Implementation of Block Matching Algorithms on CUDA

By

Chhaya Patel

10MCEC29

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

May 2012

Implementation of Block Matching Algorithms on CUDA

Major Project

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science and Engineering

PREPARED BY :

Chhaya Patel

10MCEC29

GUIDED BY :

Prof. Samir Patel

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD

DECLARATION

I, **Chhaya C. Patel**, **10MCEC29**, give undertaking that the Major Project entitled **"Implementation of Block Matching Algorithms on CUDA"** submitted by me, towards the partial fulfillment of the requirements for the degree of Master of Technology in Institute of Technology of Nirma University, Ahmedabad, is the original work carried out by me and I give assurance that no attempt of plagiarism has been made. I understand that in the event of any similarity found subsequently with any published work or any dissertation work elsewhere; it will result in severe disciplinary action.

Chhaya Patel

DEDICATION

This thesis is dedicated to my father, who taught me that even the largest task can be accomplished if it is done one step at a time. It is also dedicated to my husband, who taught me that the best kind of knowledge to have is that which is learned for its own sake.

I would also like to dedicate this thesis to my family, who passed on a love of reading and respect for education, who have never failed to give me moral support, for giving all my need during the time I developed system.

CERTIFICATE

This is to certify that the Major Project, entitled **Implementation of Block Matching algorithms on CUDA**, submitted by Ms. Chhaya C. Patel [10MCEC29], towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by her under my supervision and guidance. In many opinion, submitted work has reached a level required for being accepted for examination. The result embodied in this major project, to the best of my knowledge,haven't been submitted to any other university or institution for award of any master degree.

Prof. Samir Patel
Guide and Professor,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr. S.N.Pradhan
Professor,P.G. Coordinator,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Prof. D. J. Patel
Professor and Head,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr. Ketan Kotecha
Director,
Institute of Technology,
Nirma University, Ahmedabad

ABSTRACT

With the increasing popularity of technologies such as Internet streaming video and video conferenting, video compression has became an essential component of broadcast and entertainment media. Motion Estimation (ME) and compensation techniques, which can eliminate temporal redundancy between adjacent frames effectively, have been widely applied to popular video compression coding standards such as MPEG4 AVC/H.264. The use of a high-level parallelism of GPU architecture for the high definition images and video processing give high performance and high speed of data processing. Due to its object based nature, MPEG4 AVC is good candidate for parallelism. Among all steps of encoder, motion estimation is very time consuming step. The wide range of memories offered by the GPU architecture has been exploited to a large extent and a significant speedup of motion estimation on the GPU as compared to the serial implementation. The use of texture memory for accessing the periphery of the frames prevents the out of bound references and contributes to the speedup achieved. The use of shared memory for processing current frame and reference frame exploit significant speed for temporal redundancy. The parallel versions of Full search and Diamond search are implemented on Tesla C2070 on Ubuntu OS using CUDA4.0 technology. The results are presented along with methodology used for parallelization of code for NVIDIA GPU. We also obtain increased performance by reordering accesses to off-chip global memory to combine requests to the same or contiguous memory locations and apply classical optimizations to reduce the number of executed operations.

ACKNOWLEDGEMENT

It gives me pleasure in expressing thanks and profound gratitude to **Prof. Samir Patel**, Department of Computer Science and Engineering, Institute of Technology, Nirma University, Ahmedabad for his valuable guidance and continual encouragement throughout the Major Project. I heartily thankful to him for his time to time suggestion and clarity of the concepts of the topic that helped me a lot during my project work.

I like to give my special thanks to **Dr. S. N. Pradhan**, P.G. Coordinator, Department of Computer Science and Engineering, Institute of Technology, Nirma University, Ahmedabad for his continual kind words of encouragement and motivation throughout the project.

I would like to thanks **Dr. Ketan Kotecha**, Hon'ble Director, Institute of Technology, Nirma University, Ahmedabad for his unmentionable support, providing basic infrastructure and healthy research environment. I would also thank my Institution, all my faculty members and my colleagues without whom this project would have been a distant reality.

The blessing of God and my family members makes the way for completion of the project. I am very much grateful to them. The friends, who always bear and motivate me throughout the project,I am thankful to them.

Chhaya Patel
10MCEC29

Contents

List of Tables

List of Figures

Abbreviation Notation and Nomenclature

API ..... Application Program Interface
ALU ..... Arithmetic Logic Units
BMA ..... Block Matching Algorithm
BMME ..... Block Matching Motion Estimation
CPU ..... Central Processing Unit
CUDA ..... Compute Unified Device Architecture
CUDA FFT ..... CUDA Fast Fourier Transform
CUBLAS ..... CUDA Basic Linear Algebra Subroutine
CUP ..... Clean-Up Pass
DCT ..... Discrete cosine transform
DS ..... Diamond Search Block Matching Algorithm
DRAM ..... Dynamic RAM
DVD ..... Digital Versatile Disc
ES ..... Exhaustive Search
4SS ..... Four Step Search
FS ..... Full Search Block Matching Algorithm
GCC ..... GNU Compiler Collection
GFLOPS ..... Million Floating Point Instructions Per Second
GPGPU ..... General Purpose Graphics Processing Units
GPU ..... Graphics Processing Unit
IEEE ..... Institute of Electrical and Electronics Engineers
LDSP ..... Large Diamond Search Pattern
MIMD ..... Multiple Instruction, Multiple Data
MSE ..... Mean of Squared Error
MPEG ..... Motion Picture Experts Group
NVCC ..... CUDA Compiler
PSNR ..... Peak Signal to Noise Ratio
PTX ..... Parallel Thread Execution
MP ..... Multiprocessors
SAD ..... Sum of Absolute Difference
SDK ..... Software Development Kit
SDSP ..... Small Diamond Search Pattern
SP ..... stream processors
SIMD ..... Single Instruction, Multiple Data

SIMT ..... Single Instruction, Multiple Threads
3SS ..... Three Step Search

Chapter 1

Introduction

1.1 General

H.264 (MPEG-4 Part 10/MPEG-4 AVC) is a standard for video compression, and is currently one of the most commonly used formats for the recording, compression, and distribution of high definition video. However, a lot more computation power is required to encode the video, and it often takes hours to encode DVD-quality video clips in H.264 even on a high-performance desktop machine [1]. From all steps of video encoding, the motion estimation function, which is known to be the most computationally intensive section of the code. A motion estimation algorithm exploits redundancy between frames, which is called temporal redundancy. A video frame is divide into macroblocks,each macroblocks movement from a previous frame (reference frame) is 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 [2]. Motion estimation require more than 75% of encoding time to reduce temporal redundancy [3].

In the past few years new heterogeneous architectures have been introduced in high-performance parallel computing. Examples of such architectures include Graphics Processing Units(GPUs). GPUs are small devices with hundreds of computing cores which are designed for high performance computing. GPU computing is the use of a GPU (graphics processing unit) as a co-processor to accelerate CPUs for general purpose scientific and engineering computing. The GPU accelerates applications running on the CPU by offloading some of the compute-intensive and time consuming portions of the code. The rest of the application still runs on the CPU. From a users perspective, the application runs faster because it is using the massively parallel processing power of the GPU to boost performance. This is known as heterogeneous or hybrid computing [4].

The Compute Unified Device Architecture(CUDA) parallel computing platform provides a set of abstractions that enable expressing fine-grained and coarse-grained data and task parallelism into thousands of simultaneous threads. CUDA is a general purpose parallel computing architecture with a new parallel programming model and instruction set architecture that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C as a high-level programming language. CUDA core are three key abstractions a hierarchy of thread groups, shared memories, and barrier synchronization that are simply exposed minimal set of language extensions [5].

At this point, this thesis presents an implementation of a part of the motion estimation algorithm in a GPU as a coprocessor to assist the Central Processing Unit (CPU). Our Block Matching algorithms are optimized for CUDA by using a hundred number of threads that can execute on parallel in GPU and can make an efficient use of the shared memory to reduce global memory access.

1.2 Motivation of Project

As multicore processors become more pervasive in electronics products - PCs, mobile phones, gaming systems, network equipments and now even medical devices - the demands of developing software for these systems has become a primary concerns in the electronics industry. The advantages of multicore architectures are many, such as higher performance, lower power consumption lower cost and more flexibility but can be realized only if the corresponding software is developed to unlock these benefits. Many software developers in today's electronics industry lack the skills to write software optimized for multicore. Furthermore multicore architectures can become exponentially complex as the number of cores increases, that is traditional means of development no longer scale. The motivation behind doing this research is the need to develop a "Block matching algorithms for motion estimation of video compression technique MPEG-4AVC/H.264" a more efficient and optimized way for multicore architecture using CUDA, which best utilizes the available core and other resources on GPUs.

1.3 Objective of Study

The objective of the dissertation work can be summarized as:

- Study and configure CUDA for NVIDIA GPUs for the project work. Study the architecture, programming model, memory model, execution model etc. of NVIDIA GPUs. Next objective is to study CUDA, a new programming language used on GPUs. We need to study basic APIs of CUDA as well as advance API used to optimize the performance of any algorithm.

- The next objective is to study different block matching algorithms and implement same on NVIDIA GPUs using CUDA. We need to alter the algorithm so that it can be run in parallel. Next step is to choose the best API for our algorithm so that algorithm can run in minimum time, and best utilize the available resources.

- Finally, the objective is to measure the time required for algorithms on CUDA and compare the time with other C implementation, while preserving the video/image quality, and estimation basic feature.

1.4 Scope of Work

The experimental setup, prepared for the dissertation work, includes installation of CUDA on NVIDIA GPUs, on Ubuntu 11.10. A preliminary investigation includes the study of GPUs, programming language CUDA, and CUDA APIs, which can be used for the implementation and optimization of any algorithm. In this project we focus on the task of parallelization of the algorithms rather than spending time on their implementation. For that we need to manage the resources, the resources include the number of registers and the amount of on-chip memory used per thread, number of threads per multiprocessor, and global memory bandwidth. Out of the various estimation algorithms, the work focuses on the detailed implementation of block matching algorithms of compression technique on CUDA, and execution time comparison done with other C implementations.

