FRAMEWORK FOR PARALLELIZATION OF BLOCK MATCHING FOR MOTION ESTIMATION AND STEREO MATCHING

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree of

MASTER OF TECHNOLOGY

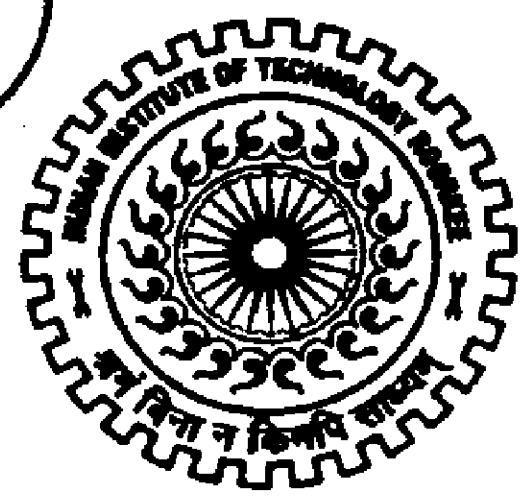
in

COMPUTER SCIENCE AND ENGINEERING

By

ABED MOHAMMAD KAMALUDDIN

M.G. CENTRAL LIBRARY
ACC No. G 20081
Date 26/10/10
I.I.T. ROORKEE



DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)

JUNE, 2010

Candidate's Declaration

I hereby declare that the work, which is being presented in the dissertation entitled, ***"Framework for Parallelization of Block Matching for Motion Estimation and Stereo Matching"***, which is submitted in the partial fulfillment of the requirements for the award of the degree of ***Master of Technology*** in ***Computer Science and Engineering***, submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee (India), is an authentic record of my own work carried out under the guidance of Dr Kuldip Singh, Professor and Dr. Ankush Mittal (Ex-Faculty), Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee. I have not submitted the matter embodied in this dissertation report for the award of any other degree.

Dated: 11-06-2010

Place: IIT Roorkee.

(Abed Mohammad Kamaluddin)

Certificate

This is to certify that above statements made by the candidate are correct to the best of my knowledge and belief.

Dated:

Place: IIT Roorkee.

14/6/10

Dr. Kuldip Singh,
Professor,
Dept. of E & CE,
IIT Roorkee.

Dr. Ankush Mittal,
Former Associate Professor,
Dept. of E & CE,
IIT Roorkee.

Acknowledgments

It gives me immense pleasure to take this opportunity to express my deep sense of gratitude to my guide Dr. Kuldip Singh, Professor, Department of Electronic and Computer Engineering, Indian Institute of Technology Roorkee, for his valuable guidance. I would also like to thank my guide Dr. Ankush Mittal, for his encouragement and constant motivation during the course of this work.

I would like to thank the staff of the Department of Electronics & Computer Engineering for their cooperation and the Institute Computer Center for making the resources available for the purpose of this dissertation work.

My special sincere heartfelt gratitude to my family, whose best wishes, support and encouragement has been a constant source of strength to me during the entire work. My friends and seniors also deserve special thanks for their support and valuable suggestions.

Finally, I would like to thank the Almighty to whom we are indebted for our very existence.

(Abed Mohammed Kamaluddin)

Abstract

SAD based block matching algorithms form the backbone of various image and video processing applications such as video encoding, 3D vision, video surveillance, robotics, image registration and contour mapping. Exploiting thread-level parallelism is a promising way to improve the performance of such algorithms for running on multi-core general-purpose processors and GPUs.

This thesis describes efficient strategies for implementation of block matching based algorithm for motion estimation and stereo vision on Intel multi-core architectures. A simple yet elegant GPU implementation of motion estimation has also been performed.

A multi-pass method to unroll and rearrange the multiple nested loops of the block matching has been exploited for parallelization using the OpenMP shared memory programming model to improve the performance of motion estimation on general-purpose multi-core processors. The results have shown good speedups of 7.01x over the sequential code performance on Intel Xeon Dual socket Quad-core processor and 1.69x on Intel Core2Duo processor. The GPU based parallel implementation using CUDA has also showed speedups up to 9.7x times.

A simple yet effective strategy has also been developed for parallelization of Stereo vision. The method determines the disparity between pixels in two images using a block matching technique and has been implemented on multi-core processors showing decent speedups of around 5x times.

This work also illustrates that by exploiting the capabilities of OpenMP and CUDA, a variety of exemplary tasks could be efficiently parallelized and the presently available general purpose multiprocessors and consumer level graphics cards can be more efficiently utilized. This should give software developers a compelling reason to multi-thread their applications.

Contents

List of Figures

List of Tables

Abbreviations

CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
MB	Macro Block
ME	Motion Estimation
NCC	Normalized Cross Correlation
SAD	Sum of Absolute Difference
SATD	Sum of Absolute Transformed Distances
SSD	Sum of Squared Differences

Chapter 1: Introduction

1.1 Introduction

Sum of Absolute Difference (SAD) is used as a similarity measure in case of block matching in several image-processing applications. SAD based block matching is one of the most commonly used techniques for motion estimation in various video coding standards [1][2] like H.26x and MPEG. It is also a frequently used routine in various other algorithms for motion detection in video surveillance algorithms [3], in Stereo Depth Analysis [4] and Correspondence Matching and forms a part of many image registration and fusion techniques.

Motion estimation technique is used to aid temporal prediction in video encoding. SAD calculation comes to play in the case of motion estimation and detection in the case of video algorithms due to the fact consecutive video frames have similarities and in most cases that can be intelligently exploited to reduce the number of bits in the encoding process or to detect motion. Most of the consecutive video frames will be similar except for the changes that might be induced by objects moving within frames.

Stereo Imaging is a powerful technique for determining the distance to objects and extraction of 3D information from digital images using a pair of camera spaced apart. This is fundamentally the same visual system used by humans and most other animals. Here, SAD based block matching is used to find the location of the possible correspondence points between image pairs. The extremely high computational requirement of stereo imaging limits its application to non-real time applications or to applications where high computational horsepower is available.

Many techniques have been proposed to speed up the computation of block matching. However most of the techniques are concerned with performance improvement of

hardwired implementations and FPGA based implementations [5] [6] [7]. As such popularity of SAD is limited to hardware architectures.

1.2 Motivation

With the advent of massively parallel GPUs and multi-core architectures it is now possible to achieve the desired speedups that were earlier possible only on dedicated hardware. The newly available multi-core processors like Intel Core2Duo and Quad-core and Intel core i series of processors (Core i3, i5 and i7) (with 2/4 processing cores and 4/8 hardware threads) have a massive amount of computing power that may be exploited for motion estimation and stereo matching.

In order to take advantage of these multiple cores, software applications need to be multi-threaded. A single threaded application will only execute on a single core at any given time and leave the other cores idle. In a well threaded application, the workload is divided into equal chunks and distributed evenly to the available cores. The explicit parallel programming offered by OpenMP shared-memory programming model and CUDA provide a rich set of features, which allow us to exploit thread-level parallelism and optimize the performance of applications.

1.3 Problem Statement

This dissertation work intends to create a framework for the parallelization and implementation of block matching algorithm on multi-core architectures and analyze their performance on these systems with respect to motion estimation and stereo matching. In this work strategies have been devised that can efficiently utilise the parallel architectures available like multi-core systems and GPUs for solving the problem by harnessing the increased computing power at our disposal in form of the parallel architectures present. Algorithms using other correlation criteria apart from SAD which we have used for demonstration may utilize the same framework for different applications.

1.4 Chapter Organization

The dissertation is organized as follows.

- In Chapter 2 we discuss an overview of the parallel architectures used and a description of OpenMP and nVidia CUDA.
- In Chapter 3 we discuss the basics of motion estimation using block matching.
- In Chapter 4 we have discussed the strategy for parallel implementation of motion estimation on multi-core and GPU as well as the results obtained.
- In Chapter 5 we discuss the basics of Stereo Vision as well as use of block matching for disparity calculation.
- In Chapter 6 a novel strategy for multi-core implementation of stereo vision and its performance has been discussed.
- Finally Chapter 7 is devoted to the conclusions and future work.

Chapter 2: Multi-core Computing

2.1 Parallelization Using OpenMP Compiler

On a multi-core architecture the programmer has to decide how the work should be distributed across multiple processors in contrast to the programming of sequential tasks on single-core architecture. The POSIX thread library is often used to develop parallelized or multi threaded code which is a tedious task for large applications. Alternatively, this additional development step can be realized using the parallel programming model of OpenMP for shared memory multiprocessors [8]. It provides the advantage to simplify managing and synchronization of program threads. OpenMP works in conjunction with the prevalent programming languages C/C++ and FORTRAN [9]. OpenMP provides a set of compiler directives and a supporting library of subroutines that control the distribution of tasks over the processor cores and the necessary synchronization of these tasks. The OpenMP API is independent of the used platform and operating system. Appropriate compilers exist for a variety of all major operating systems.

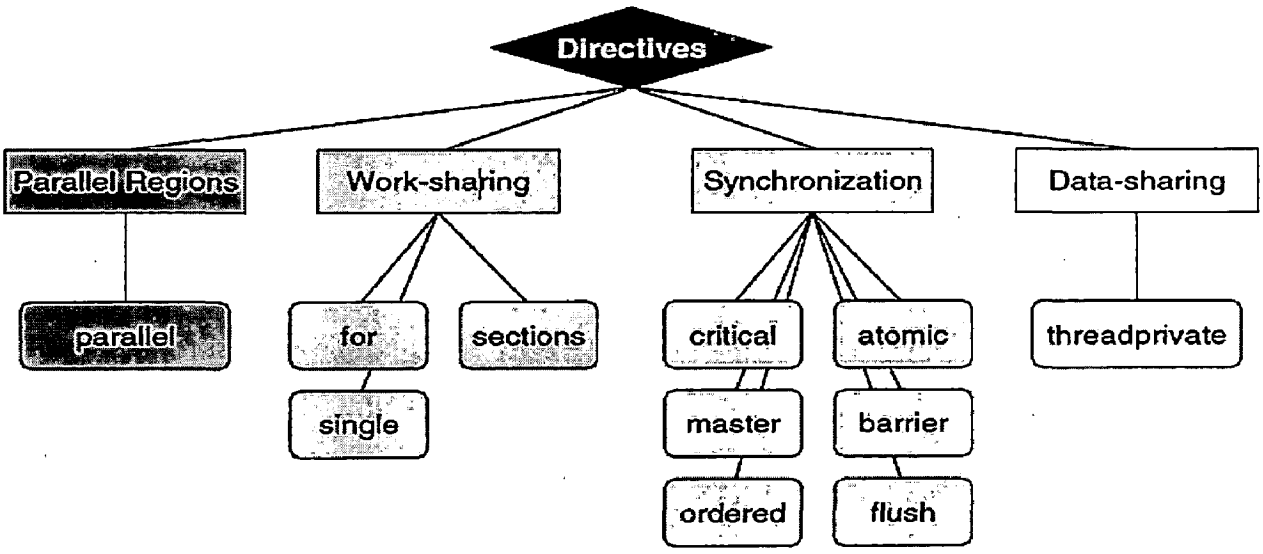


Figure 2.1: Overview of main OpenMP directives.

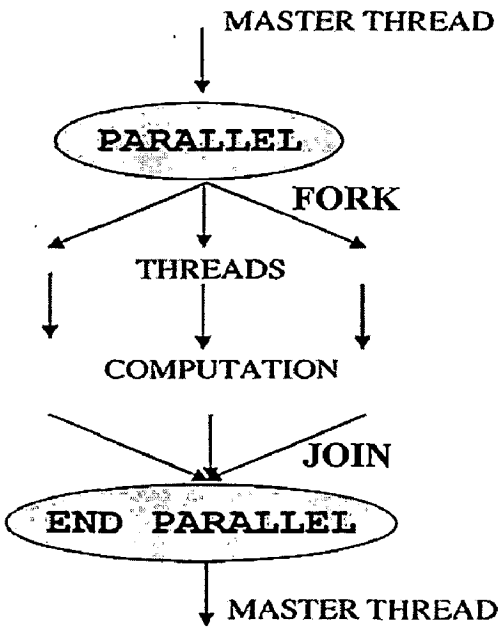


Figure 2.2: Fork-Join Model.

The directives which in the case of C/C++ programs take the form of #pragmas which are instructional notes to any compiler supporting OpenMP (e.g. Intel C++ Compiler, GCC 4.2). To enhance application portability these pragmas are ignored by any compiler not supporting OpenMP. This directive-based parallelization approach has the benefit that it allows the same source code to be used for single and multiprocessor development, since the code will be executed serially on single core and in parallel on multi-core processors. Another benefit of this approach is that it allows an incremental parallelization approach starting from an existing serial version by adding parallel code regions step by step.

The OpenMP language extensions can be separated into control structures for expressing parallelism and work-sharing on the one hand and data environment constructs for inter-thread communication and synchronization on the other [10]. Figure 2.1 gives an overview of the most important OpenMP directives. The control structures for parallelization (i.e. parallel) are embedded into a so-called fork/join execution model. Thus, they fork (i.e. start) new threads and execute an enclosed code block concurrently, and afterwards they join in parallel running threads to a serial master thread. This has been depicted in Figure 2.2. The work-sharing directives can be used to divide the work within a code block into an

existing team of threads. OpenMP features scheduling options which assign chunks to threads either statically or dynamically in order to optimize the loop performance. By default each thread is statically assigned one chunk of iterations of (nearly) equal size. Besides the parallelization of a single task, it is possible to easily assign a sequence of independent tasks to different threads with the sections work-sharing construct. The required thread synchronization can be done implicitly by OpenMP e.g. at the end of a work-sharing construct (join) or explicitly by the programmer through directives like barrier or critical. In order to reduce synchronization overhead, the implicit barriers can be explicitly removed with the no-wait clause if no synchronization is required between consecutive work-sharing constructs within a parallel region

2.2 GPU Computing

A GPU is a highly parallel computing device designed for the task of graphics rendering. However, the GPU has evolved in recent years to become a more general processor, allowing users to flexibly program certain aspects of the GPU to facilitate sophisticated graphics effects and even scientific applications. In general, the GPU has become a powerful device for the execution of data-parallel, arithmetic (versus memory) intensive applications in which the same operations are carried out on many elements of data in parallel. Example applications include the iterative solution of PDEs, video processing, machine learning, and 3D medical imaging.

The performance of GPUs is improving at a rate faster than that of CPUs as can be deduced from Figure 2.3. The capabilities of the GPU have increased dramatically and the current generation of GPUs has higher floating point performance than the most powerful (multi-core) CPUs [11]. The GPU contains hundreds of cores that work great for parallel implementation. The programming is done in SIMD style where same code is worked on different data locations. Until recently a graphics API was needed to code on GPUs which made coding for non graphics oriented calculations tough. Trying to work around this limitation nVidia released CUDA which allows GPUs to be programmed using a variation of C. This enables a low learning curve and makes programming easier.

