (pp. 13-22). Springer, Vienna.

[57] Levoy M, Hanrahan P. Light field rendering. InProceedings of the 23rd annual conference on Computer graphics and interactive techniques 1996 Aug 1 (pp. 31-42).

[58] Gortler SJ, Grzeszczuk R, Szeliski R, Cohen MF. The lumigraph. InProceedings of the 23rd annual conference on Computer graphics and interactive techniques 1996 Aug 1 (pp. 43-54).

[59] Shum HY, He LW. Rendering with concentric mosaics. InProceedings of the 26th annual conference on Computer graphics and interactive techniques 1999 Jul 1 (pp. 299-306).

[60] Chen SE, Williams L. View interpolation for image synthesis. InProceedings of the 20th annual conference on Computer graphics and interactive techniques 1993 Sep 1 (pp. 279-288).

[61] Seitz SM, Dyer CR. View morphing. InProceedings of the 23rd annual conference on Computer graphics and interactive techniques 1996 Aug 1 (pp. 21-30).

[62] Shade J, Gortler S, He LW, Szeliski R. Layered depth images. InProceedings of the 25th annual conference on Computer graphics and interactive techniques 1998 Jul 24 (pp. 231-242).

[63] Narayanan PJ, Penta SK, Reddy S. Depth+ Texture Representation for Image Based Rendering. InICVGIP 2004 Dec (pp. 113-118).

[64] McMillan Jr L. An image-based approach to three-dimensional computer graphics. The University of North Carolina at Chapel Hill; 1997.

[65] Geys I, Koninckx TP, Van Gool L. Fast interpolated cameras by combining a GPU based plane sweep with a max-flow regularisation algorithm. InProceedings. 2nd International Symposium on 3D Data Processing, Visualization and Transmission, 2004. 3DPVT 2004. 2004 Sep 9 (pp. 534-541). IEEE.

[66] Goorts P, Maesen S, Dumont M, Rogmans S, Bekaert P. Optimization of free viewpoint interpolation by applying adaptive depth plane distributions in plane sweeping a histogram-based approach to a non-uniform plane distribution. In2013 International Conference on Signal Processing and Multimedia Applications (SIGMAP) 2013 Jul 29 (pp. 7-15). IEEE.

[67] Mori T, Takahashi K, Fujii T. Real-Time Free-Viewpoint Image Synthesis System Using Time Varying Projection. ITE Transactions on Media Technology and Applications. 2014;2(4):370-7.

[68] Wu G, Masia B, Jarabo A, Zhang Y, Wang L, Dai Q, Chai T, Liu Y. Light field image processing: An overview. IEEE Journal of Selected Topics in Signal Processing. 2017 Aug 30;11(7):926-54.

[69] Ng R, Levoy M, Brédif M, Duval G, Horowitz M, Hanrahan P. Light field photography with a hand-held plenoptic camera (Doctoral dissertation, Stanford University).

[70] Perwass C, Wietzke L. Single lens 3D-camera with extended depth-of-field. In-Human vision and electronic imaging XVII 2012 Feb 17 (Vol. 8291, p. 829108). International Society for Optics and Photonics.

[71] Wilburn B, Joshi N, Vaish V, Talvala EV, Antunez E, Barth A, Adams A, Horowitz M, Levoy M. High performance imaging using large camera arrays. InACM SIGGRAPH 2005 Papers 2005 Jul 1 (pp. 765-776).

[72] camera arrays", ACM Trans. Graph, vol. 24, no. 3, pp. 765-776, 2005.

[73] Liu Y, Dai Q, Xu W. A real time interactive dynamic light field transmission system. In2006 IEEE International Conference on Multimedia and Expo 2006 Jul 9 (pp. 2173-2176). IEEE.

[74] Dąbała Ł, Ziegler M, Didyk P, Zilly F, Keinert J, Myszkowski K, Seidel HP, Rokita P, Ritschel T. Efficient Multi-image Correspondences for On-line Light Field Video Processing. InComputer Graphics Forum 2016 Oct (Vol. 35, No. 7, pp. 401-410).