1.5 Thesis Organization

The rest of the thesis is organized as follows:
Chapter 2, Literature survey on CUDA describes history of CUDA. It also describes CUDA Programming Model, Memory Architecture, Hardware implementation, CUDA Occupancy Calculator, CUDA Visual Profiler,Performance Optimization Strategies, CUDA Installation, Related Work etc...
Chapter 3, Motion Estimation of Video Compression describes overview of Video Compression, Motion Estimation, Criteria for Motion Estimation, and Block Matching algorithms.

Chapter 4, Implementation, describe CUDA implementation of Full search block matching algorithm and Diamond search block matching algorithm.
Chapter 5, Implementation Results includes speedup rate analysis of Block Matching algorithms on CUDA, and Compare the result with existing implementation on C. Implementation Results also includes consistency checking for compressed video frames.
Chapter 6, Conclusion & Future Scope describes concluding remarks and scope for future is presented.

Chapter 2

Literature Survey

2.1 Introduction to NVIDIA CUDA

NVIDIA introduced CUDA (Compute Unified Device Architecture), a general purpose parallel computing architecture [5] with a new parallel programming model and instruction set architecture - that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. NVIDIA GPUs with the new Tesla unified graphics and computing architecture run CUDA C programs and are widely available in laptops, PCs, workstations, and servers. The CUDA model is also applicable to other shared-memory parallel processing architectures, including multicore CPUs [6].

NVIDIA CUDA SDK has been designed for running parallel computations on the device hardware: it consist of a compiler, host and device runtime libraries and a driver API. CUDA software stack is composed of several layers: a hardware driver (CUDA Driver), an API and its runtime (CUDA Runtime), two higher-level mathematical libraries (CUDA Libraries) of common usage as shown in figure 2.1.

GPU performance is influenced by the architectural organization of the hardware platform. NVIDIA suggests that achieving the highest GPU occupancy and optimizing the use of the memory hierarchy are the two main factors behind GPU performance [7]. In fact, both of them are related since maximizing the occupancy can help to cover latency during global memory loads. We present several experiments aimed at analyzing their relative importance. Our results indicate that code that target efficient memory usage are the major determinant of actual performance. Overall, they ensure the best performance even if some resources remain un utilized. Therefore, maximizing occupancy should be examined at a later stage in the compilation process, once data related issues have been properly addressed.

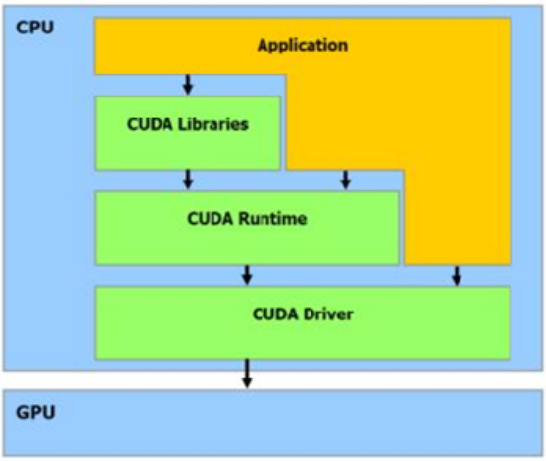

Figure 2.1: CUDA Software Stack

2.2 CUDA Hardware: Memory Model

CPU and GPU have separate memory spaces. CPU memory known as host memory and GPU memory known as Device memory. CUDA offers different types of memories with different configuration. The local, global, constant and texture spaces are regions of device memory [5]. Each multiprocessor has following memory space as shown in figure 2.2

- **Local Memory:** is small volume of memory, which can be accessed only by one streaming processor. The local memory space resides in device memory, so local memory accesses have same high latency and low bandwidth as global memory accesses. Local memory accesses only occur for some automatic variables. Automatic variables that the compiler is likely to place in local memory are [5]:
  - Arrays for which it cannot determine that they are indexed with constant quantities,
  - Large structures or arrays that would consume too much register space,
  - Any variable if the kernel uses more registers than available (this is also known as register spilling).
- **Global Memory:** the largest volume of memory available to all multiprocessors in a GPU, from 256 MB to 1.5 GB in modern solutions (and up to 4 GB in Tesla). It offers high bandwidth, over 100 GB/s for top solutions from NVIDIA, but it suffers from very high latencies (several hundred cycles). Non-catchable supports general load and store instructions, and usual pointers to memory.

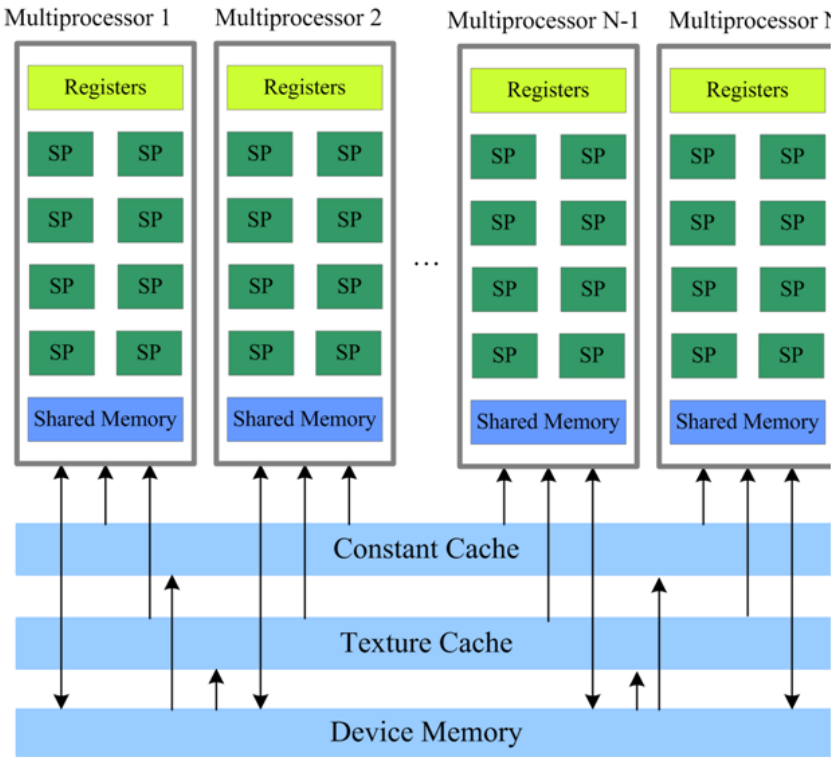

Figure 2.2: CUDA Memory Model

Global memory resides in device memory and device memory is accessed via 32, 64, or 128-byte memory transactions. These memory transactions must be naturally aligned: Only the 32, 64, or 128-byte segments of device memory that are aligned to their size (i.e. whose first address is a multiple of their size) can be read or written by memory transactions.

- **Shared Memory:** is 16-KB memory shared between all streaming processors in a multiprocessor. Because it is on-chip, the shared memory space is much faster than the local and global memory spaces.
  To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module.

- **Constant Memory:** is a 64 KB, read only memory for all multiprocessors. It's cached by 8 KB for each multiprocessor. The constant memory space resides in device memory. A constant memory request for a warp is first split into two requests, one for each half-warp, that are issued independently. A request is then split into

Memory	Location	Cached	Access	Scope
Register	On-Chip	NO	Read/Write	One Thread
Local	Off-Chip	NO	Read/Write	One Thread
Global	Off-Chip	No	Read/Write	All Threads + Host
Constant	Off-Chip	YES	Read	All Threads + Host
Texture	Off-Chip	YES	Read	All Threads + Host

Table 2.1: Characteristics of CUDA memories

as many separate requests as there are different memory addresses in the initial request, decreasing throughput by a factor equal to the number of separate requests. The resulting requests are then serviced at the throughput of the constant cache in case of a cache hit, or at the throughput of device memory otherwise. This memory is rather slow latencies of several hundred cycles, if there are no required data in cache.

- **Texture Memory:** space resides in device memory and is cached in texture cache, so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together in 2D will achieve best performance. Also, it is designed for streaming fetches with a constant latency; a cache hit reduces DRAM bandwidth demand but not fetch latency.

2.3 CUDA Hardware: Execution Model

CUDA Execution model consist of Grid, ThreadBlocks, and Threads.

- **Grid : GPU:** An entire grid is handled by a single GPU chip.
- **ThreadBlocks : MP:** The GPU chip is organized as a collection of multiprocessors (MPs), with each multiprocessor responsible for handling one or more blocks in a grid. A block is never divided across multiple MPs.
- **Threads : SP:** Each MP is further divided into a number of stream processors (SPs), with each SP handling one or more threads in a block.

2.4 CUDA Programming Model

A CUDA program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU. The GPU is viewed as a compute device : that is a coprocessor to the CPU(host), has its own DRAM(device Memory), Runs many threads in parallel [8] . Data parallel portion of application are executed on the device as kernels which run in parallel on many threads. Difference between GPU and CPU thread are:

- GPU threads are extremely lightweight and requires very little creation overhead.
- GPU needs 1000s of threads for full efficiency where as multicore cpu needs only a few.

A kernel is executed as a grid of thread blocks. A thread block is a batch of thread that can cooperate with each other by efficiently sharing data through shared memory, and synchronizing there execution for hazard free shared memory accesses. There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. Blocks are organized into a one-dimensional or two-dimensional grid of thread blocks as illustrated by figure 2.3. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system. It must be possible to execute thread block in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write code that scales with the number of cores. For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and thread synchronization is expected to be lightweight.