The three abstractions of the CUDA model are a hierarchy of thread groups, shared-memories, and barrier synchronization. Threads are arranged in the form of a grid which is a two dimensional array of thread blocks. Each thread block is a three dimensional structure that houses the threads. This type of hierarchy is given to the programmer so that the arrangement of the threads is similar to the way programmer's data is arranged (in arrays). Threads within a block can cooperate among themselves by sharing memory. Shared memory is expected to behave like an L1 cache where it resides very close to the processor core. Synchronization points can be specified by calling the function __syncthreads.

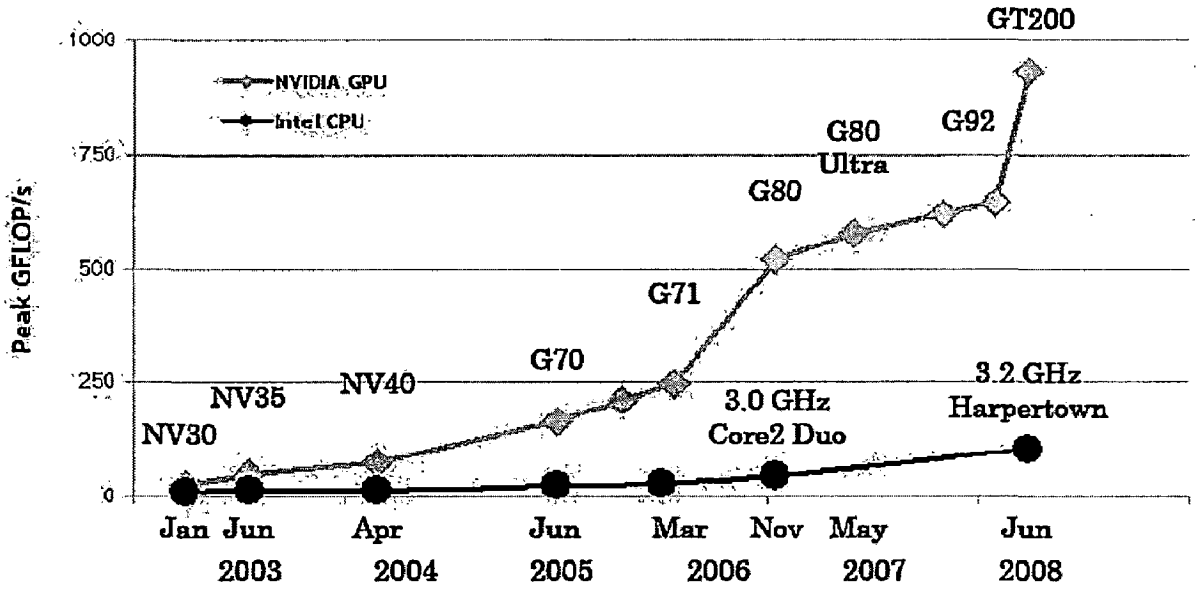


Figure 2.3: Floating point operations for the CPU and the GPU [11]

The memory available to the threads is of three types. Every thread has local memory. Number of threads which are in the same thread block can share memory. And the third type of memory is the global memory that every thread has access to. C code for both the GPU and the CPU resides in the same file. The CPU code follows a sequential flow. GPU code is called by a kernel call. This is where the code runs in parallel. A large number of threads are created by the kernel call. These threads then run in parallel on the GPU.

2.2.1 Programming Model

In this section various aspects of the CUDA programming model have been discussed.

2.2.1.1 Thread Hierarchy

Threads in CUDA are arranged in the form of a hierarchy. A number of threads house within what is known as a thread block. These thread block can be 1 dimensional, 2 dimensional or 3 dimensional. These thread blocks are placed in a structure known as thread grid. Thread grid can be either 1 dimensional or 2 dimensional.

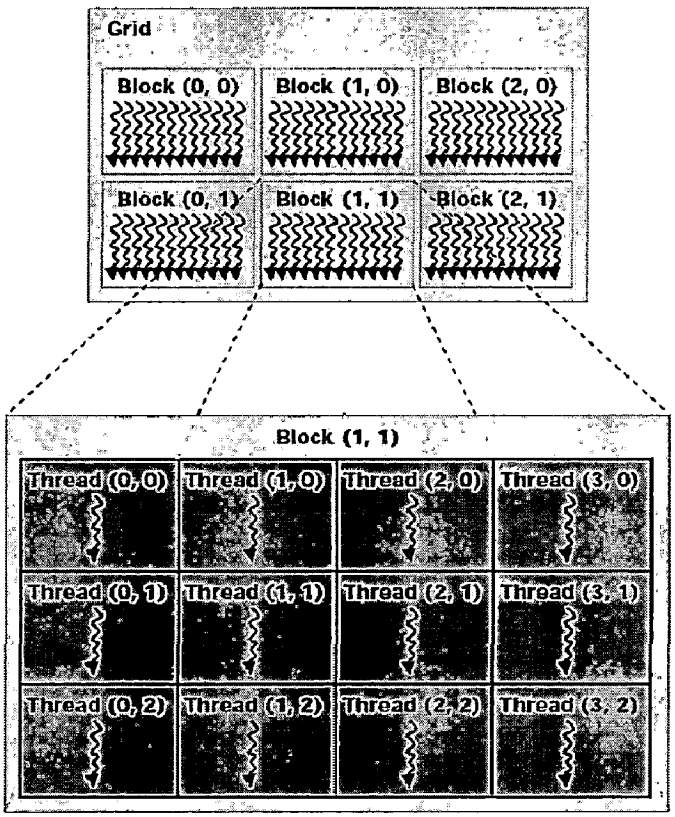


Figure 2.4: Figure showing arrangement of threads [11]

A maximum of 512 threads can be placed in a thread block. Thread block are expected to run independently of each other. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling scalable code to be written. Proper selection of grid size and block size is important to gain good speed up.

2.2.1.2 Memory Hierarchy

Threads may access memory from different memory spaces during their existence. Threads may declare local variable, may share memory with other threads that belong to the same block or may be accessing global memory.

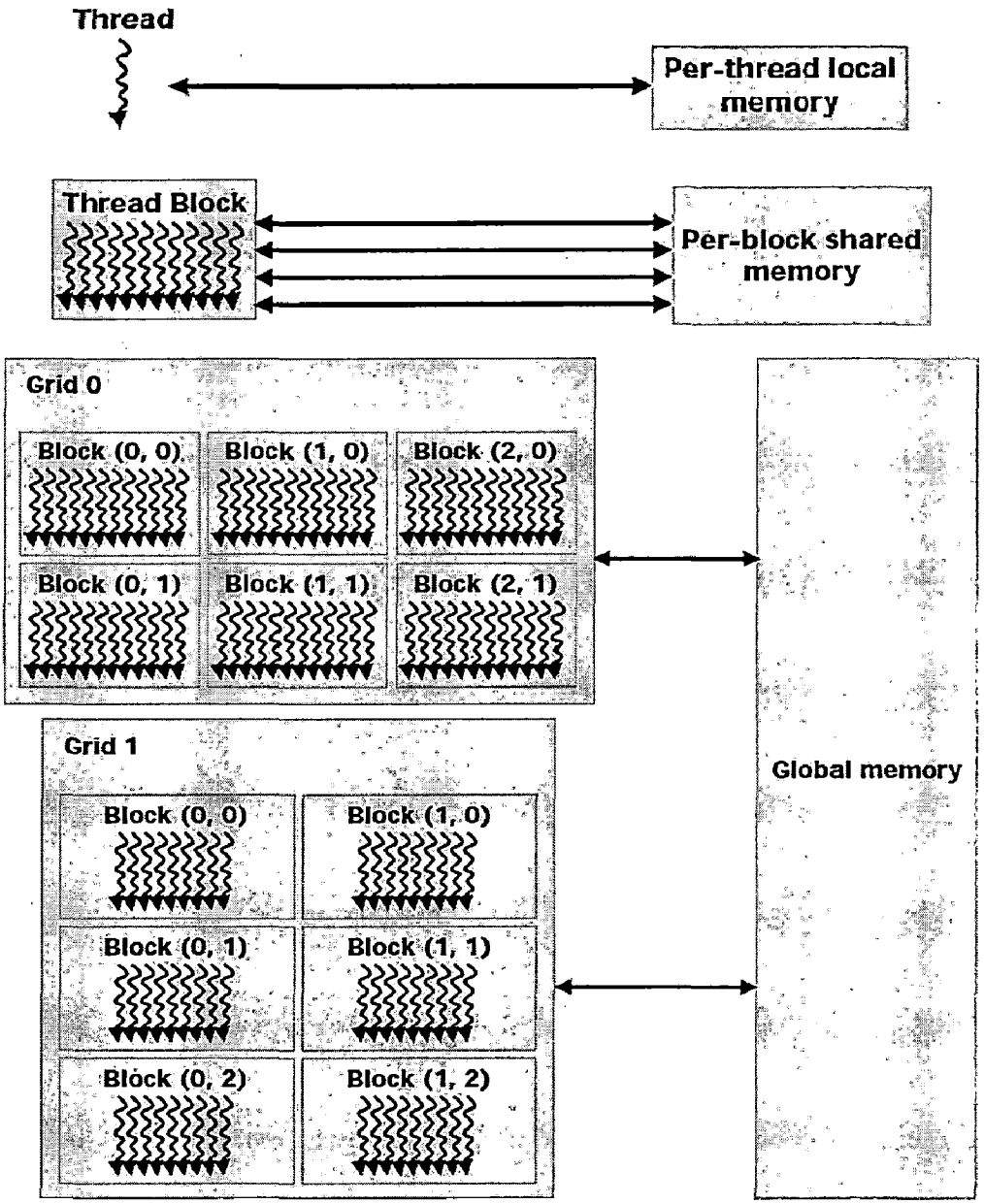


Figure 2.5: Figure showing how threads access global, shared and local memory [11]

2.2.2 GPU Implementation

When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors. A multiprocessor consists of eight Scalar Processor (SP) cores. Every multiprocessor has 8192 registers of 32 bit size each. The multiprocessor creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. The general idea is to achieve very fine grained parallelism by assigning one thread to work on one data item which is pixel of an image in our case.

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a warp start together at the same program address but are otherwise free to branch and execute independently. When a multiprocessor is given one or more thread blocks to execute, it splits them into warps. The way a block is split into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data dependent conditional branch, the warp serially executes each branch path taken; disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

A multiprocessor can work on a maximum of 8 thread blocks. However, if the thread code required a large number of registers then lesser number of thread blocks is assigned to a multiprocessor. In case a thread block is too bulky to be assigned to a multiprocessor then in such cases the kernel simply fails to launch.

Chapter 3: Motion Estimation

3.1 Introduction

A wide range of applications like digital TV, DVD, Internet streaming video, video conferencing, distance learning, surveillance and security make use video coding [1] [2] [3] as the central technology. A variety of video coding standards and algorithms have to address the requirements and operating characteristics of different applications.

High definition video coding is presently considered to be a very compute intensive process. H.264 [1] is one such current video coding standard which is aimed at high-quality coding of video contents at very low bit-rates. The new standard significantly improves the compression efficiency compared with existing standards, such as H.263+ and MPEG-4[12] and uses the same hybrid block-based motion compensation and transform coding model as existing standards. Moreover, a number of new features and capabilities like variable block-size motion compensation (MC), quarter-pixel accuracy MC, have been introduced in H.264 and as the standard becomes more complex, the encoding process requires much more computation powers than most existing standards. Therefore high-definition video encoding is still difficult to process in real-time on CPU [3] even with highly optimized code. Hence, we need a number of mechanisms to improve the speed of the encoder.

Tools	Datapath Operation (MIPS)	Percentage (%)
Motion Estimation	24,768.2	97.94
Transform & Quantization	432.527	1.710
Others	88.0320	0.348

Table 3.1: Instruction profiling results of MPEG-4 encoder

A number of studies have analyzed the computation profile of video encoders and shown that motion estimation is the most compute intensive procedure of video encoding. An

analysis[13] of MPEG-4 has shown that motion estimation forms 97.94% of the total computation which highlights the fact that improving its performance would speedup video encoding. From Figure3.1 [14], which shows the execution time breakdown for H.264 encoding, we can see that the motion estimation algorithm accounts for as much as 86% of the total execution time.

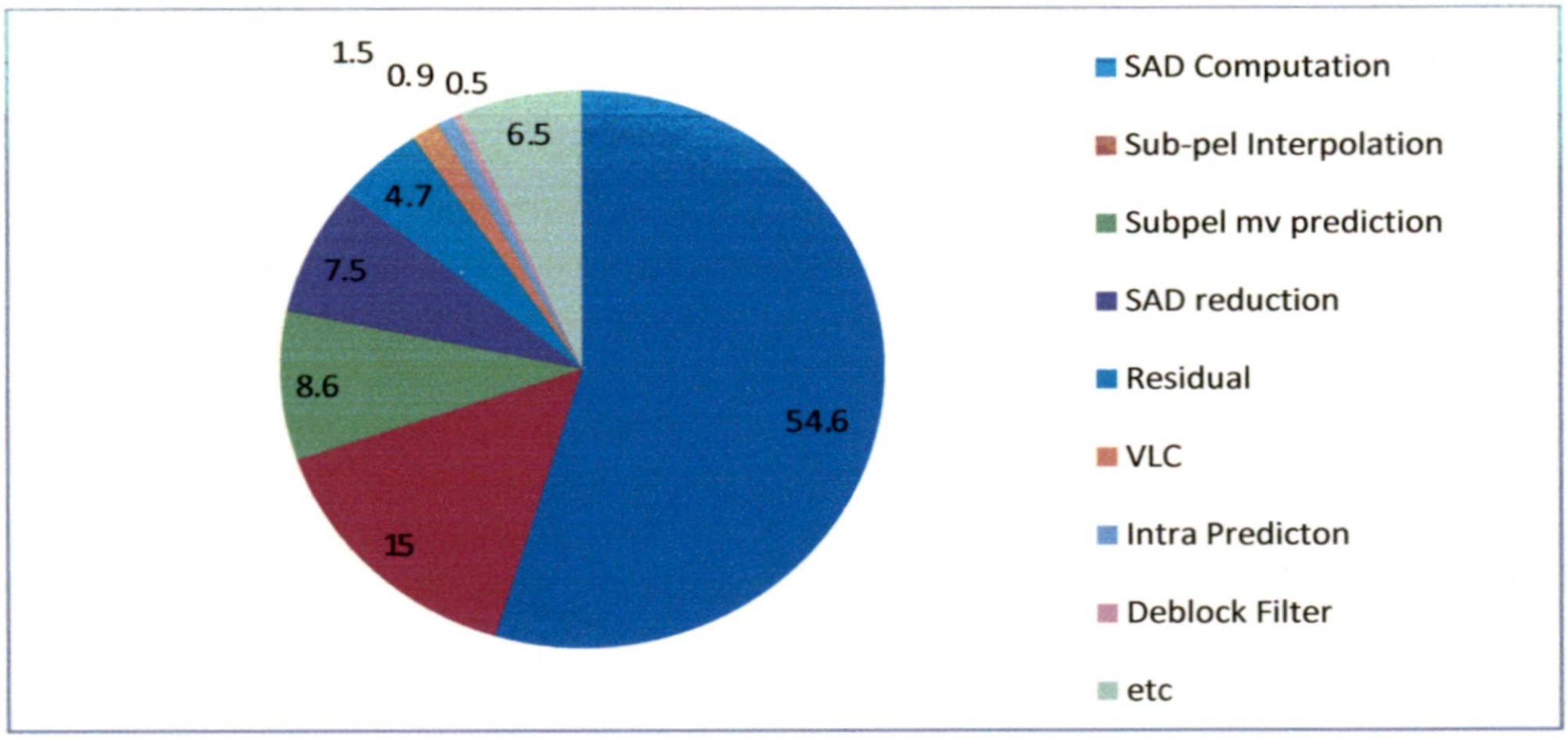


Figure 3.1: Breakdown of execution time for H.264 encoding.