[75] Sabater N, Boisson G, Vandame B, Kerbiriou P, Babon F, Hog M, Gendrot R, Langlois T, Bureller O, Schubert A, Allie V. Dataset and pipeline for multi-view light-field video. InProceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops 2017 (pp. 30-40).

[76] Collins RT. A space-sweep approach to true multi-image matching. InProceedings CVPR IEEE Computer Society Conference on Computer Vision and Pattern Recognition 1996 Jun 18 (pp. 358-363). IEEE.

[77] Gallup D, Frahm JM, Mordohai P, Yang Q, Pollefeys M. Real-time plane-sweeping stereo with multiple sweeping directions. In2007 IEEE Conference on Computer Vision and Pattern Recognition 2007 Jun 17 (pp. 1-8). IEEE.

[78] Chuchvara A, Barsi A, Gotchev A. Fast and accurate depth estimation from sparse light fields. IEEE Transactions on Image Processing. 2019 Dec 17;29:2492-506.

[79] Achanta R, Shaji A, Smith K, Lucchi A, Fua P, Süsstrunk S. SLIC superpixels compared to state-of-the-art superpixel methods. IEEE transactions on pattern analysis and machine intelligence. 2012 May 29;34(11):2274-82.

[80] Qin F, Guo J, Lang F. Superpixel segmentation for polarimetric SAR imagery using local iterative clustering. IEEE Geoscience and Remote Sensing Letters. 2014 Jun 24;12(1):13-7.

[81] Yang F, Sun Q, Jin H, Zhou Z. Superpixel segmentation with fully convolutional networks. InProceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition 2020 (pp. 13964-13973).

[82] Ren CY, Prisacariu VA, Reid ID. gSLICr: SLIC superpixels at over 250Hz. arXiv preprint arXiv:1509.04232. 2015 Sep 14.

[83] Oxford University. gSLICr source code [Internet]. [place unknown]: Robotic Laboratory.2022. Available at: GitHub - carlren/gSLICr: gSLICr: Real-time superpixel segmentation

[84] Middlebury College. Middlebury Stereo Dataset [Internet]. Vermont: [publisher unknown].2021. Available at: vision.middlebury.edu/stereo/data

[85] Max Plank Institute. Efficient Multi-image Correspondences for On-line Light Field Video Processing [Internet]. Germany: [publisher unknown].2018. Available at: https://resources.mpi-inf.mpg.de/LightFieldVideo/Dataset.html

[86] Savaresa S, Week 1: Camera Model. CS231A: Computer Vision From 3D Reconstruction to Recogntion, Stanford University; lecture given winter 2022, Available at: https://web.stanford.edu/class/cs231a/course_notes/01-camera-models.pdf

[87] Makinen T, Modeling Environment Using Multi-view Stereo [MS.c Thesis], Tampere: University of Tampere; 2019

[88] Ewbank T, Efficient and Precise Stereoscopic Vision for Humanoid Robots [MS.c Thesis], Liege Universite ; 2016-2017

[89] Savaresa S, Week 3: Epipolar Geometry. CS231A: Computer Vision From 3D Reconstruction to Recogntion, Stanford University; lecture given winter 2022, Available at: https://web.stanford.edu/class/cs231a/course_notes/03-epipolar-geometry.pdf

[90] Suominen O, Transform-based Methods for Stereo Matching and Dense Depth Estimation [MS.c Thesis], Tampere: Tampere University of Technology; 2012

[91] Stanford University. The Stanford Multi-camera Array [Internet]. [place unknown]: [publisher unknown].2002. Available at: http://graphics.stanford.edu/projects/array/

[92] Wikipedia. Light field camera [Internet]. [place unknown]: [publisher unknown].2012. Available at: https://en.wikipedia.org/wiki/Light_field_camera

[93] Anantpur J, Dwarakanath NG, Kalyanakrishnan S, Bhatnagar S, Govindarajan R. RLWS: A Reinforcement Learning based GPU Warp Scheduler. arXiv preprint arXiv:1712.04303. 2017 Nov 17.