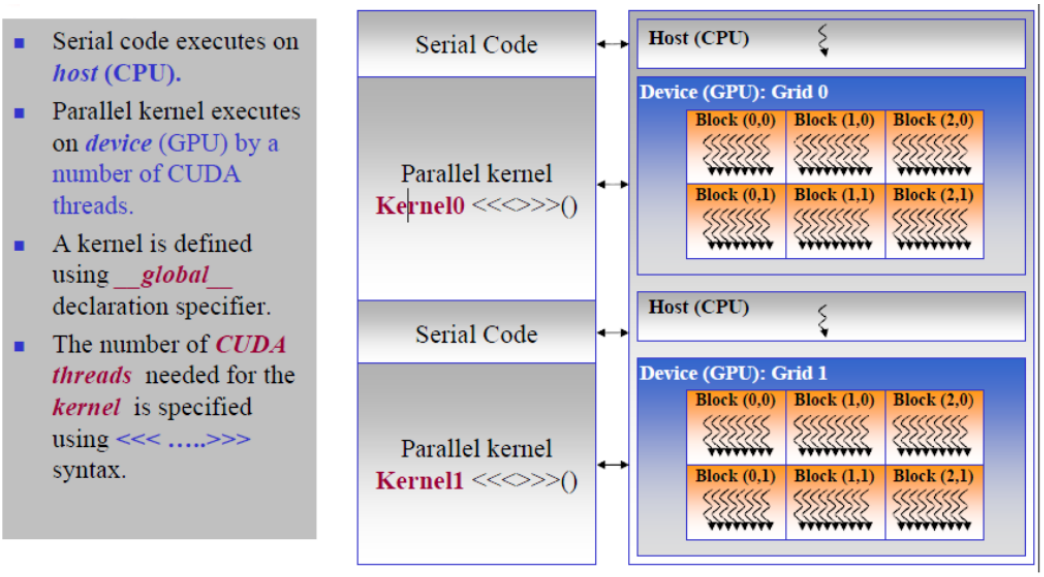

Figure 2.3: CUDA Programming/Execution Model.

2.4.1 Executing Code on the GPU

Initially program execution start from CPU,serial portion of code are run on CPU and parallel portion are transfer to GPU. it means computionally intensive part are transfer to the GPU and are execute with number of threads. figure 2.4. shows execution on CPU and GPU [9].

Program execution steps [5]:

1. First program execution start from CPU.
2. Allocate memory on host and device. for memory allocation on CPU, malloc() and calloc() functions are used. for memory allocation on GPU, cudaMalloc() and cudaMemset() functions are used. for example.

```
cudaMalloc(void **pointer, size_t nbytes)
cudaMemset(void *pointer, int value, size_t count)
```

3. Initialize data.

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *d_a = 0;
```

4. Copy form HOST to DEVICE using cudaMemcpy().

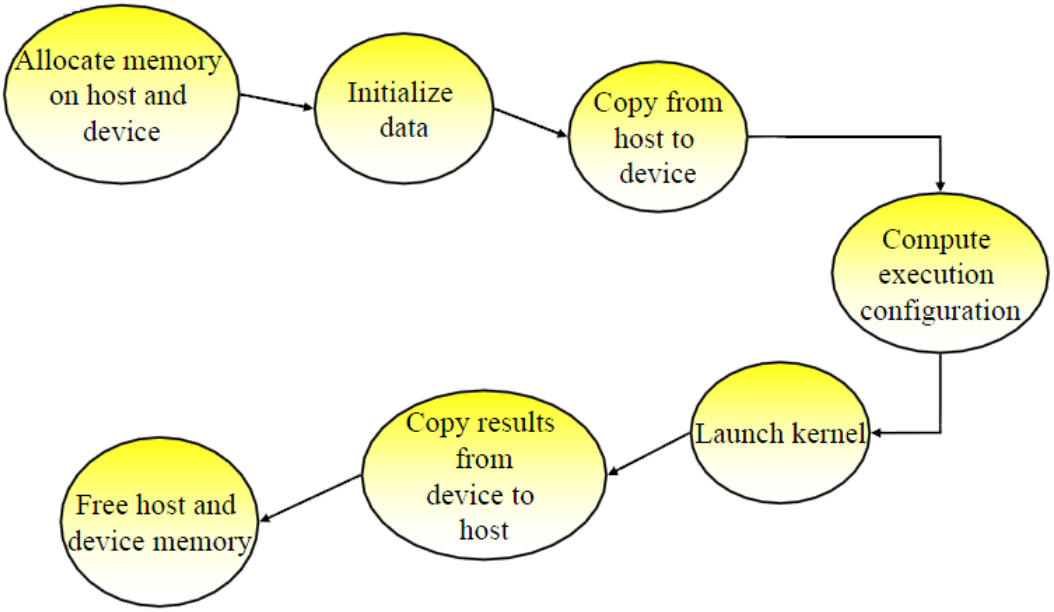

Figure 2.4: Complete Programming Process on CPU and GPU

```
cudaMemcpy(void *dst, void *src, size_t nbytes,
enum cudaMemcpyKind direction);
```

- direction specifies locations device of src and dst.
- enum cudaMemcpyKind may be one of the following.
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToDevice

5. Executing code on GPU.

```
__global__ void assign2D(int* d_a, int w, int h, int value)
{
int iy = blockDim.y * blockIdx.y + threadIdx.y;
int ix = blockDim.x * blockIdx.x + threadIdx.x;
int idx = iy * w + ix;
d_a[idx] = value;
}
...
...
```

C functions knows as Kernels are executing on GPU with some restrictions.

- Can only access GPU memory

- Must have void return type
- No variable number of arguments
- Not recursive
- No static variables

Function arguments automatically copied from CPU to GPU memory. C functions can be declared with a qualifier:

- global : invoked by CPU and execute on GPU,cannot be called from GPU and must return void
- device : called from other GPU functions,cannot be called from host (CPU). code
- host : can only be executed by CPU, called from host

All global and device functions have access to following automatically defined variables.

- **gridDim:** This variable is of type dim3 and contains the dimensions of the grid.
- **blockIdx:** This variable is of type uint3 and contains the block index within the grid.
- **blockDim:** This variable is of type dim3 and contains the dimensions of the block.
- **threadIdx:** This variable is of type uint3 and contains the thread index within the block.
- **WarpSize:** This variable is of type int and contains the warp size in threads.

6. Lunching Kernel Kernel is lunch by following code:

```
kernel<<<dim3 grid, dim3 block>>>()
```

Execution configuration:

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

7. Copy form DEVICE to HOST using cudaMemcpy().

```
cudaMemcpy(void *dst, void *src, size_t nbytes,
enum cudaMemcpyKind direction);
```

- direction specifies locations host of src and dst.
- enum cudaMemcpyKind is the following.
  - cudaMemcpyDeviceToHost

8. Free HOST and DEVICE memory using free() and cudaFree().