A motion estimation algorithm exploits the temporal redundancy between frames. A video frame is broken down into macroblocks (each macroblock typically covers 16×16 pixels) and each macroblock's movement from a previous frame (reference frame) is tracked and represented as a vector, called motion vector. Storing this vector and residual information instead of the complete pixel information greatly reduces the amount of data used to store the video. Among many motion estimation algorithms, we adopted the exhaustive search algorithm which is more suitable for parallelization than the other algorithms due to its simplicity.

In the past, motion estimation has been accelerated by the use of dedicated hardware of multiple parallel processing elements (PEs) [5] [6]. Recently, various GPU-based motion estimation algorithms and methods [15] [16] have also been proposed.

3.2 Block Matching for Motion Estimation

Motion estimation is defined as searching for the best motion vector, being the displacement of the coordinates of the most similar block in the previous frame compared to the block in the current frame. Full-search block-matching is the most popular algorithm to perform ME, and it searches through every candidate location to find the best match. To do this, the current frame is partitioned into two-dimensional blocks (typically 8x8 or 16x16 pixel blocks) and a search window (typically 32x32) in the reference frame is defined. Each block of the current frame is compared with all the blocks of a previous frame within the same window.

The final motion vector corresponds to the block with minimum distortion within the search window. The most commonly used metric to calculate the distortion is the SAD. However other metrics like SATD, Normalized Cross Correlation, SSD etc. may also be used. The general algorithm for computing the sum of absolute difference is depicted in the equation **3.1**.

$$SAD(x,y,i,j) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} |A_{x+u,y+u} - B_{x+i+u,y+j+u}| \tag{3.1}$$

In this equation (x,y) is the location of current block in the image and (i,j) is the motion vector specifying the block shift and NxN is block size.

The calculation of the SAD value between a reference and a candidate block is performed as follows. For every pixel in the reference block, the value of the corresponding pixel in the candidate block is subtracted from it and the absolute value is taken. In other words the absolute difference is taken between two pixel values. Lastly, all these absolute differences are summed up, to obtain the sum of absolute difference (SAD) value of a block. The SAD value can be seen as an error value i.e. if the SAD value equals zero, the blocks are exactly the same.

Figure 3.2 describes how search area is organized for every current Macroblock (MB). Corresponding pairs of pixels are used to calculate SAD values for one reference MB in search area. Corresponding pairs of pixels are paired by same position in reference and current MB. Number of pairs per MB matches number of pixels in MB. All SAD values from every pair for one MB are accumulated and that accumulated SAD value represents SAD value for that reference MB in search area at the end there are search area number of SAD values for every current MB in a frame.

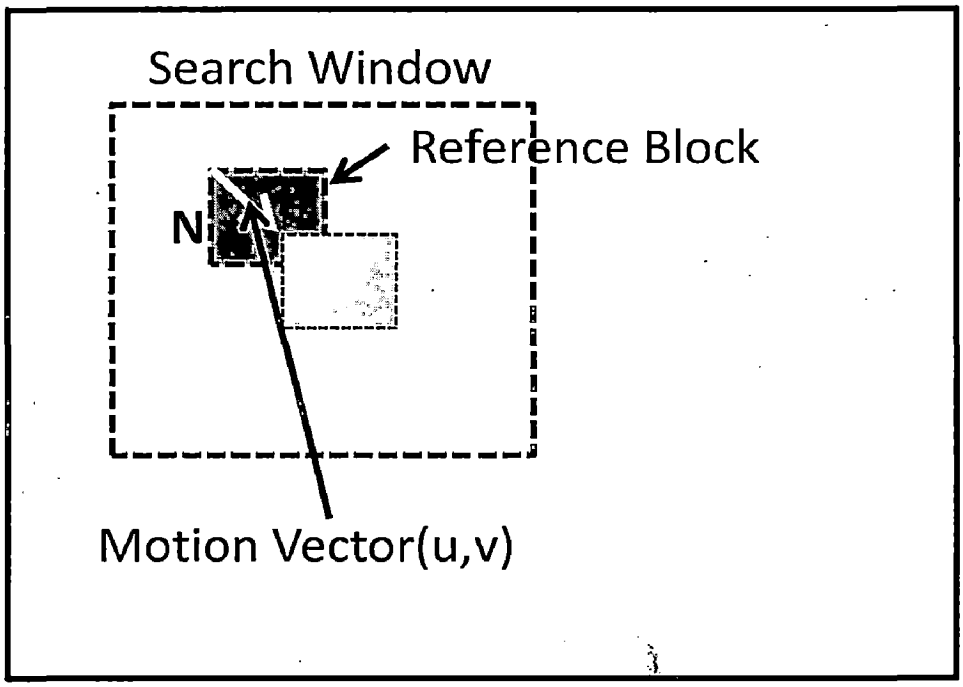


Figure 3.2: Search area for one current block that has to be covered using full search in motion estimation

For block matching, the two source frames are divided into small blocks, with typical values of 4 x 4,8 x 8 or 16 x 16 pixels. Then, for each block in the first frame, the algorithm searches for a similar block in the second frame. Lastly, the block from the original frame is set in the new intermediate frame at a position in between the matched blocks of the two target frames. This way, motion is interpolated by means of a matching process between two blocks. The pseudo code for motion estimation is given in Figure 3.3.

```
For each block in the image{
best_sad = Infinity; // Initialize best sum of difference with infinity
For each candidate position{
sad = compare_blocks (candidate_block, reference_block_in_old_frame);
if (sad < best_sad) {
best_sad = sad;
best_block = candidate_block; }
}
output_position(best_block);
}
compare_blocks(a,b){
sum = 0;
For each pixel p {
difference = a[p] - b[p];
sum += abs(difference);
}
return sum;
}
```

Figure 3.3: Pseudo code for Motion Estimation

In more detail, the different steps can be distinguished for each block - called the reference block and the candidate blocks:

1. First of all, the reference block from the first frame must be compared with a number of candidate blocks from the second frame. Typically, this is done within a search window, limiting the number of comparisons. Typical window size is of 32 x 32 pixels. For example, within a window of 32 times 32 pixels a total of 256 different 16 times 16 blocks must be compared using a sum of absolute difference (SAD) technique.

2. The result of the comparisons denotes the similarity to the reference block. All obtained results need to be sorted and the best candidate needs to be found - the winning block. In the case of a SAD comparison, the candidate with the lowest SAD value is the winning block.

Chapter 4: Motion Estimation on Multi-core

This section describes the implementation of block matching based motion estimation on a multi-core CPU as well as on a Graphical Processing Unit. The strategy used for parallelization has also been mentioned in this chapter.

4.1 Motion Estimation on Multi-core using OpenMP

The full search motion estimation algorithm is complex and time consuming because multiple nested loops are required. Beside the sum of absolute difference (SAD) computation and SAD comparison, in the classic ME algorithm mentioned in Figure 4.1, there are four nested loops.

Due to the high regularity and weak data dependencies the parallelization of this algorithm with OpenMP is straight forward. However we have to decide which for-loop to be parallelized. Here, it is best to select the outermost loop, which controls the vertical iteration over the search blocks, since there are no data dependencies between the SAD calculations of the single blocks. Due to this, the workload being distributed over simultaneously running threads can be maximized and the synchronization overhead can be minimized.

The approach of distributing SAD calculations per block is shown in Figure 4.2. This approach has been implemented by for use in embedded devices. However the disadvantage of this approach is that it allows parallelization only on a single level. However as the classic algorithm shows that there are four different loops and the internal loops can also be run simultaneously due to the absence of data dependencies.

However, nested looping is difficult to implement in OpenMP and may also lead to load imbalance and significant overheads. In our parallel ME algorithm implementation shown in Figure 4.3, the four classic nested loops are unrolled and rearranged to a one-

tiered loop and a two-tiered loop. This two-pass approach is better suited to multi-core architecture and is therefore used in our implementation. This strategy would also more effectively utilize the L1 cache (32KB per processor core in our case) and would also reduce the complexity of parallelization caused due to the nested looping. Assuming n cores on the multi-core CPU, all the operations would be scheduled, folded and executed concurrently on the n cores.

```
Loop(rows of macro blocks (MBs)){
    Loop(columns of MBs){
        Loop(rows of search range (SR)){
            Loop(columns of SR){
                SAD computation;
                SAD comparison;
            }
        }
    }
}
```

Figure 4.1: The pseudo code of the classic ME algorithm

```
#Parallel region
Loop(rows of macro blocks (MBs)){
    Loop(columns of MBs){
        Loop(rows of search range (SR)){
            Loop(columns of SR){
                SAD computation;
                SAD comparison;
            }
        }
    }
}
```

Figure 4.2: Basic Parallelization of ME algorithm

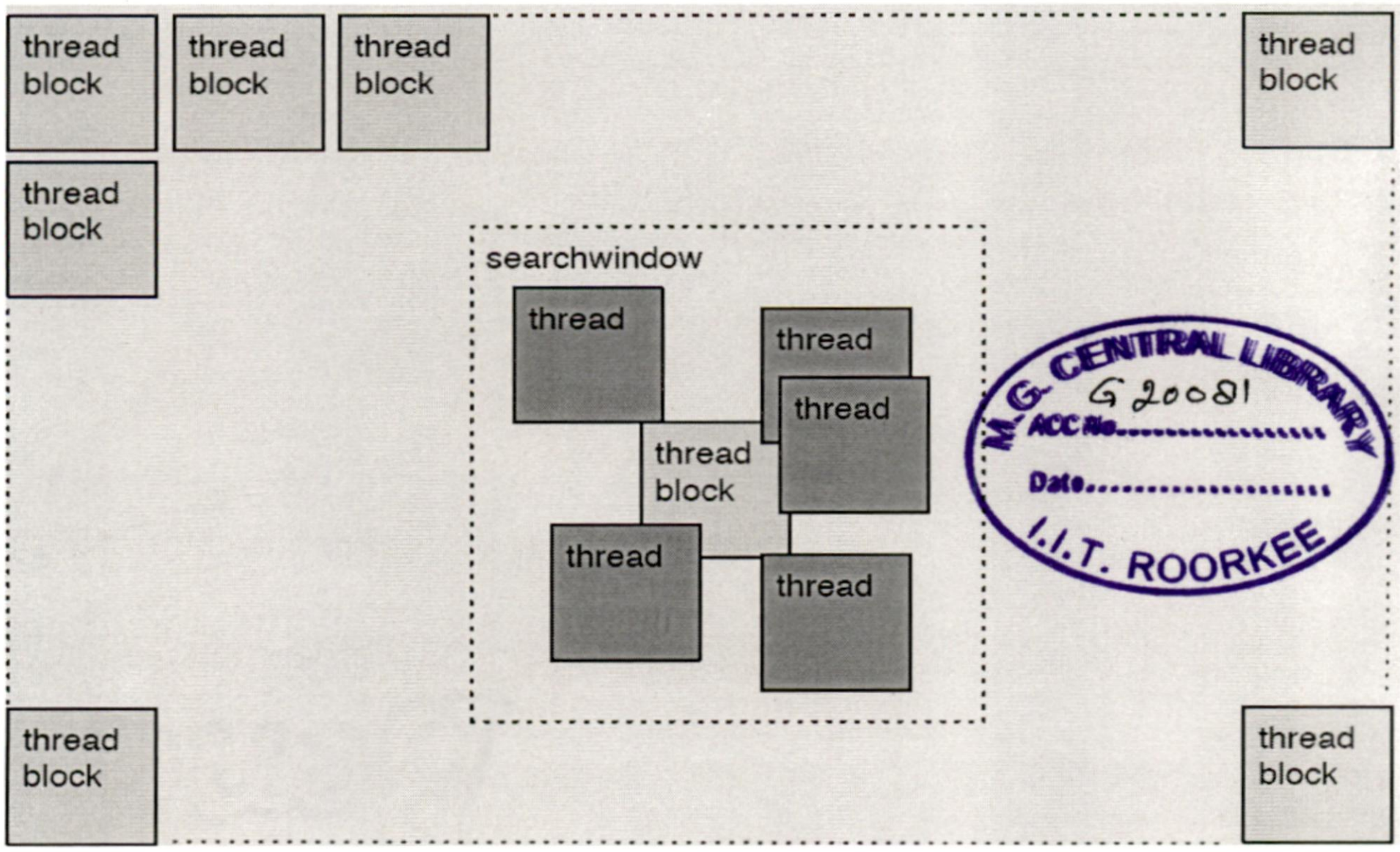
```
#Parallel region1:
Loop(candidates per core/thread){
        SAD computation;
      }

#Parallel region2:
Loop(rows of a MB){
  Loop(columns of a MB){
    SAD comparison;
  }
}
```

Figure 4.3: The pseudo code of ME algorithm with two parallel regions

Therefore we have implemented a variant of this ME algorithm which is executed in two passes. In the first pass, we perform SAD computation and the SAD values corresponding to an MB is generated. In order to perform the full search ME, the number of candidates are grouped and mapped to one core or one thread (in the case of multiple threads being assigned per core) in the corresponding MB. This is the only loop in the first pass. In the second pass, each core compares the SAD values computed from the first pass in each MB, to find the smallest global minimum SAD value and the corresponding motion vector. This has been illustrated in Figure 4.3.

4.2 Motion Estimation on GPU using CUDA

We have used a single kernel approach. First, the complete frame is divided into blocks with a size equal to the chosen reference block size, for example 16 times 16 pixels. These blocks are mapped one on one onto threadblocks. So, the number of threadblocks is equal to the number of reference blocks in the image. This division is depicted in Figure 4.4.



M.G. CENTRAL LIBRARY
G 20081
ACC No.....
Date.....
I.I.T. ROORKEE

Figure 4.4: Mapping of thread blocks

Since each thread-block now represents all the processing involved with one reference block, the details for each reference block can be described according to these steps:

1. First, the reference block needs to be compared with each candidate block in the second frame. In order to do so, a number of threads is instantiated within the threadblock, equal to the number of candidate blocks available. The task of each thread is to calculate one SAD value, i.e., to compare one candidate block with the reference block.

2. After the first step is complete, the winning candidate i.e. candidate with minimum SAD needs to be found. The result of the previous step consists of a number of SAD values, equal to the number of threads in the threadblock.