```
Free(void *pointer)
cudaFree(void *pointer)
```

2.5 Performance Optimization Strategies

Performance optimization revolves around three basic strategies [5] :

- Maximize parallel execution to achieve maximum utilization
- Optimize memory usage to achieve maximum memory throughput
- Optimize instruction usage to achieve maximum instruction throughput

2.5.1 Maximize Utilization

To maximize utilization the application should be structured in a way that it exposes as much parallelism as possible and efficiently maps this parallelism to the various components of the system to keep them busy most of the time.

Application Level

At a high level, the application should maximize parallel execution between the host, the devices, and the bus connecting the host to the devices, by using asynchronous functions calls and streams. It should assign to each processor the type of work it does best: serial workloads to the host; parallel workloads to the devices [10]. For parallel execution program is divided into threads, this threads need to share data with each other, there are two cases:

- If this threads belong to same block, they should use _ syncthreads() and share data through shared memory.
- If threads belong to different blocks, they must share data through global memory. In this case two separate kernel invocations are required, one for writing to and one for reading from global memory.

The second case adds extra overhead of kernel invocations and also increase global memory traffic. Its occurrence should therefore be minimized by mapping the algorithm to the CUDA programming model in such a way that the computations that require inter-thread communication are performed within a single thread block as much as possible.

Device Level

At a lower level, the application should maximize parallel execution between the multiprocessors of a device. For devices of compute capability 1.x, only one kernel can execute on a device at one time, so the kernel should be launched with at least as many thread blocks as there are multiprocessors in the device. For devices of compute capability 2.x, multiple kernels can execute concurrently on a device, so maximum utilization can also be achieved by using streams to enable enough kernels to execute concurrently [5].

Multiprocessor Level

At an even lower level, the application should maximize parallel execution between the various functional units within a multiprocessor. To maximize utilization, a GPU multiprocessor relies on thread-level parallelism to maximize utilization of its functional units. Utilization is therefore directly linked to the number of resident warps. At every instruction issue time, a warp scheduler selects a warp that is ready to execute its next instruction, if any, and issues the instruction to the active threads of the warp. The number of clock cycles it takes for a warp to be ready to execute its next instruction is called latency, and full utilization is achieved when the warp scheduler always has some instruction to issue for some warp at every clock cycle during that latency period, or in other words, when the latency of each warp is completely hidden by other warps. How many instructions are required to hide latency depends on the instruction throughput. The number of blocks and warps residing on each multiprocessor for a given kernel call depends on the execution configuration of the call, the memory resources of the multiprocessor, and the resource requirements of the kernel. To assist programmers in choosing thread block size based on register and shared memory requirements, the CUDA Software Development Kit provides a spreadsheet, called the CUDA Occupancy Calculator, where occupancy is defined as the ratio of the number of resident warps to the maximum number of resident warps. The performance of an application is also depends on the kernel code. The number of threads per block should be chosen as a multiple of the warp size to avoid wasting of computing resources with under-populated warps as much as possible.

2.5.2 Maximize Memory Throughput

Memory optimizations are the most important area for performance. The goal is to maximize the use of the hardware by maximizing bandwidth. Bandwidth is best served by using as much fast memory and as little slow-access memory as possible. This section discusses best way to set up data items to use the memory effectively.

Data Transfer between Host and Device

There are various ways to transfer data between host and device, which method provides best performance depend on the type of application and type of data, the application has to process.

Global Memory

The global memory space resides in device memory, so global memory accesses have high latency and low bandwidth accesses and are subject to the requirements for memory coalescing. To achieve high bandwidth we can use following options:

- **Minimize data transfer between the host and the device:**
  Applications should be structure in such a way that minimize data transfer between the host and the device.

- **Group transfers:**
  Also, because of the overhead associated with each transfer, batching many small transfers into a single large transfer always performs better than making each transfer separately.

- **Page-Locked Data Transfers:**
  On systems with a front-side bus, higher performance for data transfers between host and device is achieved by using page-locked host. In addition, when using mapped page-locked memory, there is no need to allocate any device memory and explicitly copy data between device and host memory. Data transfers are implicitly performed each time the kernel accesses the mapped memory. For maximum performance, these memory accesses must be coalesced as with accesses to global memory. Assuming that they are and that the mapped memory is read or written only once, using mapped page-locked memory instead of explicit copies between device and host memory can be a win for performance.
  Allocating too much page-locked memory can reduce overall system performance, so we need to test our system and application to find their limits.

Shared Memory

To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module.

Texture Memory

Textures are always read-only. You cannot write to a texture memory (which is actually bound to global memory). Texture cache is present in every multi-processor. Texture should be used when threads of a block accesses different areas of the bound-global memory in a non-orderly fashion. However if they exhibit spatial locality (2D or 1D) then you need to bind your texture in an appropriate way (1D or 2D) to take advantage. here are cases where multiple blocks operating in a multi-processor taking advantage of the caches done by the other concurrently executing block on that MP. When accessing 2D arrays in global memory, using the Texture Cache has many benefits, like filtering and not having to care as much for memory access patterns. Textures can be faster, the same speed, or slower than "naked" global memory access. There are no general rules of thumb for predicting performance using textures, as the speed up (or lack of speed up) is determined by data usage patterns within your code and the texture hardware being used.

2.5.3 Maximize Instruction Throughput

To maximize instruction throughput the application should :

- Minimize the use of arithmetic instructions with low throughput; this includes trading precision for speed when it does not affect the end result, such as using intrinsic instead of regular functions, single-precision instead of double-precision, or using re-normalized numbers to zero.

- Reduce the number of instructions, for example, by optimizing out synchronization points whenever possible.

2.6 CUDA Installation

CUDA installation consists of

1. **Driver :** Required component to run CUDA applications.

2. **CUDA Toolkit (compiler,libraries):** The CUDA Toolkit is a C language development environment for CUDA-enabled GPUs. The CUDA development
   - NVCC compiler
   - CUDA FFT and BLAS libraries for the GPU
   - gdb debugger for GPU
   - CUDA runtime driver is included into the standard NVIDIA driver
   - NVCC Profiler
3. **CUDA SDK (example codes):** The CUDA Developer SDK provides examples with source code to help
   - Parallel bitonic sort
   - Matrix multiplication
   - Parallel prefix sum (scan) of large arrays
   - CUDA BLAS and FFT library usage examples
   - Device query etc

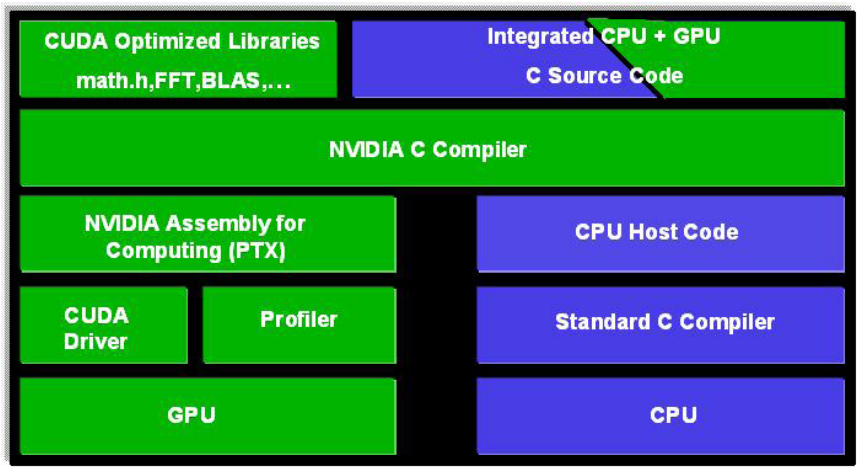

Figure 2.5: CUDA Software Development

As figure 2.5 shows, CUDA includes C/C++ software development tools, function libraries, and a hardware abstraction mechanism that hides the GPU hardware from developers. Although CUDA requires programmers to write special code for parallel processing, it doesn't require them to explicitly manage threads in the conventional sense, which greatly simplifies the programming model. CUDA development tools work along side a conventional C/C++ compiler, so programmers can mix GPU code with general-purpose code for the host CPU [1].

2.6.1 CUDA Compiler - NVCC

- Any source file containing CUDA language extensions (.cu) must be compiled with nvcc.
- NVCC is a compiler driver works by invoking all the necessary tools and compilers like cudacc, g++, cl etc
- NVCC can output:
  - Either C code (CPU Code) - To be compiled with the rest of the application using another tool.
  - Or PTX object code directly.
- An executable with CUDA code requires:
  - The CUDA core library (cuda)
  - The CUDA runtime library (cudart)

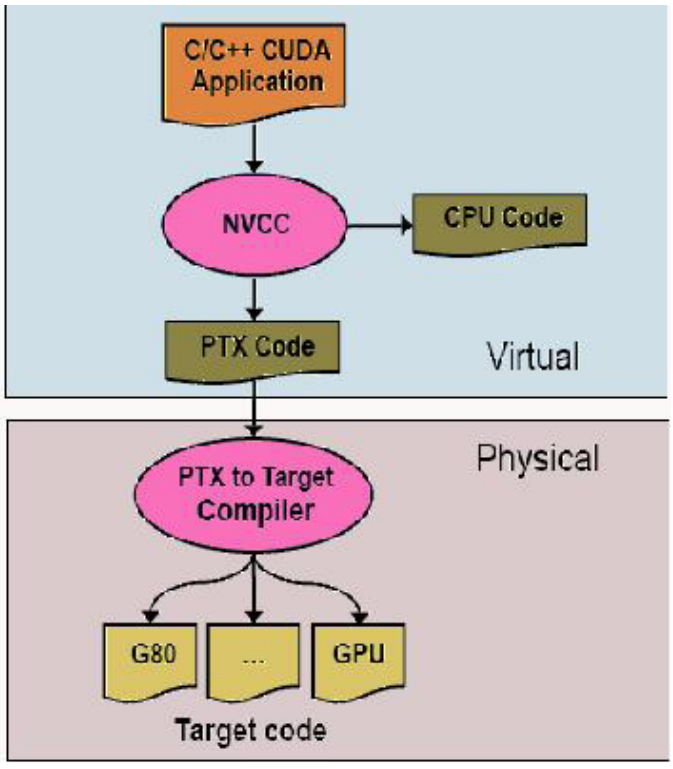

Figure 2.6: CUDA Compiler

Figure 2.6 shows CUDAs compilation process. Source code written for the host CPU follows a fairly traditional path and allows developers to choose their own C/C++ compiler, but preparing the GPUs source code for execution requires additional steps. Among

the unusual links in the CUDA tool chain are the EDG preprocessor, which separates the CPU and GPU source code; the Open64 compiler, originally created for Itanium; and NVIDIAs PTX -to-Target Translator, which converts Open64s assembly- language output into executable code for specific NVIDIA GPUs.

2.6.2 Build Configurations

- nvcc filename.cu[-o executable]
  Builds release mode.

- nvcc -g filename.cu
  Builds debug mode can debug host code but not device code.

- nvcc -deviceemu filename.cu
  Builds device emulation mode.
  All code runs on CPU, no debug symbols.

- nvcc -deviceemu -g filename.cu
  Builds debug device emulation mode.
  All code runs on CPU, with debug symbols.

Now following code find GPU device on machine.

```
// CUDA Device Query
#include <stdio.h>
// Print device properties
void printDevProp(cudaDeviceProp devProp)
{
    printf("Major revision number:%d\n",  devProp.major);
    printf("Minor revision number:%d\n",  devProp.minor);
    printf("Name:%s\n",  devProp.name);
    printf("Total global memory:%u\n",  devProp.totalGlobalMem);
    printf("Total shared memory per block:%u\n",  devProp.sharedMemPerBlock);
    printf("Total registers per block:%d\n",  devProp.regsPerBlock);
    printf("Warp size:                     %d\n",  devProp.warpSize);
    printf("Maximum memory pitch:          %u\n",  devProp.memPitch);
    printf("Maximum threads per block:     %d\n",  devProp.maxThreadsPerBlock);
    for (int i = 0; i < 3; ++i)
    printf("Maximum dimension %d of block:  %d\n", i, devProp.maxThreadsDim[i]);
    for (int i = 0; i < 3; ++i)
    printf("Maximum dimension %d of grid:   %d\n", i, devProp.maxGridSize[i]);
```

```
    printf("Clock rate:                    %d\n",  devProp.clockRate);
    printf("Total constant memory:         %u\n",  devProp.totalConstMem);
    printf("Texture alignment:             %u\n",  devProp.textureAlignment);
    printf("Concurrent copy and execution:%s\n",(devProp.deviceOverlap?"Yes":"No"));
    printf("Number of multiprocessors:     %d\n",  devProp.multiProcessorCount);
    printf("Kernel execution timeout:%s\n",(devProp.kernelExecTimeoutEnabled?Yes":"No"));
    return;
}
int main()
{
    // Number of CUDA devices
    int devCount;
    cudaGetDeviceCount(&devCount);
    printf("CUDA Device Query...\n");
    printf("There are %d CUDA devices.\n", devCount);
    // Iterate through devices
    for (int i = 0; i < devCount; ++i)
    {
        // Get device properties
        printf("\nCUDA Device #%d\n", i);
        cudaDeviceProp devProp;
        cudaGetDeviceProperties(&devProp, i);
        printDevProp(devProp);
    }
     printf("\nPress any key to exit...");
    char c;
    scanf("%c", &c);
    return 0;
}
```

figure 2.7 shows output of above code.

2.6.3 CUDA Occupancy Calculator

CUDA Occupancy calculator used for finding occupancy of CUDA resources. Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep the hardware busy [11]. Occupancy = Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently.

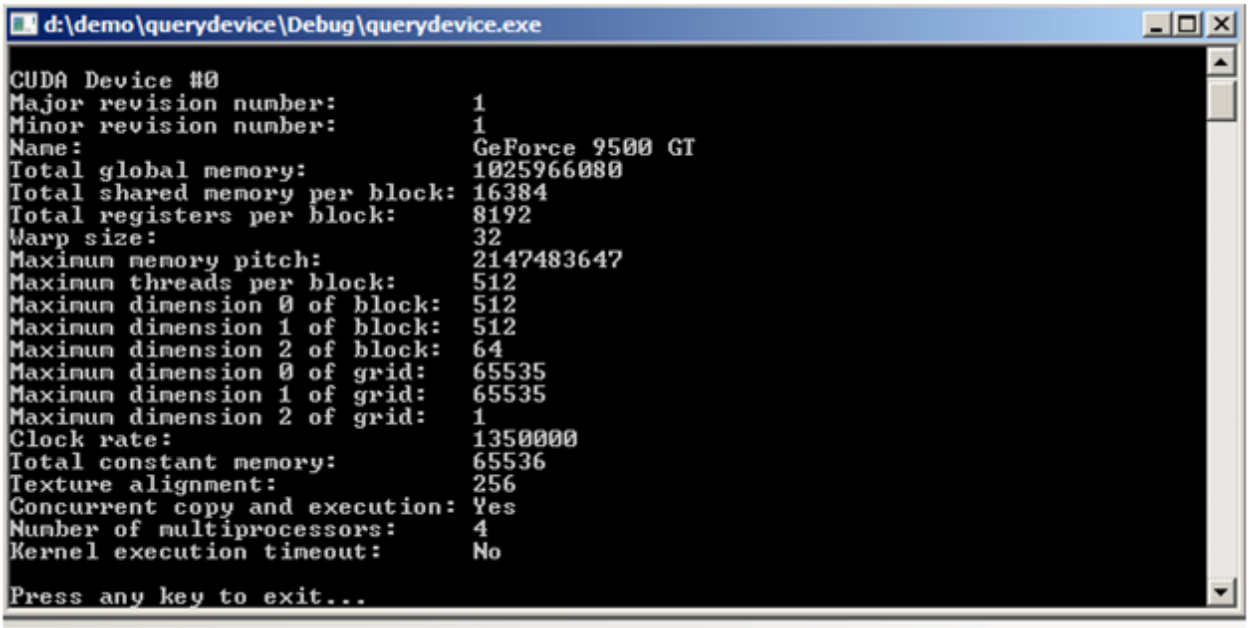

Figure 2.7: Device Query Output

- Limited by resource usage: 1.Registers 2.Shared memory
- Choose threads per block as a multiple of warp size
  Avoid wasting computation on under-populated warps
  Facilitates coalescing
- More threads per block always not higher occupancy
- Granularity of allocation
  Eg. compute capability 1.1 (max 768 threads/multiprocessor)
  512 threads/block = 66
  256 threads/block can have 100
- Heuristics
  Minimum: 64 threads per block - Only if multiple concurrent blocks
  192 or 256 threads a better choice -Usually still enough registers to compile and invoke successfully.

In this figure 2.8 display the CUDA Occupancy Calculator. Which we can measure the resource usage. Here we can input the resource usage like, number of thread blocks used, number of registers per thread block and shared memory per block in bytes. So as per the compatibility of GPU it gives the GPU occupancy of data like active threads per multiprocessor, active warps per multiprocessor and thread blocks per multiprocessor.

So we can get the occupancy of each multiprocessor. Also in this calculator, as per the resource usage graphs are plotted for measuring the occupancy. Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts and shared memory allocation.

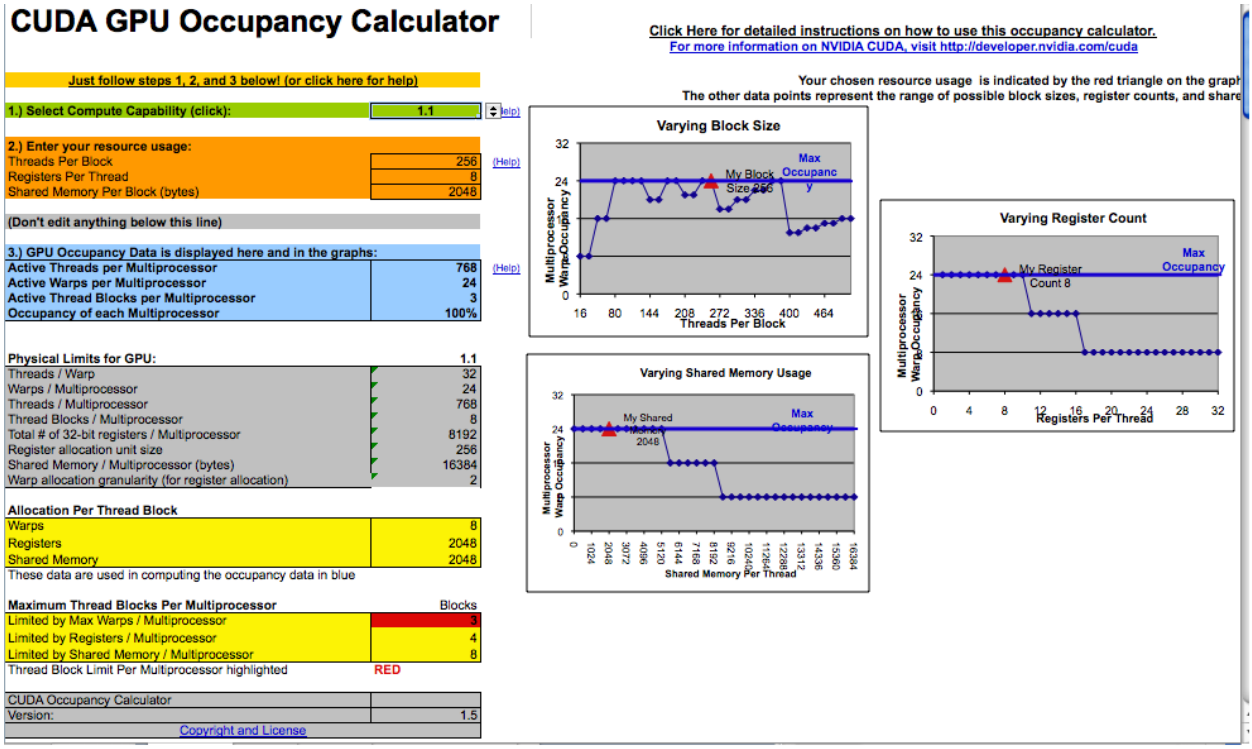

Figure 2.8: CUDA Occupancy Calculator

2.7 Related Work

Following recent work has been focused on GPU to perform ME in H.264 video encoding.

- In this work authors implemented a BMA with an FS over all possible candidate vectors on a regular grid [12]. The multicore GPU implementation has two relevant stages:

  - Start a thread to work with quadruplet (k,l,m,n) where k and l are image block identifiers and m and n are identifiers of one candidate displacement vectors. Each thread will compute the jk,l(V m,n),defined as SAD, for the Bk,l block and the displacement candidate, ,and store it to a global memory. So for each, , block we will have a total of M*N threads computing all possible values of Jk,l(Vm,n).
  - Next a trivial thread is launched to find the minimum,Jk,l(Vm,n) , value over m=1,2,....,M,n=1,2...,N,(M*N), , stored values. Global memory access is one of the main GPU bottlenecks. To minimize this, in Code 1 we use two mechanisms: 1) the reference,It(i,j) , and target,It+1(i,j), images are stored in 2D cached texture memory, 2) all other variables are stored in fast register memory associated with the processor, and only one write to global memory is done at the thread end in order to store the calculated value of Jk,l(Vm,n).

  The GPU execution time, for HDTV images, was 7.23 seconds whereas for CPU implementation it was 8025.00 seconds or 2 hours 13 min 45 seconds, so the speedup for processing images in HDTV resolution was almost 1110 fold.

- In this paper [13], author proposed a CUDA based parallelized approach to implement the most time consuming H.264 coding process, FS motion estimation. CUDA is a powerful GPU architecture, which offers parallel computation capability through hundreds of highly decoupled processing cores to accelerate arithmetic intensive applications. In proposed algorithm, whole full-pixel motion estimation for a MB is integrated in a single CUDA kernel, parallel calculating and comparing variable block-size SAD values. At the same time, this method takes full advantage of high-speed on-chip memory of GPU, such as registers and shared memory to minimize the amount of device DRAM access. Experimental results show that the proposed approach can be 50 times faster than the traditional CPU implementation; and even 4CIF sequences are close to real-time applications. For the performance comparison author might not consider memory transfer for GPU to CPU.

- Y Lin et al. [14] proposed a multi-pass algorithm to accelerate the motion estimation on traditional GPU architecture. With the multi- pass method to unroll and rearrange the multiple nested loops in motion estimation, about 2 times and 14 times speed-up can be achieved for integer-pixel ME and half-pixel ME respectively. However, the multi-pass algorithm based on traditional GPU architecture cannot take full use of the powerful computing resources of GPU.

- In [15] the authors were able to achieve 1.47 times speed up for H.264/AVC motion estimation.A serial dependence between motion estimation of macro blocks in a video frame is removed to enable parallel execution of the motion estimation code. Although this modification changes the output of the program, it is allowed within the H.264 standard.

- In [16] the authors were used texture memory to store current frame and reference frame instead of global memory. The texture memory space is cached so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the texture cache. The source and reference frames are first stored in texture memory in each iteration. This speeds up the memory transfer for SAD-Calculation,while authors did not take advantage of shared memory to preloaded the needed pixel values.

Chapter 3

Motion Estimation of Video Compression

3.1 Introduction of Video Compression

In order to understand what 'Motion Estimation' is, it is essential to first have an overview of video encoding process of MPEG4 AVC/H.264 [2] [19]. MPEG4 AVC/H.264 thus has the following scopes of video encoding:

- Step 1: Reduction of the Resolution
- Step 2: Motion Estimation
- Step 3: Discrete Cosine Transform (DCT)
- Step 4: Quantization
- Step 5: Entropy Coding

Among all video encoding steps, motion estimation is very time consuming step. from JM reference 12 [3] we analyze that motion estimation require about 75% time of whole compression technique. figure 3.1 shows time line chart of video encoding. Up to this point we are considering only motion estimation step thus it is require lots of time for comparison and computation.

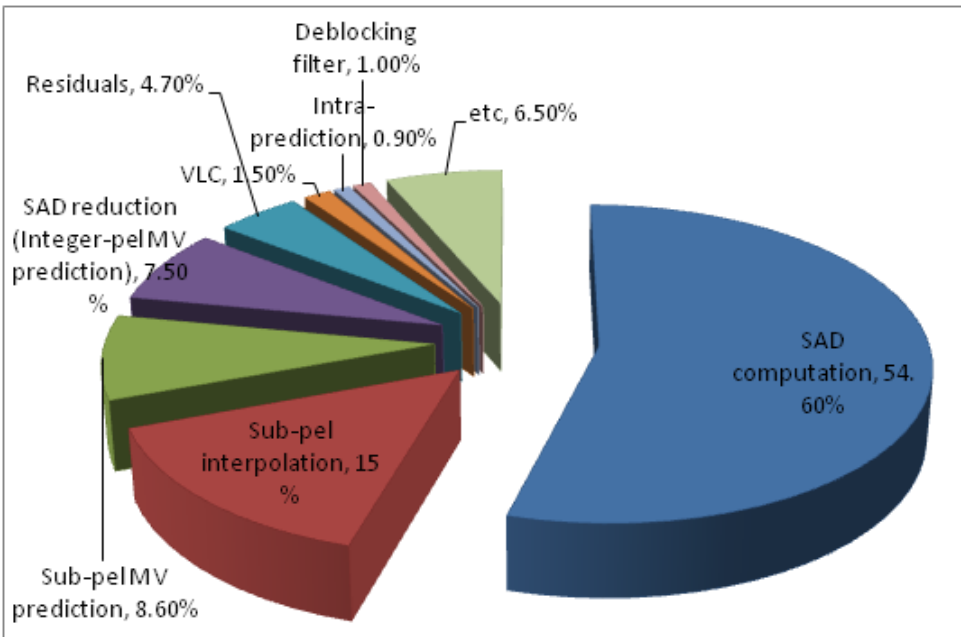

Figure 3.1: A time line chart of MPEG4 compression technique

3.2 Motion Estimation

Video are basically consists of images/frames. figure 3.2 shows video with 5 frames. these five frames having lots of similarity between frames and within frames [17]. There are two redundancy reduction principles used:

- Spatial redundancy (Intra-frame prediction)
- Temporal redundancy (Inter-frame prediction)

Motion Estimation is a part of 'Inter-frame prediction' technique. Inter coding refers to a mechanism of finding 'co-relation' between two frames (still images), which are not far away from each other as far as the order of occurrence is concerned, one called the reference frame and the other called current frame, and then encoding the information which is a function of this 'co-relation' instead of the frame itself. Motion Estimation is the basis of inter coding, which exploits temporal redundancy between the video frames, to scope massive visual information compression [18]. Changes between frames are mainly due to the movement of objects. for the current processing frame find movement of object form reference frame is known as motion estimation. as output it find motion vectors of objects and it is used for further processing. The encoder then uses this motion information to move the contents of the reference frame to provide a better prediction of the current frame. This process is known as motion compensation (MC),

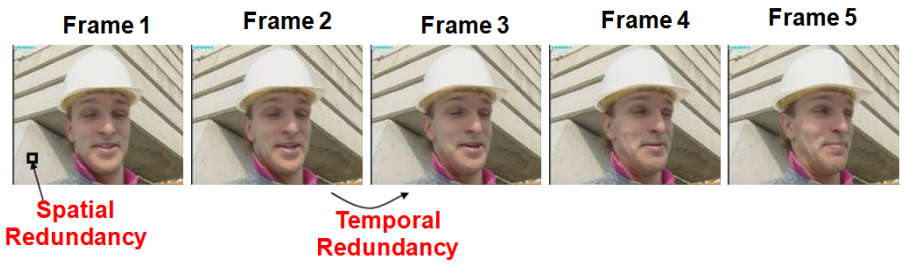

Figure 3.2: Video Frames

and the prediction so produced is called the motion-compensated prediction (MCP) or the displaced-frame (DF). figure 3.3 shows process of motion estimation.

Figure 3.3: Motion Vector information

3.3 Block Matching Methods

In a typical Block Matching Algorithm, each frame is divided into blocks, each of which consists of luminance and chrominance component blocks. Usually, for coding efficiency, motion estimation is performed only on the luminance component. Each luminance block in the present frame is matched against candidate blocks in a search area on the reference frame. These candidate blocks are just the displaced versions of original block. The best candidate block is found and its displacement (motion vector) is recorded. In a typical interframe coder, the input frame is subtracted from the prediction of the reference frame. Consequently the motion vector and the resulting error can be transmitted instead of the original luminance block; thus interframe redundancy is removed and data compression

is achieved. At receiver end, the decoder builds the frame difference signal from the received data and adds it to the reconstructed reference frames. figure 3.4 shows process of block-matching algorithm. This algorithm is based on a translational model of the

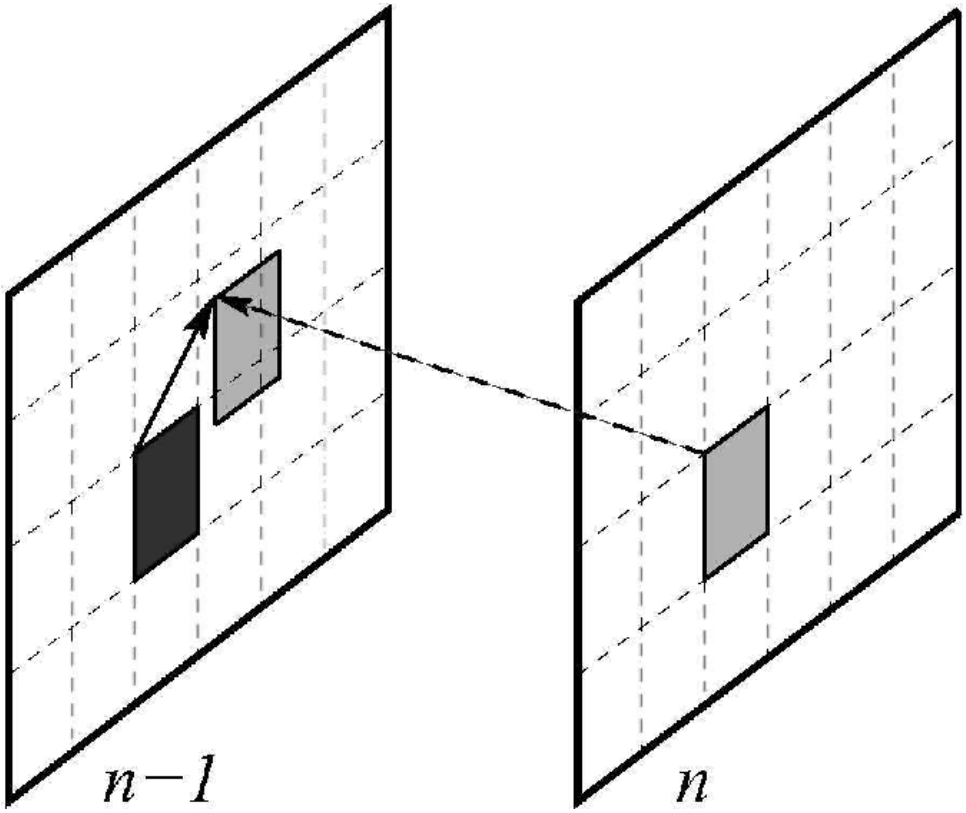

Figure 3.4: Block-matching Motion Estimation

motion of objects between frames. It also assumes that all pixels within a block undergo the same translational movement. There are many other ME methods, but BMME is normally preferred due to its simplicity and good compromise between prediction quality and motion overhead.

There are many other approaches to motion estimation, some using the frequency or wavelet domains, and designers have considered scope to invent new methods since this process does not need to be specified in coding standards. The standards need only specify how the motion vectors should be interpreted by the decoder. Block Matching (BM) is the most common method of motion estimation. Typically each macro block (8×8 pixels) in the new frame is compared with shifted regions of the same size from the previous decoded frame, and the shift which results in the minimum error is selected as the best motion vector for that macro block. The motion compensated prediction frame is then formed from all the shifted regions from the previous decoded frame [20].

3.3.1 Block Matching