In pseudo-code, the complete algorithm as described can be summarized as seen in Figure 4.5.

```
for all threadblocks
  for all threads
    SAD1 = SAD( referenceBlock , candidateBlock )
  end
winningBlock =  minimum(SAD1 )
for all threads
writeData ( winningBlock )
end
end
```

Figure 4.5 Pseudo-code for implementation on CUDA

As discussed in Section 2.2.1, within a threadblock, a fast shared memory is available. For the SAD comparison the reference block is used for every thread, while the candidate blocks are partially overlapping each other. A typical reference block can fit easily in the shared memory, leaving space to schedule other threadblocks with their reference blocks onto the same SM. In this way, the reference block is stored once in fast local memory and can be shared among each thread. In order to do so, each thread loads zero or more pixels from the reference block into the shared memory. Because the number of threads can differ from the number of pixels in a reference block due to parameters for the algorithm, control has to be added in the case of unfixed parameters at design time.

In the second step of the block matching algorithm, the shared memory is used again, this time to communicate all the resulting SAD values and to do the comparisons. Apart from the shared memory, grouping threads into threadblocks makes synchronization between threads possible. As seen from the algorithm, between each of the three steps a synchronization barrier is needed, since the algorithm must completely finish each step before being able to proceed to the next step.

4.3 Performance Evaluation

The tests have been performed on a Dell Precision T7400 workstation with an Intel Dual Socket Quad-core with 256 KB of L1 cache (32KB x 8 cores) and 12 MB of L2 cache and 8 GB of RAM. The other PC used has a Core 2 Duo P8400 2.26GHz Processor with 128KB L1 cache and 3MB L2 cache and 3GB of RAM.

All the tests have been performed using Linux Fedora Core 11 operating system. The program to perform motion estimation is written in C programming language. The C compiler used is GCC version 4.4.0 and the OpenMP package used is OpenMP 2.0.

The nVidia GPU used for analysis is Quadro FX 4600. This GPU card is attached to the Dell Workstation. The CUDA compute capability of the GPU 1.0. This device has 768 MB of memory and has 12 multiprocessors of 1.2 GHz each.

The sequential implementation has been labeled as serial code. There are two parallel implementations. The first is the parallel implementation using OpenMP for execution on general purpose multiprocessor. The second parallel implementation using CUDA is labeled CUDA code and has been executed on the GPU. A video sequence 'taxi' with frame size 640 x 480 and 256 x 191 has been used as input with a search range of 32 and varying block sizes.

Figure 4.6 depicts the execution time of the sequential and parallel program for completely processing one frame of size 640 x 480 in milliseconds for all the 3 implementations and for different block sizes. Table 4.1 shows the speedups obtained for implementations using different blocks and frame sizes. Therefore it can be seen that speedups of about 7 times have been obtained on Intel architecture and around 9 times on nVidia GPU.

Figure 4.7 shows the execution time bars for frames of different sizes using a search window size of 32 and block size as 4. It can be seen that as the frame size increases and we move on to higher resolution videos the computation costs would become extremely

high. The comparative performance on different number of cores (2 cores and 8 cores) shows that speedup scales linearly with the increase in the number of cores. We have also varied the number of threads to analyze the performance. The optimum performance in the case of the Xeon eight core processor was achieved when using 16 threads. However on increasing the degree of multithreading to 32 the performance was reduced slightly due to threading overheads. This has been depicted in Figure 4.8. In Figure 4.9 we have plotted the motion vectors for an image of size 480 x 640 using a search window size 32 and block size 16 using OpenCV.

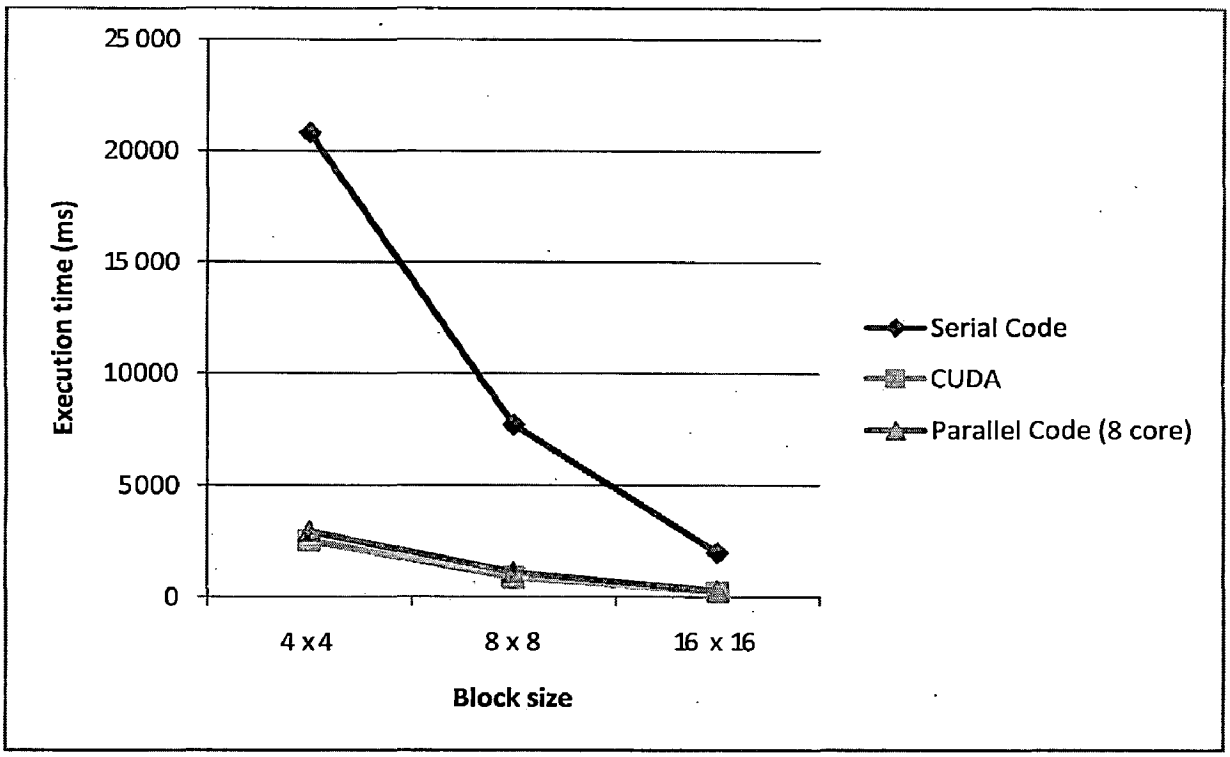


Figure 4.6: Execution times for different block sizes

Processor	No. of Processing cores	Average Speedup Obtained
Parallel Code (Core2Duo)	2	1.69
Parallel Code (Xeon 8 core)	8	7.01
CUDA (Quadro FX 4600)	12	9.7

Table 4.1. Speedup Obtained versus No. of Cores

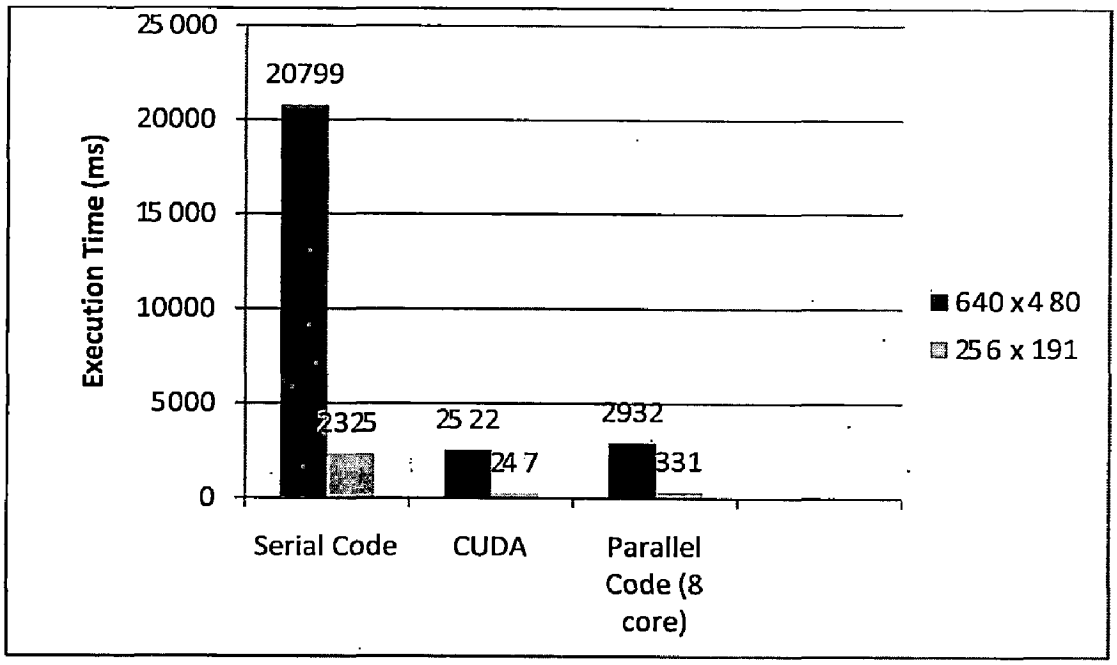


Figure 4.7: Comparison of processing times of different sized frames

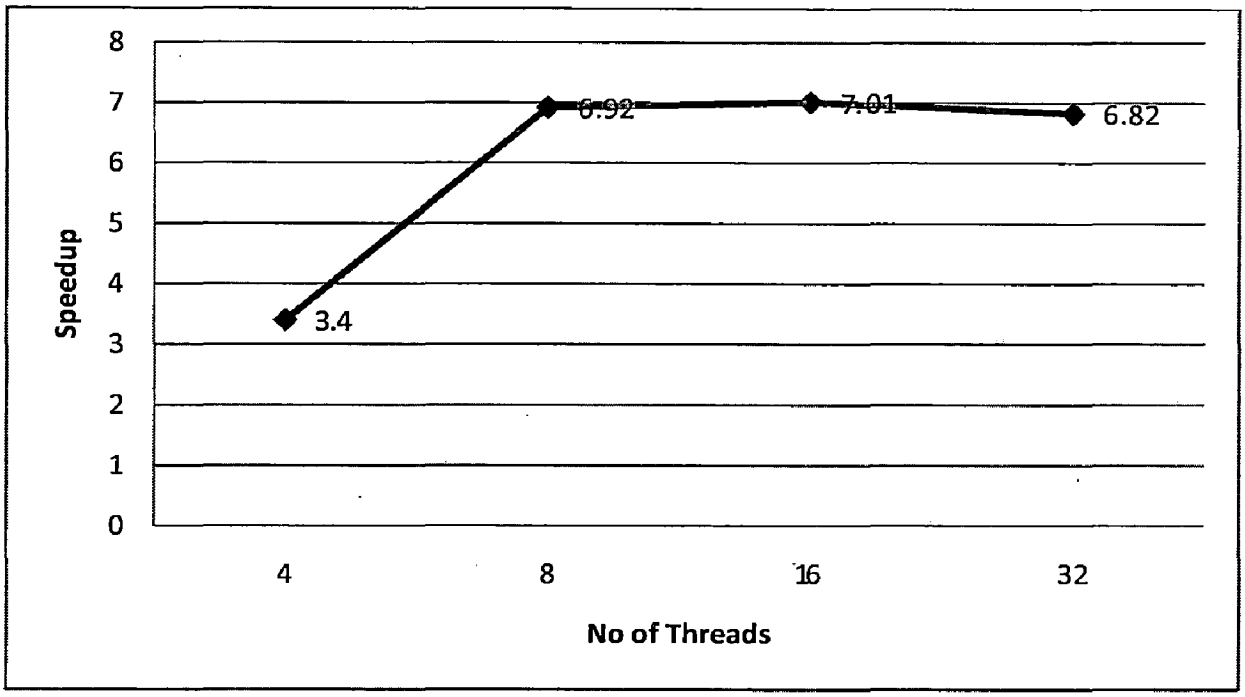


Figure 4.8: Speedup obtained on Intel Xeon dual Socket Quad-core versus the no. of threads

Chapter 5: Stereo Imaging

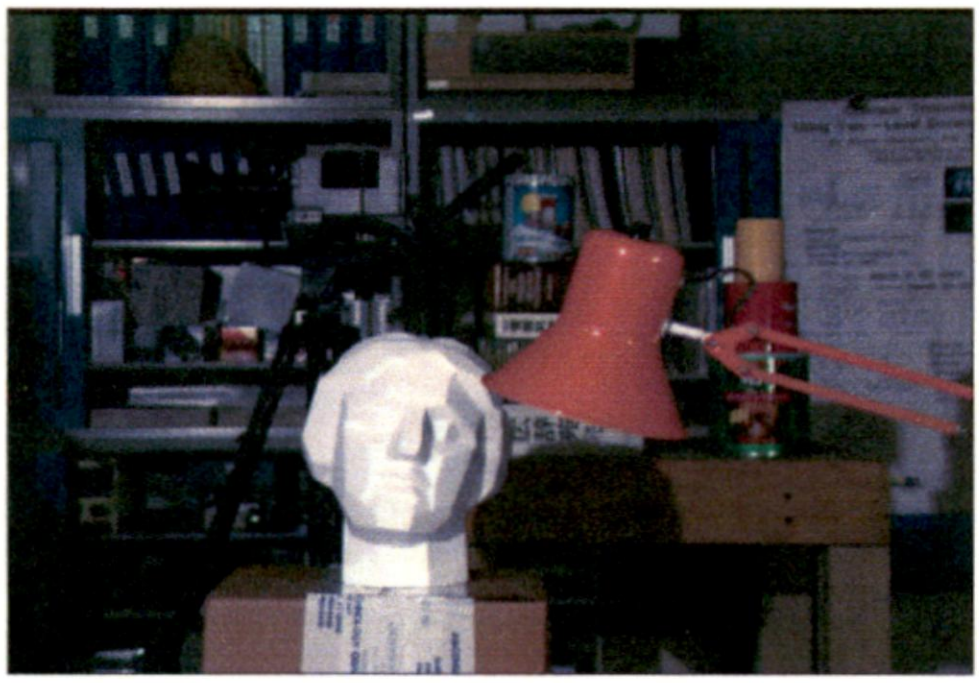
5.1 Introduction

Given a series of two-dimensional images it is possible to extract a significant amount of auxiliary information about the scene being captured. One of the most useful of these pieces of information is knowledge about the relative depth of objects in the scene. It is known that, given two images of a single scene it is possible to extract the depth of various objects in the scene from the disparity between the two images. The human brain handles this task constantly, adapting a stream of paired two dimensional images to provide us with what is commonly known as depth perception: that is a feel for the relative depth of objects in the scene. In a much simpler case we can consider how to extract this scene disparity from two images viewing the same scene from close but not identical positions. It should be noted that without a significant amount of extra information and calculations, it is generally not possible to ascertain an exact depth measurement but rather we can isolate "planes" of depth, i.e. localize which parts of the scene are at the same, or relatively close, depths.