Block-matching motion estimation (BMME) is the most widely used motion estimation method for video coding. Interest in this method was initiated by Jain and Jain and he proposed a block-matching algorithm (BMA)in 1981 [17]. The current frame is first divided into blocks of $M \times N$ pixels. The algorithm then assumes that all pixels within the block undergo the same translational movement. Thus, the same motion vector is assigned to all pixels within the block. This motion vector is estimated by searching for the best match block in a larger search window (double then the size of macroblocks), pixels centered at the same location in a reference frame.

3.3.2 Matching Criteria for Motion Estimation

Inter frame predictive coding is used to eliminate the large amount of temporal and spatial redundancy that exists in video sequences and helps in compressing them. In conventional predictive coding the difference between the current frame and the predicted frame is coded and transmitted. The better the prediction, the smaller the error and hence the transmission bit rate when there is motion in a sequence, then a pixel on the same part of the moving object is a better prediction for the current pixels. There are a number of criteria to evaluate the goodness of a match. Three popular matching criteria used for block-based motion estimation are

- Mean of Squared Error (MSE)
- Sum of Absolute Difference (SAD)

MSE Criterion

Considering (k-l) as the past references frame l ¿ 0 for backward motion estimation, the mean square error of a block of pixels computed at a displacement (i, j) in the reference frame is given by:

$$MSE(i,j) = \frac{1}{N^2} \sum_{n_1=0}^{N-1} \sum_{n_2=0}^{N-1} [s(n_1, n_2, k) - s(n_1 + i, n_2 + j, k - l)]^2$$

Consider a block of pixels of size $N \times N$ in the reference frame, at a displacement of, where i and j are integers with respect to the candidate block position. The MSE is computed for each displacement position (i, j),within a specified search range in the reference image and the displacement that gives the minimum value of MSE is the displacement vector which is more commonly known as motion vector.

The MSE criterion requires computation of N^2 subtractions, N^2 multiplications (squaring) and (N^2-1) additions for each candidate block at each search position. This is computationally costly and a simpler matching criterion, as defined below is often preferred over the MSE criterion.

SAD Criterion

Like the MSE criterion, the sum of absolute difference (SAD) too makes the error values as positive, but instead of summing up the squared differences, the absolute differences are summed up. The SAD measure at displacement (i, j) is defined as

$$SAD(i,j) = \frac{1}{N^2} \sum_{n_1=0}^{N-1} \sum_{n_2=0}^{N-1} [s(n_1, n_2, k) - s(n_1 + i, n_2 + j, k - l)]$$

SAD find displacement vector of block with respect to current frame and reference frame. The SAD criterion requires N^2 computations of subtractions with absolute values and additions N^2 for each candidate block at each search position. The absence of multiplications makes this criterion computationally more attractive and facilitates easier for implementation.

3.3.3 Block Size

Important factor of the BMA is the block size. If the block size is smaller, it achieves better prediction quality, because a smaller block size reduces the effect of the accuracy problem.Since a smaller block size means that there are more blocks (and consequently more motion vectors) per frame, this improved prediction quality comes at the expense of a larger motion overhead. Most video coding standards use a block size of 16×16 as a compromise between prediction quality and motion overhead. A number of variable-block-size motion estimation methods have also been proposed in the literature. H.263 and MPEG AVC standards allows adaptive switching between block sizes of 16×16 and 8×8 on an Macroblock (MB) basis. Motion compensation for each 16×16 macroblock can be performed using a number of different block sizes and shapes. The original luminance component of each macroblock (16×16) may be spilt into 4 kinds of size: 16×16, 16×8, 8×16, 8×8, as shown in figure 3.5 Each of the sub-divided regions is a macroblock partition. If the 8×8 mode is chosen, each of the four 8×8 may be spilt into 4 kinds of size: 8×8, 8×4, 4×8, 4×4. These partitions and sub-partitions compose to a large number to a large number of possible combinations [21].

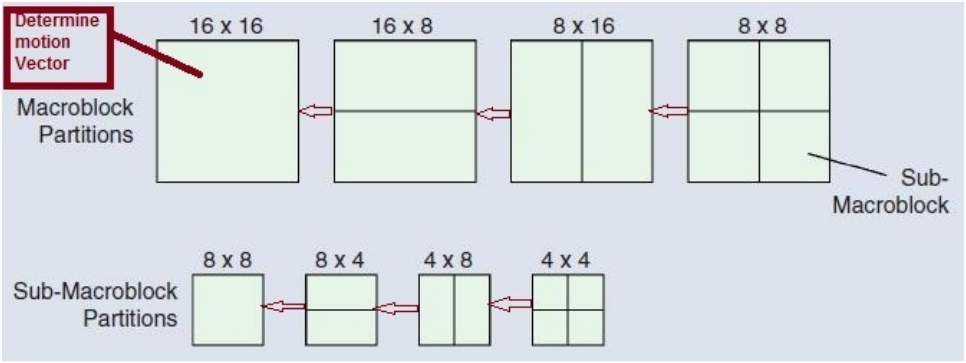

Figure 3.5: Macroblock sizes

3.3.4 Search Range

The maximum allowed search range, has a direct impact on both the computational complexity and the prediction quality of the BMA. A small search range results in poor compensation for fast-moving areas and consequently poor prediction quality. A large search range, results in better prediction quality but leads to an increase in the computational complexity. A larger search range can also result in longer motion vectors and consequently a slight increase in motion overhead. In general,search range is double then the size of macroblock, so can get better compensation result with low computational complexity.

3.3.5 Quality Judgment

The quality of a video scene can be determined using both objective and subjective approaches. The most widely used objective measure is the Peak-Signal-to-Noise- Ratio (PSNR) which is defined as:

$$PSNR = 10log_{10}[\frac{(peaktopeakvalueoforiginaldata)^2}{MSE}]$$

where the MSE is the Mean Square Error of the decoded frame and the original frame. The peak value is 255 since the pixel value is 8 bits in size. The higher the PSNR, the higher the quality of the encoding. The PSNR and bit-rate are usually conflicting, the most appropriate point being determined by the application. Although PSNR can objectively represent the quality of coding, it does not equal the subjective quality. Subjective quality is determined by a number of human testers and a conclusion is drawn based on their opinions. There exist cases for which high PSNR results in low subjective quality. However, in most cases, PSNR provides a good approximation to the subjective measure[2].

3.4 Block Matching Algorithms

- **Full Search(Exhaustive Search (ES))** Full Search block matching algorithm, is the most computationally expensive block matching algorithm of all. This algorithm calculates the cost function at each possible location in the search window. As a result of which it finds the best possible match and gives the highest PSNR amongst any block matching algorithm. The obvious disadvantage to FS is that the larger the search window gets the more computations and more compression it requires [13]. figure 3.6 shows full search block matching algorithm.

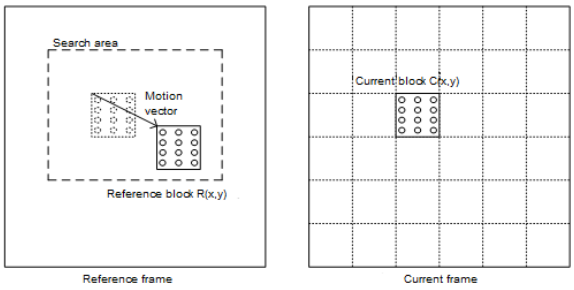

Figure 3.6: Full Search Block Matching Algorithm

- **Three Step Search (3SS)** Three Step search algorithm starts with the search location at the center and sets the step size say S = 4, for a usual search parameter value of 7. It then searches at eight locations +/- S pixels around center(0,0). From these nine locations searched so far select one having least cost and makes it the new search center. It then sets the new step size S = S/2, and repeats similar search for two more iterations until S = 1. At that point it finds the location with the least cost function and the macro block at that location is the best match.It will be used for best motion vector information [22][26]. figure 3.7 shows four step search block matching algorithm.

- **Four Step Search (4SS)** Four Step search,pattern size S=2 at the first step,and looks at 9 locations in a 5x5 window. If the least weight is found at the center of search window the search jumps to fourth step. If the least weight is at one of the eight locations except the center, then we make it the search origin and move to the second step. The search window is still maintained as 5x5 pixels wide. Depending on where the least weight location was, we might end up checking weights at 3 locations or 5 locations. Once again if the least weight location is at the center of the 5x5 search window we jump to fourth step or else we move on to third step. The

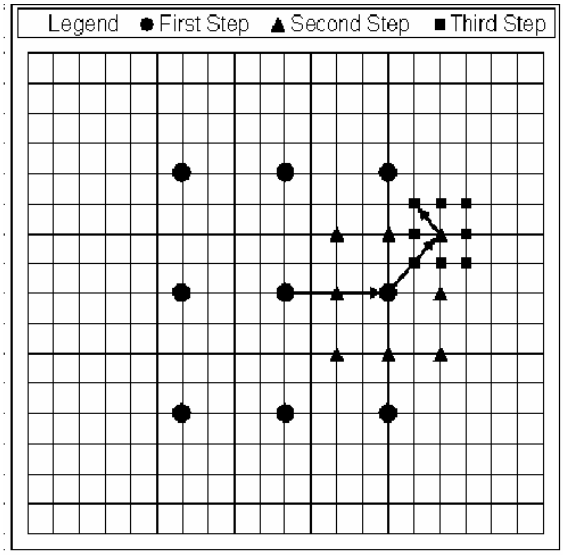

Figure 3.7: Three Step Search Block Matching Algorithm

third is exactly the same as the second step. IN the fourth step the window size is dropped to 3x3, i.e. S = 1. The location with the least weight is the best matching macro block and the motion vector for that block [24][26]. figure 3.9 shows full search block matching algorithm.

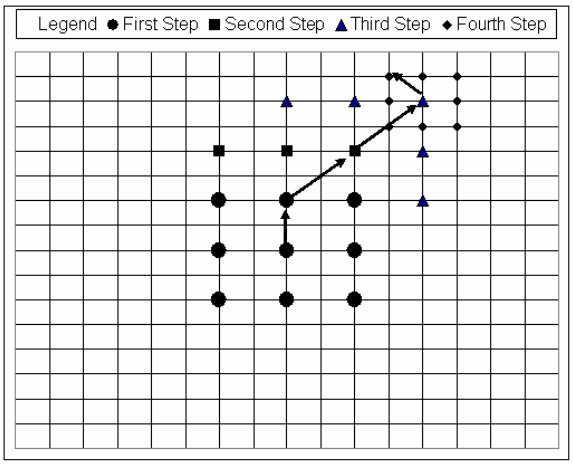

Figure 3.8: Four Step Search Block Matching Algorithm

- **Diamond Search (DS)** Diamond Search, where the search is diamond, and there is no limit on the number of steps that the algorithm can take. DS uses two different types of fixed patterns, one is Large Diamond Search Pattern (LDSP) and the other is Small Diamond Search Pattern (SDSP). At the first step uses LDSP and if the least weight is at the center location we jump to fourth step. The consequent steps,

except the last step, are also similar and use LDSP, but the number of points where cost function is checked are either 3 or 5 and are illustrated in second and third steps of procedure shown in Fig.9. The last step uses SDSP around the new search origin and the location with the least weight is the best match. As the search pattern is neither too small nor too big and the fact that there is no limit to the number of steps, this algorithm can find global minimum very accurately. The end result should see a PSNR close to that of ES while computational expense should be significantly less [23]. figure 3.9 shows diamond search block matching algorithm.

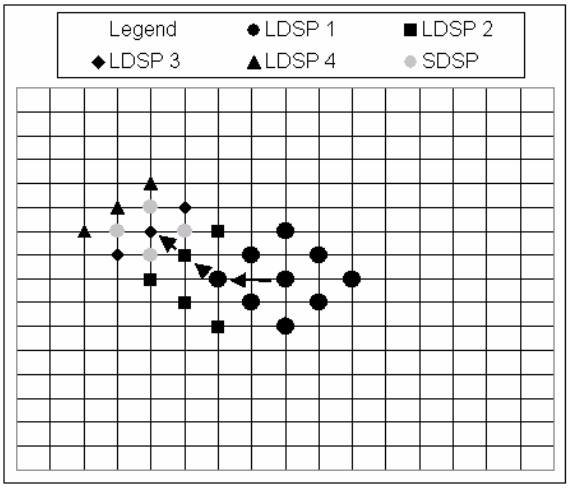

Figure 3.9: Diamond Search Block Matching Algorithm

Sequence	*Caltrain*	*Foreman*	*Weather*	*Carphone*
ES	204.2828	204.2828	204.2828	204.2828
TSS	24.3838	23.2916	23.1343	22.5824
4SS	20.4460	18.4625	15.7979	17.5911
DS	15.6392	16.1977	12.2388	15.2116

Figure 3.10: Comparison of Block Matching Algorithms [26]

Figure 3.10 shows the average of search points per macroblock for tested sequences obtained for a search window size of 7x7. While the ES test around 205 search points per macro-block, the other ME algorithms accomplish a good performances with a higher

speed-up ratio. For all algorithms, even if the number of comparison required per macro-block is clearly reduced by reference to ES, an average of 15 search points for DS, so we conclude that ES(Full Search) is very time consuming and DS is related require less time.

Sequence	*Caltrain*	*Foreman*	*Weather*	*Carphone*
ES	30,54	38,52	40,68	38,55
TSS	27,53	38,02	40,28	37,93
4SS	29,06	38,11	40,37	38,06
DS	29,56	38,35	40,63	38,43

Figure 3.11: Average values of PSNR (dB) Block Matching Algorithms [26]

Figure shows 3.11 average values of PSNR obtained for each video sequence. Results demonstrate that the algorithm DS have nearly same results as the ES algorithm for the four sequences and achieves consistent improvement in PSNR over the TSS algorithm. improvement in PSNR over the TSS algorithm. Quality of FS and DS are relatively the same still FS require more time then DS. so as part of thesis we implemented worst case and best case of block matching algorithm using CUDA.

Chapter 4

Implementation

4.1 Full Search Block Matching Algorithm

Following code shows serial C implementation:

```
for(rows of macro blocks)
  {
    for(columns of macro blocks)
        {
        for(rows of template)
            {
            for(columns of template)
                {
                SAD computation;
                SAD comparison; (Find Minimum SAD)
                }
        }
    }
 }
```

For parallel CUDA implementation we drawn two approach, one kernel approach and three kernel approach. with threads and threadblock configuration we used different types of memory for further optimization. CUDA having different types of memories, to use texture memory for image reading, the image data needs to be bind to a texture.Location of image remains off-chip memory,only a type of memory will change.Texture memory is read-only and cache so first data loaded into texture memory. in block matching algorithm data will be re-use within threadblock, so use of texture give better performance then global memory. off-chip shared memory is faster then global memory. access speed of

shared memory is same as registers. here shared memory used when data is re-used within threads in a threadblock. Also, the shared memory can be used (together with a synchronization barrier) to communicate data over threads, as is done for parallel reduction. and shared memory can be used as an extension to the register file, providing more storage space.

4.1.1 One Kernel Approach

First frame are divide equal size of marcroblocks known as candidate block. choose reference block size for example 16×16 pixels. These blocks are mapped to threadblocks, so number of threads are equals to the number of reference blocks. Now each threadblock represent all processing part of reference block. following steps are performed in one kernel approach.

- Now task of each thread are calculate SAD value,comparing candidate blocks with reference blocks. In this way, the reference block is stored once in fast shared 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.

- After the first step all threads have SAD value. so total SAD values are equal to number of threads in threadblocks. Now find minimum SAD value from all SAD in a threadblocks,By using parallel reduction find minimum SAD value from all, this method known as Tree Structure Motion Estimation. As the starting of the parallel reduction is half the number of candidate blocks, half of the threads are completely idle during the whole task, while other threads are partially idle. The shared memory is used again here, this time to communicate all the resulting SAD values and make the comparisons. Apart from the shared memory, grouping threads into threadblocks makes synchronization between threads possible.

- After finding minimum SAD candidate, values are returns to intermediate frames. Ideally, one thread should write one pixel,the number of threads in a threadblock cannot change within one kernel, it is equal to the number of candidate blocks and not to the number of pixels in a reference block. so at time of parallel reduction some threads will me idles.

Parallel CUDA implementation