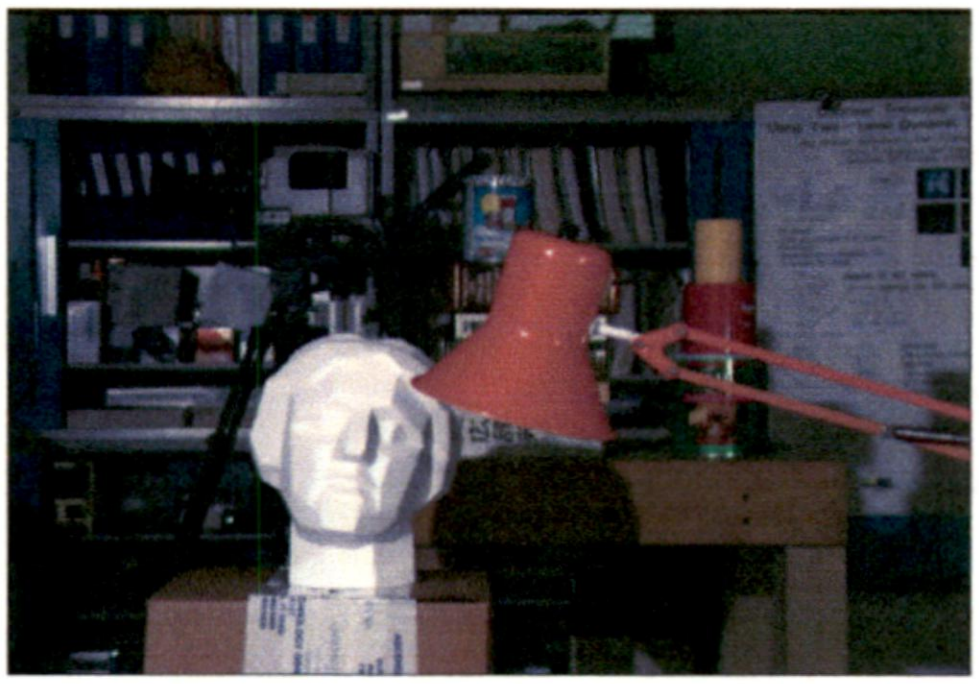
Being able to retrieve this depth information is useful for many applications. Stereo vision is highly important in automated systems such as robotics and auto-guided vehicles to extract information about the relative position of 3D objects in their vicinity, for object recognition, where depth information allows for the system to separate occluding image components. Scientific applications for digital stereo vision include the extraction of information from aerial surveys, calculation of contour maps or surface recovering for automatic 3D-model acquisition.

In order to obtain the desired depth information we need to first determine the disparity between the two images. Traditionally these two images are referred to as the left and right images. Since the left and right images are viewing the plane from different place, there will be a noticeable disparity between the two images. If we are able to calculate the relative disparity between points in a scene across the two different images we should be

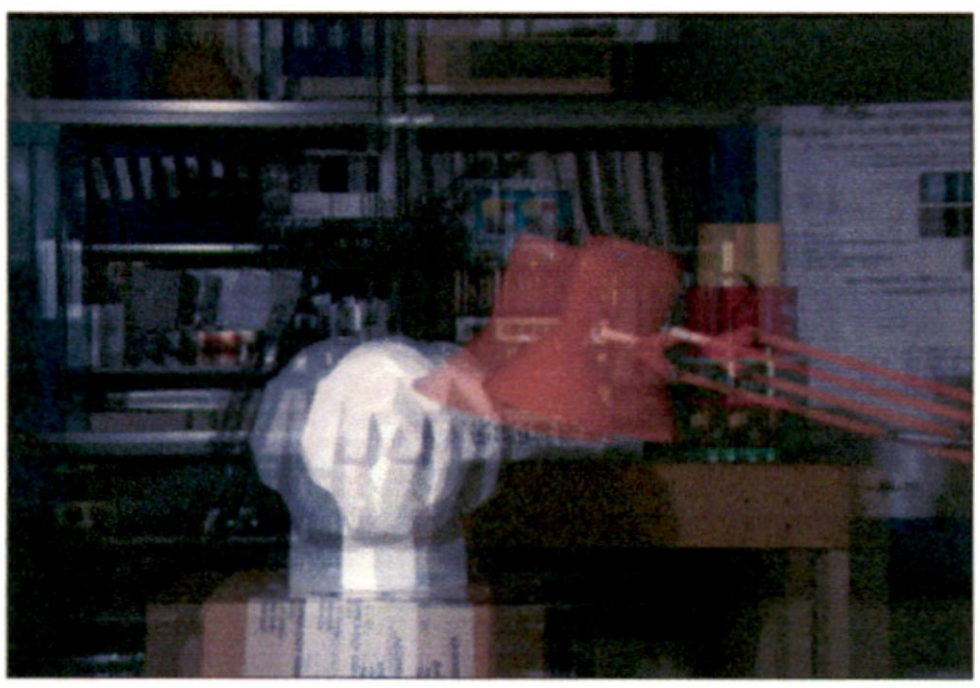
able to create a depth map from that information. The vital point here is that points at similar depth levels in the world will have similar disparities across the left and right stereo images. This is similar to moving our head laterally: objects close to move a large distance in the field of view while those further away move a small distance. By determining a displacement for each point in an image, we can determine roughly which depth layer it belongs to. Thus given a set of point correspondences between the left and right images, we can determine the depth map of the scene.



(a) left camera image I_L



(b) right camera image I_R



(c) right/left image overlay

(d) Ground truth disparity

Figure 5.1: Sample of a stereo image set, captured by two parallel cameras. This set called Tsubuka , is commonly used in literature as a reference set to compare disparity mapping algorithms[17]. (a) and (b) show the two camera images (c) shows an overlay of them. It is visible that close objects, like the lamp, are shifted horizontally by a bigger distance. (d) shows a ground truth disparity map, which indicates the true disparity of objects of the left camera image.

To compare the images, the two views must be transformed as if there were being observed from a common projective camera and the relative shifts between the two images can then be seen to be due to parallax, as long as the front face of the images to be compared is visible from this location, and that occlusion or transparency does not interfere with the calculation. This transformation is called rectification.

5.1.1 Basic Stereo System

Stereo algorithms intend to recover depth information by combining information from two stereo images. Most stereo algorithms are composed of the following steps:

1. Pre-processing: Removal of noise and image rectification i.e. the image is projected back to a common plane to allow comparison of the image pairs.
2. Stereo matching: Displacement of relative features is measured to calculate depth map.
3. Disparity refinement: Clustering of the depth map.

The overview of a basic stereo system is depicted in Figure 5.2.

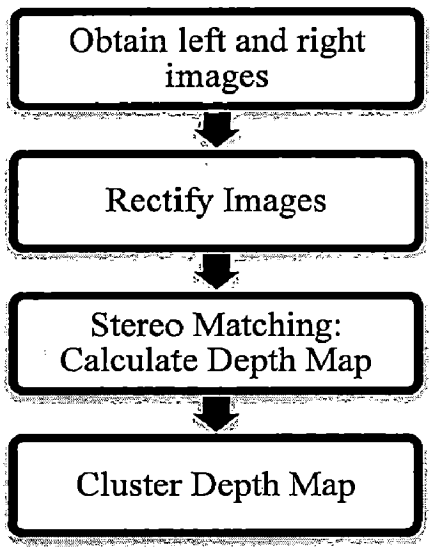


Figure 5.2: Overview of Stereo System

5.2 Epipolar Geometry

Epipolar geometry is a specific example of multiview geometry, which is the only available geometry constraint between a stereo pair of images of a single scene. It has been extensively studied in computer vision [18][19]. Let us consider a stereo imaging setup as shown in Figure 5.3. Let and be the optical centers of the first and second cameras and let the plane and be the first and second image planes. According to epipolar geometry, for a given image point in the second image, its corresponding point in the first image is constrained to lie on line . This line is called the epipolar line.

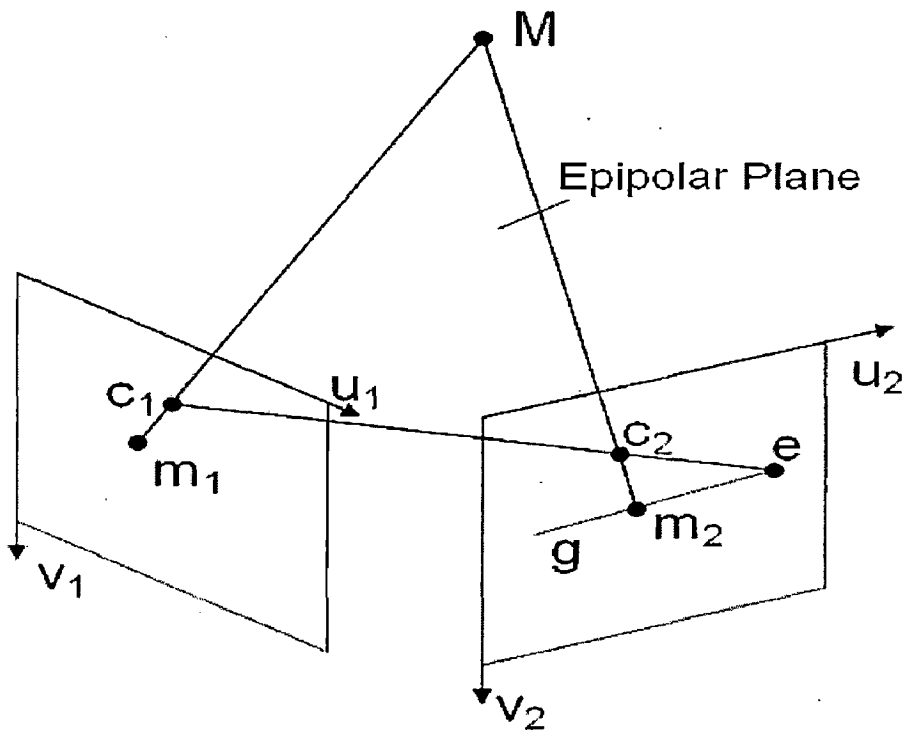


Figure 5.3: General epipolar geometry

With two cameras arranged arbitrarily, the general epipolar geometry is shown in Figure 5.4. The relative position of both cameras is known and $C1$ and $C2$ point out the optical centres of each camera. The straight line connecting both optical centres is called baseline. Each point M observed by the two cameras at the same time along with the two corresponding light rays through the optical centres $C1$ and $C2$ form an epipolar plane.

The epipole *e* is the intersection of the baseline with the image plane. The epipolar line is therefore defined as a straight line *g* through *e* and *m* that is the intersection of the line through *M* and the optical centre with the respective image plane. The point *M* in Figure 5.3 is projected as *m*1 in the left image plane. The corresponding point in the right image therefore lies on the previous described epipolar line *g*. This reduces the search space from two dimensional, which would be the whole image, to one dimensional, a straight line only.

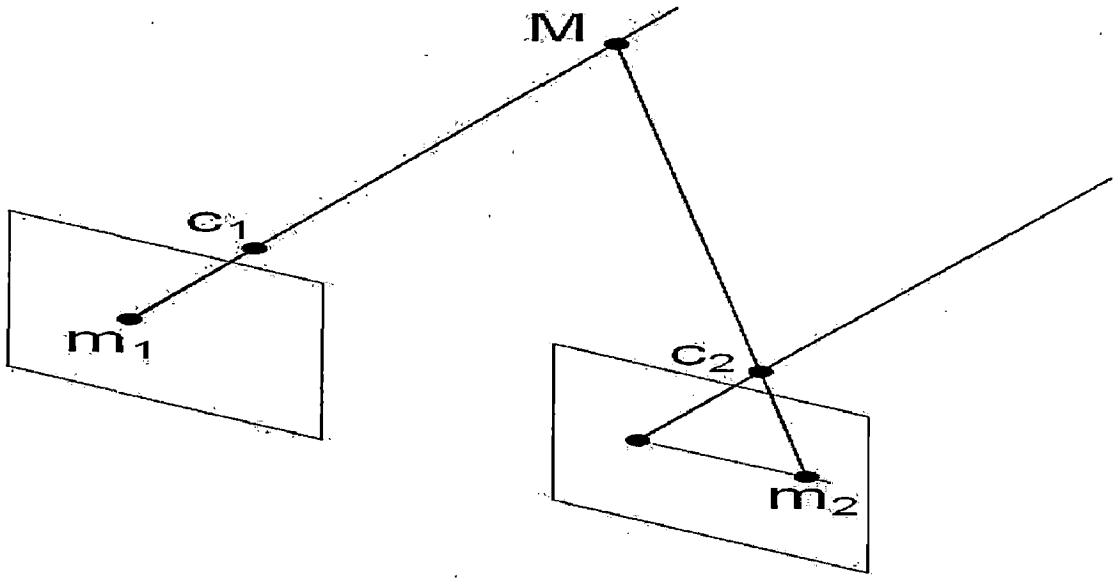


Figure 5.4: Stereo epipolar geometry

A simplification of the general epipolar geometry is shown in Figure 5.4. Both cameras are arranged in parallel, their focal length is identical and the two retinal planes are the same. Assuming these conditions all epipolar lines are horizontal within the retinal planes and the projected images *m*1 and *m*2 of a point *M* will have the same vertical coordinate. Therefore the corresponding point of *m*1 lies on the same horizontal line in the right image.

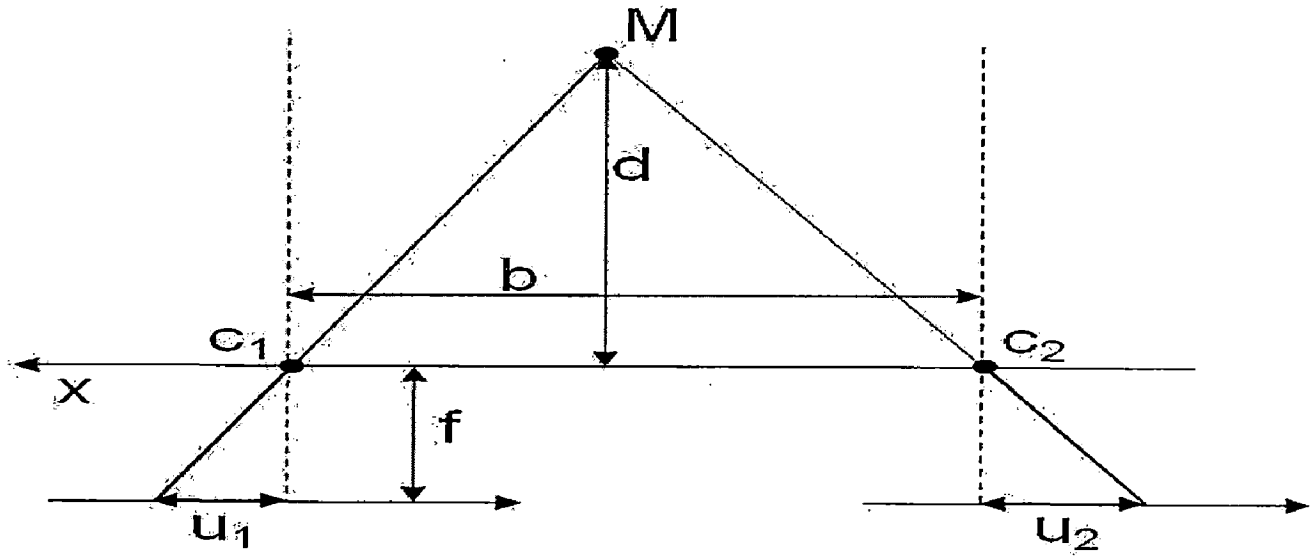


Figure 5.5: Disparity

According to the stereo epipolar geometry the disparity is defined as $D = c2 - c1$ as seen in Figure 5.5. The depth d therefore calculated by triangulation is

$$d = b \frac{f}{D} \tag{5.1}$$

where b is the distance of the two optical centres and f is the focal length. Therefore it can be seen the the depth is inversely related to disparity. Therefore if we know the disparity the corresponding depth information can be retrieved. A disparity of zero indicates that the depth of the appropriate point equals infinity. In order to assure that images follow stereo epipolar geometry the rectification of both images is necessary.

5.3 Stereo Matching

Stereo matching tries to solve the problem of finding which pixels or objects in one image correspond to a pixels or objects in the other. This is also known as the Correspondence Problem.

The existing techniques for general two-view stereo correspondence roughly fall into two categories: local method and global method [20]. Local methods use only small

areas/neighbourhoods surrounding the pixels, while global methods optimize some global (energy) function. Local methods, such as block matching [4], gradient-based optimization and feature matching [21] are very efficient, but they are sensitive to locally ambiguous regions in images (e.g., occlusion regions or regions with uniform texture). Global methods, such as dynamic programming [22], intrinsic curves and graph cuts can be less sensitive to these problems since global constraints provide additional support for regions difficult to match locally. However, these methods are more expensive in their computational cost.

The algorithms can roughly also be divided into *feature based* and *area based*, also known as region based or intensity based. Area based algorithms solve the correspondence problem for every single pixel in the image. Therefore they take colour values or intensities into account as well as a certain pixel neighbourhood. A block consisting of the middle pixel and its surrounding neighbours will then be matched to the best corresponding block in the second image. These algorithms result in dense depth maps as the depth is known for each pixel. But selecting the right block size is difficult because a small neighbourhood will lead to less correct maps but short run times whereas a large neighbourhood leads to more exact maps at the expense of long run times.

Feature based correspondence algorithms on the other hand extract features first and then try to detect these features in the second image. These features should be unique within the images, like edges, corners, geometric figures, whole objects or part of objects. The resulting maps will be less detailed as the depth is not calculated for every pixel. But since it is much more unlikely to match a feature incorrectly because of its detailed description, feature based algorithms are less error sensitive and result in very exact depth maps.

Besides area based and feature based correspondence algorithms, there are also *phase based* algorithms that transform the images using FFT (fast fourier transformation) first. The depth is therefore proportional to the phase displacement. *Wavelet based* algorithms are a subcategory of phase based algorithms and use a wavelet transformation first.

There are a number of problems all correspondence analysis algorithms have to deal with. An object seen by one of the cameras could be occluded when seen by the other camera that has a slightly different point of view. This object will cause wrong correspondences when trying to match images. The cameras itself my cause distorted images due to lens distortion which will lead to wrong correspondences especially in the outer regions of the image.

Some problems are caused by the objects themselves. Having lots of small objects that look alike or having a special pattern that iterates quite often makes it hard to find the matching object as there is more than one possible match. This is known as the aperture problem. Another big problem is homogeneity. Big homogeneous regions are difficult to match when seen through a small window only. The same textures on different positions in the image will cause similar problems.

5.2.2 Block Matching for Stereo Matching

The block matching method is one of the most popular local methods because of its simplicity in implementation. The basic idea of block matching for stereo correspondence is as follows: to estimate the disparity of a point in the left image, firstly, we define a reference block surrounding this point; and then, find the closest matched block, within a search range in the right image, using a pre-specified matching criterion; thus, the relative displacement between the reference block and the closest matched block constitutes the disparity of the point being evaluated. The commonly used matching criteria are the sum of absolute differences (*SAD*), the sum of squared (*SSD*) and the normalized *SSD*.

With the given matching criteria, the correspondence problem results in essentially a search problem, and the standard search method for block matching is an exhaustive search, where the matching criterion is calculated for all pixels at all possible search positions. This strategy can guarantee that the best-matched block is found with respect to the chosen criterion. However, the computation loads of such methods are very demanding, even by using the epiploar lines constraint, and therefore many different fast algorithms have been developed.

In order to obtain the disparity of a point, the candidate block in the search range that best matches the reference block is of the main interest. In this dissertation work we propose an improved parallel block matching method to efficiently solve the stereo correspondence problem using rectified stereo images that uses SAD as the matching measure and calculates the disparity map.

Chapter 6: Parallel Stereo Matching

6.1 Matching Algorithm Description

The fundamental problem for the disparity determination is the identification of the two corresponding or matching pixels that describe the same spot in the two images. Our approach belongs to the class of area-based matching algorithms, which are suitable for parallel implementation [22]. We choose stereo images that have been rectified on the basis of the epipolar constraint which leads to a one dimensional search space parallel to the horizontal image lines. This constraint is the key to the efficiency for the parallel implementation. Our implementation performs a correspondence search for each pixel in the right image block matching, thereby producing a disparity measurement.

First, given any two or more views of the same scene, at some image scale, a degree of similarity exists between the views, and in general, the coarser the scale the more similar the views become. These effects form the basis for matching area based stereo algorithms by the explanation that now follows. If a view is spatially quantised into smaller subregions, eventually any given subregion will begin to look more similar to its corresponding subregion in the other view. In this way, the similarity values are computed by comparing a fixed window in the reference image to a shifting window in the second image. The shifting window is moved over the first one by integer increments along the corresponding epipolar line and a curve of similarity values is generated for integer disparity values.

In Figure 6.1, I_r is the right Image and I_l is the left stereo image. We illustrate the Right to Left correspondence search for a pixel p_1 having pixel coordinates (x_1, y_1) in I_r comparing it to candidate pixels in I_l. The candidate pixels are all the pixels in I_l of coordinates $(x_1 + t; y_1)$, for all t, where $0 < t < d_{limit}$. The value d_{limit} represents a chosen limit to the search space size also represents the highest disparity that can be measured. The computational complexity is also directly related to d_{limit}. The comparison of p1 and a candidate pixel $p_{candidate}$ considers a window of size (2H+1)*(2W+1) around the pixels,

the so called blocks. The window around p_1 is called reference window, the overlapping windows around the candidate blocks constitute the search window. The block with the highest similarity to the reference window is chosen as the match guess and its offset d is the measured disparity.

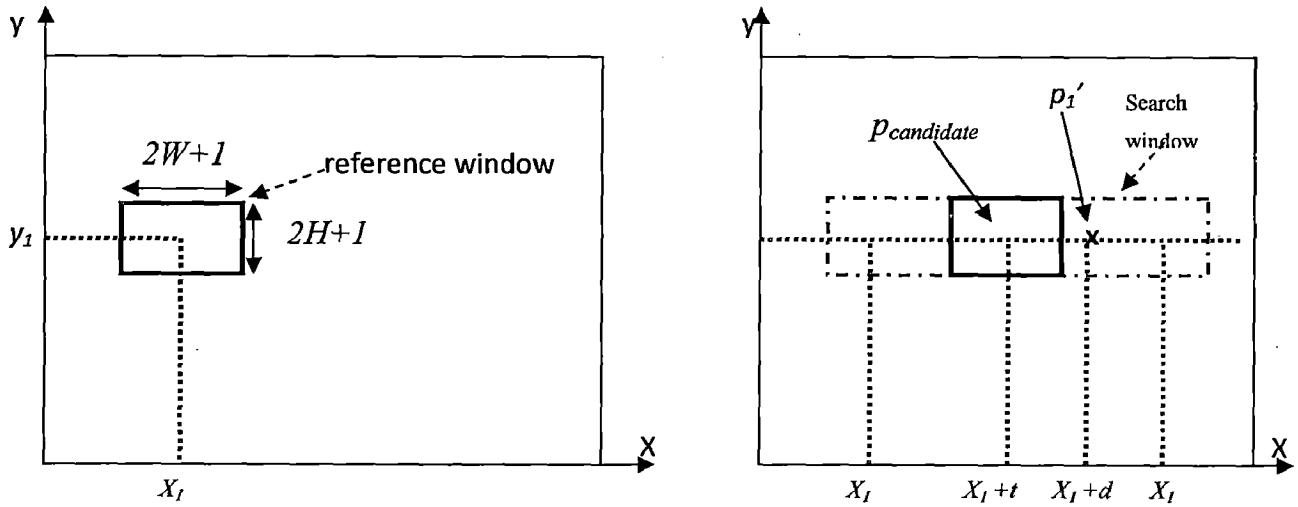


Figure 6.1: Reference and search window for disparity computation of pixel $p_1(x_1, y_1)$. The pixel p_0' represents the corresponding pixel for p_1, and is located at ($x_1 + d$, y1).

The sum of absolute differences (SAD) computation technique was chosen in our implementation due to its low implementation complexity. However we can also use different measures such as SSD, normalized correlation, SATD etc. using the same framework that has been explained in this section. The area-based algorithm using the SAD for calculating disparity can be resumed in equation 3:

$$\min_{d \in [0, d_{limit}]} \left\{ \sum_{i=-H}^{H} \sum_{j=-W}^{W} |I_R(x+i, y+j) - I_L(x+i+d, y+j)| \right\} \quad (6.1)$$

Where : I_R and I_L are the right and left image respectively,

x is an index on the columns,

y is an index on the rows,

d is the disparity index,

H and W define the size of the correlation window and

d_{limit} is the maximum value for disparity

If equation 1 is applied to each pixel in the images, it can be rewritten as:

$$min\left\{\begin{array}{l}\left\{\sum_{i=-H}^{H}\sum_{j=-W}^{W}|I_R(x+i,y+j)-I_L(x+i,y+j)|\right\}\\ \left\{\sum_{i=-H}^{H}\sum_{j=-W}^{W}|I_R(x+i,y+j)-I_L(x+i+1,y+j)|\right\}\\ \vdots\\ \left\{\sum_{i=-H}^{H}\sum_{j=-W}^{W}|I_R(x+i,y+j)-I_L(x+i+d_{limit},y+j)|\right\}\end{array}\right\} \quad (6.2)$$

If for each disparity index we allocate a thread to compute the window, then equation 2 suggests an implementation. However, the principal inconvenient for this implementation is that it would incur a large amount of overhead as well as memory to store the windows until the minimisation operation takes place. To solve this problem, it was noticed that the calculation of windows is a recursive computation, where adjacent pixels in overlapping windows are present, so it is not necessary to compute the complete similarity value for a pixel if the adjacent pixel have one already computed.

6.2 Computational optimisation

The most expensive task performed by the stereo algorithm is the computation of SAD scores, which are needed to carry out the direct matching phase. In this section we outline the optimisation techniques adopted to avoid redundant calculations. We show first the basic calculation scheme, already described in [23], and then propose an additional level of incremental calculation aimed at achieving further speed-up.

Suppose that SAD(x, y, d) is the SAD score between a window of size (2n+1)*(2n+1) centered at coordinates (x,y) in the left image and the corresponding window centered at (x+d,y) in the right image:

$$SAD(x,y,d) = \left\{ \sum_{i=-n}^{n} \sum_{j=-n}^{n} |I_R(x+j, y+i) - I_L(x+d+j, y+i)| \right\} \quad (6.3)$$

Observing Figure 6.2 [23], it is easy to notice that SAD(x, y +1) can be attained from SAD(x,y,d) :

$$SAD(x, y+1, d) = SAD(x, y, d) + U(x, y+1, d) \quad (6.4)$$

with U(x,y+1,d) representing the difference between the SADs associated with lowermost and uppermost rows of the matching window (shown in light-gray in Figure 6.2):

$$U(x, y+1, d) = \left\{ \sum_{j=-n}^{n} |I_R(x+j, y+n+1) - I_L(x+d+j, y+n+1)| - \sum_{j=-n}^{n} |I_R(x+j, y-n) - I_L(x+d+j, y-n)| \right\} \quad (6.5)$$

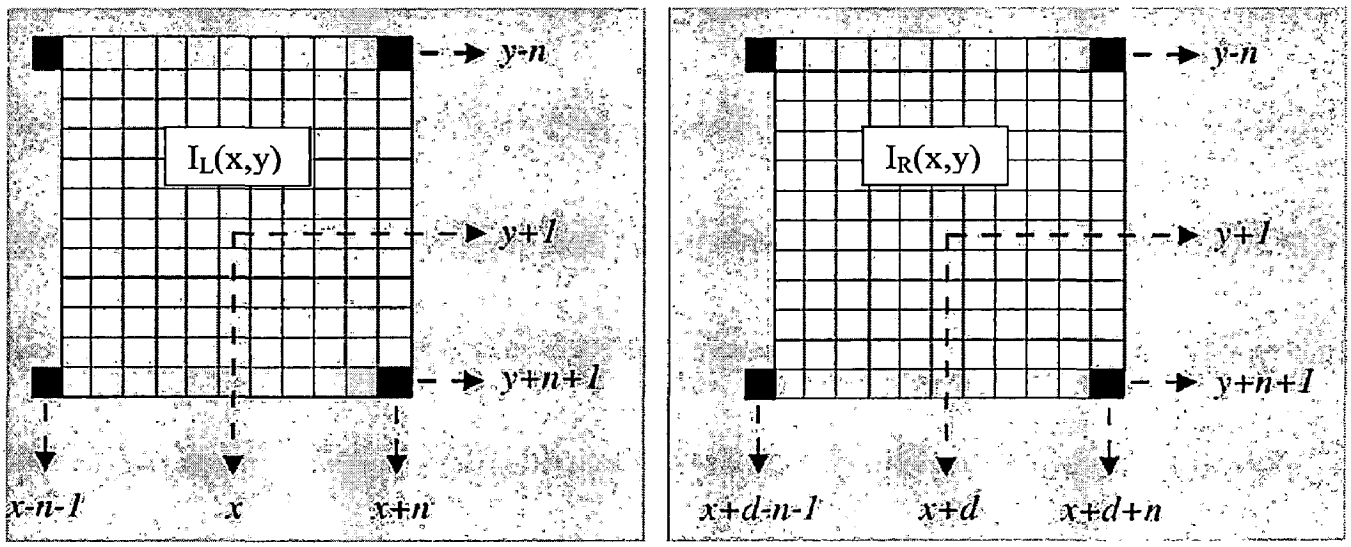


Figure 6.2: Incremental calculation scheme.

Thus it can be observed from the Equation 6.5 that we need not calculate the SAD for each and every pixel but it can be calculated in an incremental manner. We utilize the above equation and modify it further for implementation on multi-core processor which will be discussed in the next section.

At this point, the main problem with the SAD matching technique is the size of the window, which needs to be large enough to include enough variations of intensity to make the matching, but small enough to prevent projective distortion. If the window is too small and it does not cover sufficient variation of intensity, it gives a poor estimation because its SNR is low. On the other hand, if the window is too large and covers a region in which the depth of the points of the scene varies then the position of the SAD minimum cannot present correct coincidence due to different projective distortions between the left and right images. In the literature, a recommend window size of 7×7 for real time applications [19].

6.3 Strategy for parallel implementation on Multi-core

Even for the simple block-matching algorithm described above there are many different approaches to implementation on the Multi-cores. The strategy employed in this example is highly optimized, but certainly not guaranteed to be the only fast approach.

The primary goal of this implementation is to be fast, with a secondary goal of being fairly flexible to allow changing of parameters. Some critical guidelines for optimizing this application that we have followed are avoiding obscenely redundant computation – Many computations preformed for one pixel can also be used by neighbours. We have also tried to minimize global memory reads/writes and create enough threads to keep the processors busy.

Based on the discussion in section 6.2, we have implemented a disparity computing system as shown in Figure 6.3. Here the first module performs similarity measurement

between pixels from the reference image and the pixels from the shifted image and then sums it to obtain column-wise SADs. The second module obtains the sum of the results from the column submodule to form the block-wise SAD. Finally, the minimum of the block-wise SADs is chosen to obtain the disparity. Both the above modules are computed in parallel.

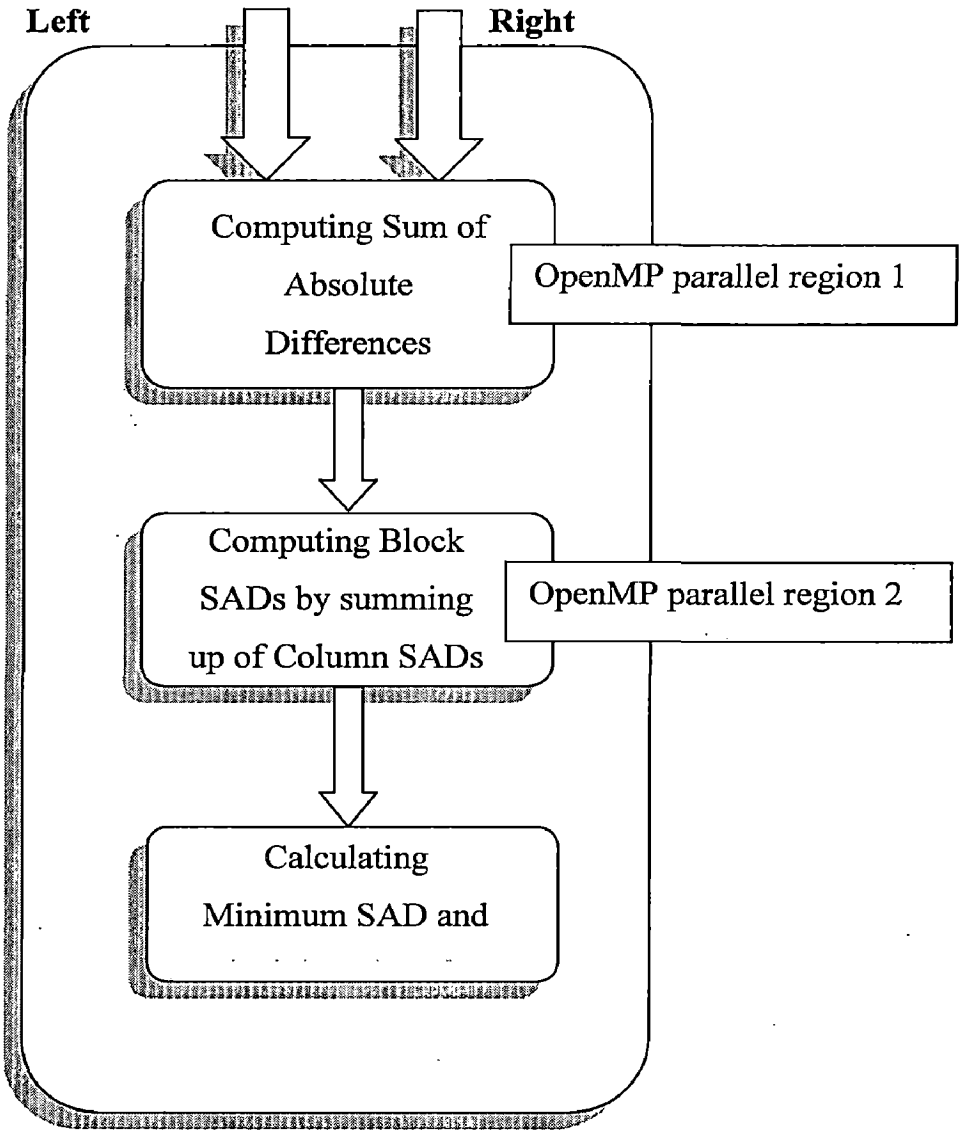


Figure 6.3: Implemented Stereo Matching System

The next step is to determine how to allocate threads to the problem. Our approach uses a thread to process a column of pixels. This is a departure from the common approach of using a thread per pixel but this allows us to eliminate some redundant computation and

reduces threading overheads. The thread per pixel approach is more suited to GPU implementations.

In Figure 6.4 we illustrate the overall scheme using a 7x7 block size and 16 threads. Each thread sums the absolute differences of a column of pixels the height of the kernel. It accumulates absolute differences between the pixels in the reference image (left mage) and the comparison image (right image).

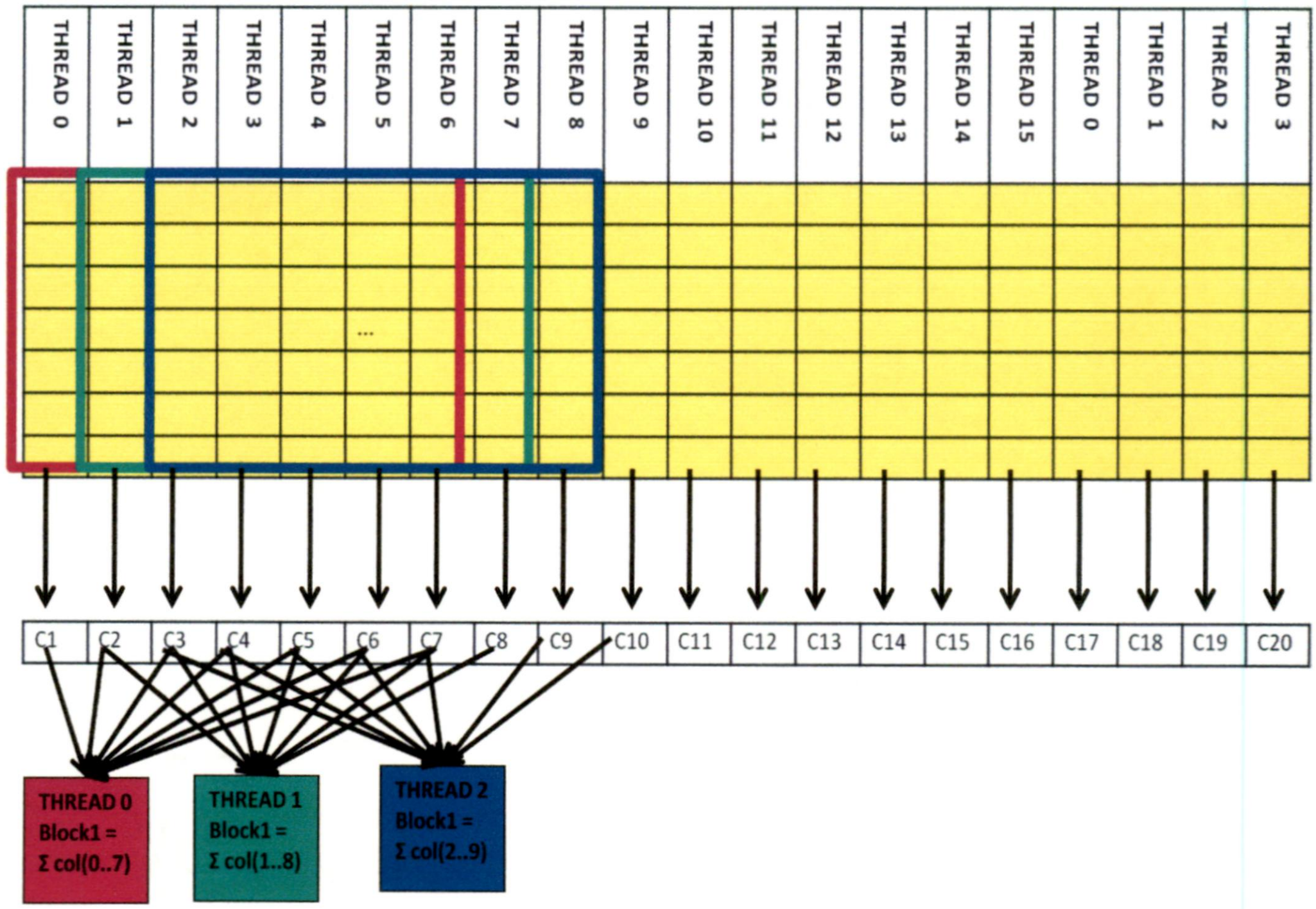


Figure 6.4: Allocation of threads

The sum of absolute differences for each column is stored in shared memory such that it can be accessed by multiple threads at a time. After the column sum-of-absolute differences are completed for the block of threads then each thread sums the column SAD values from the neighbouring columns within the block to determine the total SAD for

the entire block. This value is tested to determine if the current disparity value is the best correspondence match for the present pixel. Three pixels have been highlighted for illustration in red, green and blue with the associated kernel pixels outlined in a line of the same colour.

After the first row of pixels has been processed by a thread block subsequent rows can be processed with increased efficiency. A rolling window scheme is used. Rather than repeat the summation of all the absolute differences in the column, the absolute difference of the pixels in the first row is subtracted from the previous column sum, and then the absolute difference of pixels in a new row is added to the column sum. This value is equivalent to re-summing the rows involved, but requires only two absolute difference computations and two additions. This rolling window computation continues until SAD is determined for all the rows allocated to a thread.

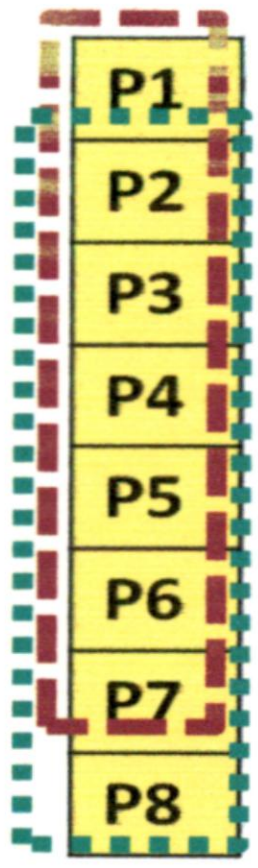


Figure 6.5: Rolling window scheme for calculation of column SAD

This can be illustrated as follows from Figure 6.5. Let the SAD for the red column block be SAD_{red} and for the green column block be SAD_{green}. So once we have SAD_{red} we can calculate SAD_{green} as follows:

$$SAD_{green} = SAD_{red} - SAD_{p1} + SAD_{p8}$$

where SAD_{p1} and SAD_{p8} are the SADs for the pixels p1 and p8. We already have the previous column (red) SADs stored in memory. Every following sum can therefore be calculated from the previous one with one addition and one subtraction and then stored. This way a lot of computation is saved. Also since the array of intermediate column SADs are stored in shared memory this operation can also be performed

At each disparity step, the SAD value determined for the kernel is compared with the current minimum SAD value computed from prior disparity steps. The current minimum SAD result and the corresponding *d* value are stored in memory in arrays the size of the image. If the newly computed SAD value is less than the previous minimum SAD value, this new value becomes the minimum and is stored along with the corresponding *d* value in memory. At the end of this process a disparity value indicative of the best correspondence has been computed and stored in global memory.

6.4 Performance Analysis

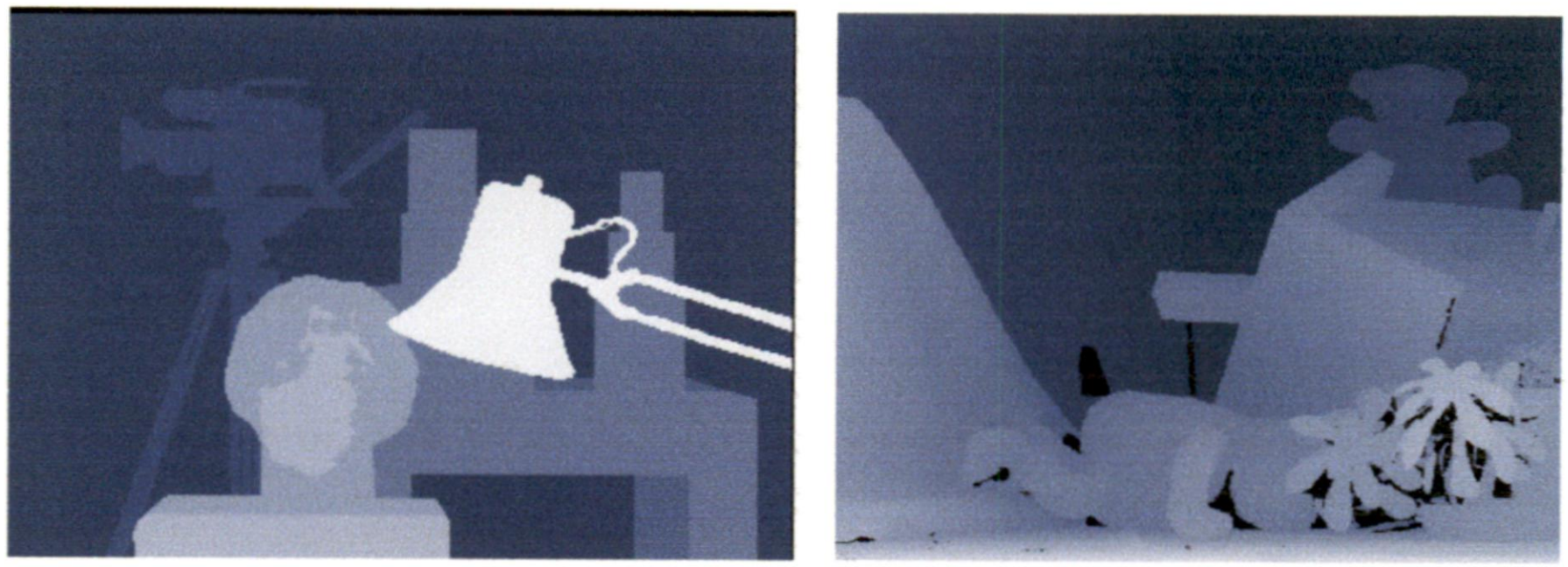
We conducted experiments to compare the performance of the sequential implementation and the OpenMP based parallel implementation. The tests have been performed on a Dell Precision T7400 workstation with an Intel Dual Socket Quad-core with 256 KB of L1 cache (32KB x 8 cores) and 12 MB of L2 cache and 8 GB of RAM. The maximum number of processors used for the experiments was eight and the memory was shared.

All the tests have been performed using Linux Fedora Core 11 operating system. The program to perform motion estimation is written in C programming language. The C compiler used is GCC version 4.4.0 and the OpenMP package used is OpenMP 2.0.

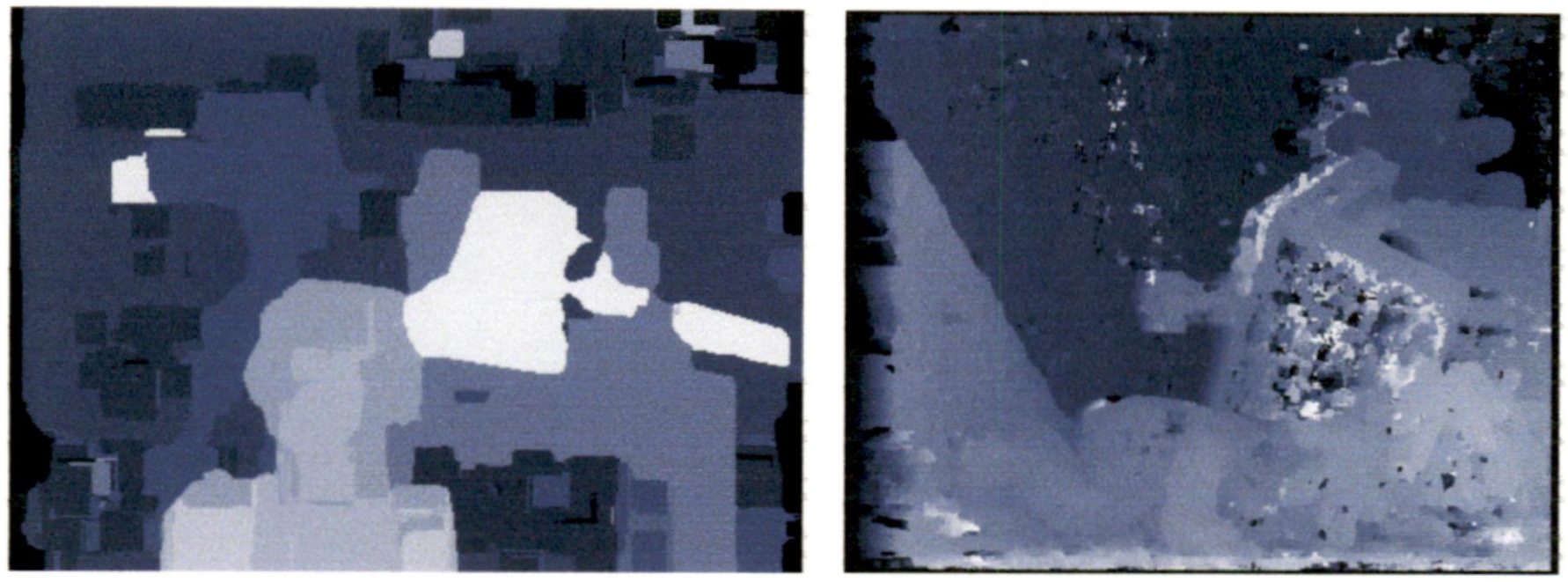
The stereo image pairs for the experiments were taken from the middlebury university vision database [17]. The image pair used was tsukuba and teddy (Figure 6.5(a)) of size 384 x 288 pixels. The maximum number of disparity levels was taken as 16 for the tsukuba image and as 32 for the teddy image. Figure 6.6 shows the performance for both

the serial and parallel implementations. The benefit of using the parallel version can be seen from the figure as it reduces the time spent by almost 5 times.

(a)



(b)



(c)

Figure 6.5: Disparity Map (a) Test Image (b) Ground Truth (c) Obtained disparity Map

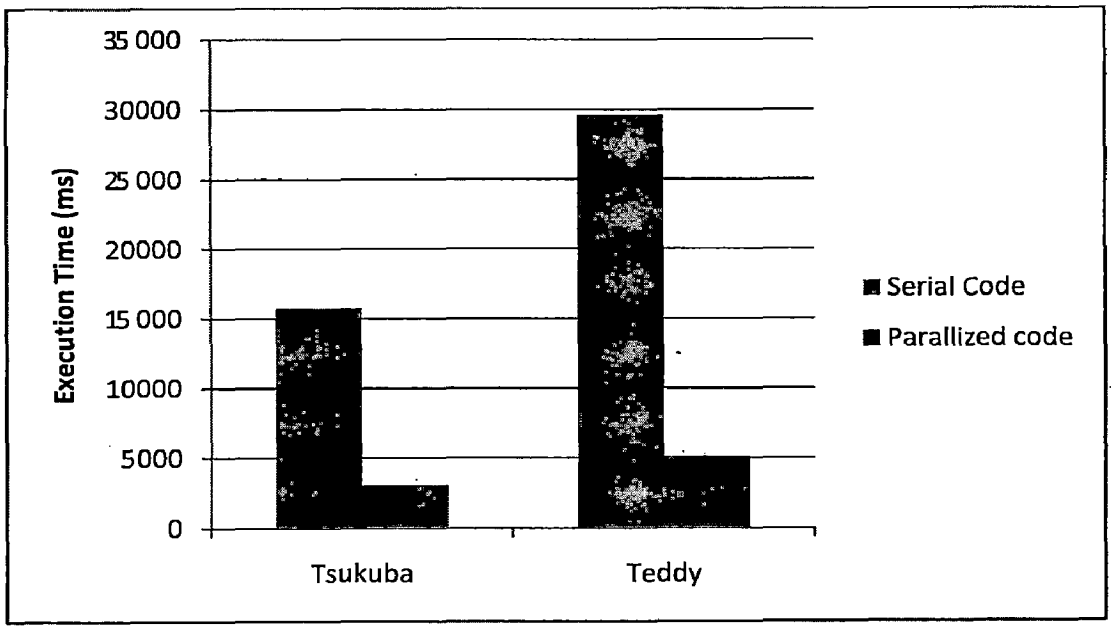


Figure 6.6: Comparison of Execution times of Serial and Parallel implementations

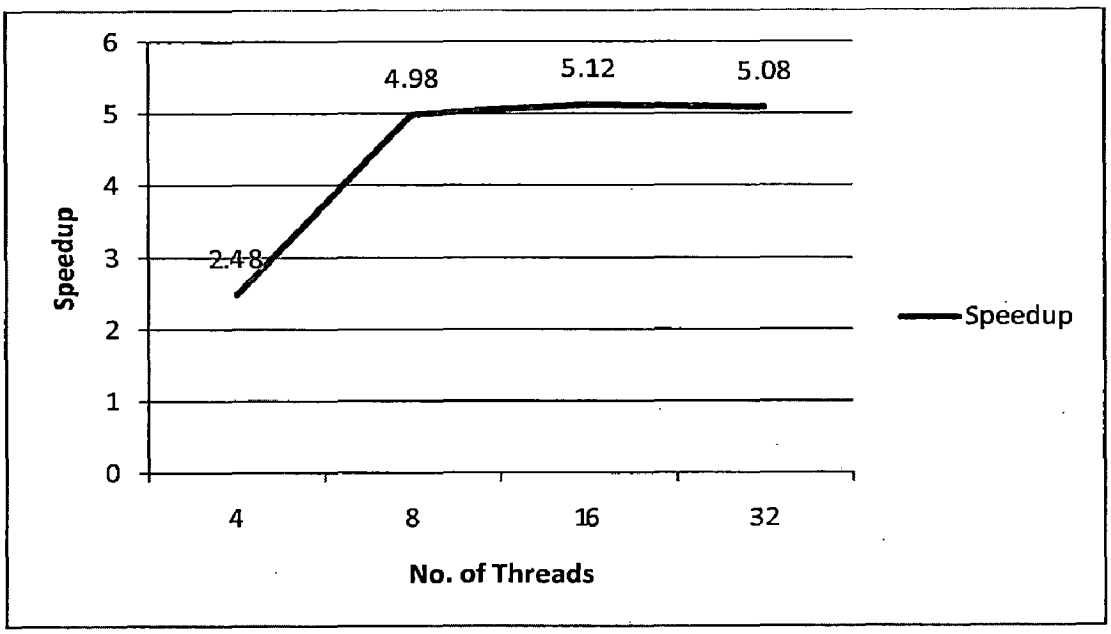


Figure 6.7: Speedup obtained versus No. of threads

The performance scales approximately the same with the increase number of processors used. The performance almost doubles when we use all 8 cores as can be seen from Figure 6.7, as when the number of threads is 4, the number of processors used is also 4 while for 8, 16 and 32 threads all the 8 processing cores are used. One more inference

from Figure 6.6 is that when we increase the number of disparity levels from 16 to 32 for the teddy image, the time taken does not exactly double but is around 1.7 times. This can be attributed to considerable savings due to the use of the rolling window scheme.

The vision research centre of Middlebury University has an online evaluation website [24] that can be used to evaluate results and find the percentage of bad pixels. We used the evaluation mechanism for evaluating our results (Figure 6.5) i.e. the comparison of the obtained depth map with the ground truth thereby determining the accuracy. The percentage of bad pixels in the obtained map is approximately 14% for tsukuba image and 31% for the teddy image. Though this percentage is normal for most area based implementations, however other phase based and feature based methods have known to give percentage of bad pixels as less as 5%. However such implementations are extremely compute intensive.

Conclusion

With the emergence of parallel GPUs and multi-core architectures, a massive amount of computing power is available that was exploited in this work to satisfy the increasing computation needs of image and video processing algorithms. This dissertation presents efficient multithreaded implementations of block matching algorithm which forms a backbone of various such algorithms.

Efficient strategies were described for implementation of block matching based algorithm for motion estimation on Intel multi-core architectures using OpenMP compiler giving 7 times speedup. A simple yet elegant GPU implementation of motion estimation has also been performed which gives approximately 9 times speedup.

A parallel strategy for implementation of an algorithm using block matching for stereo matching has also been shown. The implementation written in C and OpenMP, and performed on a dual socket quad-core Intel Xeon Windows machine shows that using shared memory the algorithms run time is reduced by approximately 5 times. The use of shared memory allows maximum benefit of the multi-core processor since there is no overhead incurred for splitting the data between the different cores.

Our multithreaded implementation based on OpenMP and CUDA programming model also demonstrates the inherent parallelism of image processing algorithms and the parallel computation capabilities of symmetric multiprocessor architectures can be efficiently exploited in the future for real time applications currently possible only on dedicated hardware.

Our focus in the thesis was only on the motion estimation part of video encoding. In the future implementation of complete video encoding algorithms on GPU and multi-cores can be explored, expecting to get higher performance and approaching the real-time processing speeds.

We have used the SAD criteria as the similarity measure for all implementations. In the future, the same framework may be implemented using other correlation criteria instead of the sum of absolute differences like SATD, SSD, Normalized Correlation etc., to minimise noise and/or explore different performance for a specific application. Of future interest also would be to investigate the implications of parallel implementation of the algorithms using MPI and no shared memory.

References

[1] T. Wiegand, "Joint final committee draft for joint video specification H.264," Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, Tech. Rep. JVT-D157, 2002 .

[2] L. Yu, F. Yi, J. Dong, and C. Zhang, "Overview of AVS-Video: tools, performance and complexity," *in Proc. Visual Communications and Image Processing* (VCIP), 2005.

[3] Y. Wang, J. Ostermann, and Y. Zhang, Video Processing and Communications, Prentice Hall, 2002.

[4] B.D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," Proc. Int'l Joint Conf. Artificial Intelligence, pp. 674-679, 1981.

[5] Y. Jehng, L. Chen, and T. Chiueh, "An efficient and simple VLSI tree architecture for motion estimation algorithms," IEEE Trans. Signal Processing, vol. 41, no. 2, February 1993, pp. 889–900.

[6] H. Yeo and Y. H. Hu, "A novel modular systolic array architecture for full-search block matching motion estimation," IEEE Trans. Circuits Syst. Video Technol., vol. 5, no. 5,, Oct. 1995 pp. 407–416.

[7] A. Estrada, J. M. Xicotencatl, "Multiple Stereo Matching Using an Extended Architecture," *in Proc. Field Programmable logic and Application,* Springer-Verlag, vol. 2147, Aug. 2001, pp. 203-212.

[8] R. Chandra, Parallel Programming in OpenMP, Morgan Kaufmann, 2001

[9] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, E. Su, "Intel OpenMP C++/FORTRAN Compiler for Hyper-Threading Technology: Implementation and Performance," Intel Technology Journal, Vol. 6, Q1,2002

[10] H. Blume, J. Livonius, L. Rotenberg, T. Noll, H. Bothe, J. Brakensiek, "OpenMP-based parallelization on an MPCore multiprocessor platform - A performance and power analysis", Journal of Systems Architecture, Volume 54, Issue 11, November 2008, pp 1019-1029.

[11] NVIDIA Corporation: NVIDIA CUDA compute unified device architecture programming guide, Version 2.0, NVIDIA Corporation, July 2008.

[12] International Standard Organization, "Information Technology-Coding of Audio-Visual Objects,Part2 --Visual," ISO/IEC 14496-2.

[13] H.-C. Chang, L.-G. Chen, M.-Y Hsu, and Y-C. Chang, "Performance analysis and architecture evaluation of mpeg-4 video codec system," in Proc. IEEE International Symposium on Circuits and Systems, May 2000, pp. 449-452

[14] K. Shring, "H.264/AVC Software Coordination". http://iphome.hhi.de/suehring/tml

[15] G. Shen, G. P. Gao, S. Li, H. Y. Shum, and Y-Q. Zhang, "Accelerate video decoding with generic GPU," IEEE Trans. Circuits Syst. Video Technology, vol. 15, no. 5, May 2005, pp. 685-69

[16] W. Chen and H. Hang, "H.264/AVC motion estimation implementation on compute unified device architecture (CUDA)", Multimedia and Expo, 2008 IEEE International Conference on, April 2008, pp 697–700.

[17] "Middlebury Stereo Datasets" http://vision.middlebury.edu/stereo/data/, Last Accessed 02nd June 2010

[18] R. Hartley and A. Zisserman, Multiple View Geometry in Computer Vision. Cambridge, U.K.: Cambridge Univ. Press, 2000.

[19] D. V. Papadimitriou and T. J. Dennis,"Epipolar line estimation and rectification for stereo image pairs," IEEE Trans. Image Processing, vol.5, no.4, 1996, pp.672–676.

[20] M. Z. Brown, D. Burschka, and G. D. Hager, "Advances in computational stereo," IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 25, no. 8, 2003, pp.993–1001.

[21] C. Schmid and A. Zisserman, "The geometry and matching of curves in multiple views," Proc. European Conf. Computer Vision, pp. 104-118, 1998.

[22] O. Faugeras, B. Hotz, H. Matthieu, T. Vieville, Z. Zhang, P. Fua, E.Theron, L. Moll, G. Berry, J. Vuillemin, P. Bertin, and C. Proy, "Real time correlation-based stereo: algorithm, implementations and applications," INRIA Technical Report 2013, 1993.

[23] L. Di Stefano, M. Marchionni, and S. Mattoccia. A fast area-based stereo matching algorithm. Image and Vision Computing, 22(12):983–1005, Oct 2004.

[24] "Middlebury Stereo Evaluation – Version 2"
http://vision.middlebury.edu/stereo/eval/, Last Accessed 02nd June 2010

List of Publications

- Abed Mohammad Kamaluddin, Dr Kuldip Singh and Dr Ankush Mittal, "Parallelization of Multipass Algorithm For Motion Estimation on Multicore CPUs," In Proceedings of IEEE International Conference on Advances in Communication, Network and Computing, Calicut, India, 04-05 Oct 2010 [Accepted for Publication]