```
for all threadblocks
  for all threads
      SAD_calculation = SAD(reference block , candidate block )
```

```
  end
  Minimum_SAD = parallelReduction (SAD_calculation)
  for all threads
  writedata(Minimum_SAD)
 end
 end
```

Drawback of one kernel approach

One kernel is designed with three steps, so kernel having bug task to do. while CUDA programming guide [5] suggest small kernel with hundreds of threads rather than few big threads. we cannot change threads within kernel so in the second and third step some threads will be idle and some. this drawback resolved by three kernel approach.

4.1.2 Three Kernel Approach

This approach designed three kernel having small task rather than one kernel with big task. same as one kernel approach, first frame are divide equal size of marcroblocks known as candidate block. choose reference block size for example 16×16 pixels. These blocks are mapped to threadblocks. following are pseudo-code for three kernel approach.

Parallel CUDA implementation

```
for all reference blocks
    CUDAkernel1( )
    CUDAkernel2( )
    CUDAkernel3( )
end
```

Steps for processing

- **CUDAKernel1:** Now, each SAD value is calculated by one threadblock , which is divided into threads according to the number of pixels in a candidate block. Each thread has a light weight task, calculating one absolute difference value between the reference block and its candidate block. When all threads calculated their absolute difference value, the sum must be taken to obtain the SAD value. This is done using parallel reduction The goal of the first kernel is to obtain the required 4x4 SAD costs, which are needed to build the structured motion tree in the next kernel. search area are double then size of macro block. The search area positions distribution follows a spiral pattern A GPU thread is generated for each position in

the search area for each MB in a frame. 256 GPU threads are grouped into a GPU thread block, obtaining the SAD costs for correlative positions inside the search area; one thread block calculates the SAD costs for positions 0-255, another for positions 256-511 and so on. GPU threads use 16 registers and no shared memory, obtaining a 100% GPU occupancy factor. The SAD value is obtained for 4x4 blocks, so each MB is divided into sixteen 4x4 blocks for each search area, and each thread calculates the 16 4x4 SAD costs of its associated position. These 16 SAD value are the basic data to build the structured motion tree in the next step, and they are stored in the GPU global memory. following are CUDAkernel1 code.

```
//CUDAkernel 1
for all threadBlocks
for all threads
absolute difference_4x4 (referenceblock , candidateblock)
end
sum of absolute difference = parallelreduction ( absolute difference )
end
```

- **CUDAKernel2:** Result of kernel 1 is used by kernel 2. the second kernel is to build the structured motion tree, obtaining in this way the SAD costs for all subpartition. kernel 2 makes a first reduction of the generated data. As input, kernel2 takes the motion information of 16 4x4 blocks of a MB for 64 positions and produce the motion information for all partition combinations, and reduces the amount of information, obtaining the best motion vector for each subpartition of the 64 positions. so here, 64 GPU threads are grouped into a thread block, each of them building the SAD costs for a position for all subpartitions. Intermediate results are stored in multiprocessor shared memory. following are CUDAkernel2 code.

```
//CUDAkernel 2
for all threads within threadblock
   writedata_8x8(Minimum_SAD)
```

- **CUDAKernel3:** In order to obtain the motion information for the 4x8 and 8x4 subpartitions it is only necessary to add 2 4x4 SAD costs, for the 8x8 partitions it is only necessary to add 2 4x8 SAD costs, for the 8x16 and 16x8 partitions it is only necessary to add 2 8x8 SAD costs, and finally to obtain the motion information for the 16x16 partition it is only necessary to add 2 8x16 SAD costs. Intermediate results for all partitions/subpartitions are stored in the global memory. after all

three kernel execution finally SAD value calculated by GPU will copy form DEVICE to HOST. following are CUDAkernel3 code.

```
//CUDAkernel 3
for all threads within threadblock
   writedata_16x16(Minimum_SAD)
```

4.2 Diamond Search Block Matching Algorithm

4.2.1 Two Kernel Approach

For diamond search algorithm we drawn 2 kernel approach. here we designed 2 kernel, one for SAD calculation of LDSP and SDSP pattern, and second find minimum SAD from first kernel result. with threads and threadblock configuration we used different types of memory for further optimization. CUDA having different types of memories, to use texture memory for image reading, the image data needs to be bided to a texture. Location of image remains off-chip memory,only a type of memory will change. Texture memory is read-only and cache so first data loaded into texture memory. in block matching algorithm data will be re-use within threadblock, so use of texture give better performance then global memory. off-chip shared memory is faster then global memory. access speed of shared memory is same as registers. here shared memory used when data is re-used within threads in a threadblock. Also, the shared memory can be used (together with a synchronization barrier) to communicate data over threads, as is done for parallel reduction. and shared memory can be used as an extension to the register file, providing more storage space.
Steps for processing

- **CUDAkernel1:** Every threads in threadblock loads its candidate vector from off-chip memory. for candidate vector perform blockmatching with shared memory. then synchronize all threads and continue with one thread. then make summation of all threads and result stored in global memory.
- **CUDAkernel2:** like full search code, make parallel reduction for finding best candidate that will become motion information.

Finally motion information calculated by GPU will copy form DEVICE to HOST.

Chapter 5

Implementation Results

5.1 Test Video and Machine Configuration

We have used some video frames for testing of our system. To measure the timing of CPU and GPU implementation, we have chosen very small video frame and big video frame for testing, still big video frame are useful to get reasonable time value. These video frame are taken from https://trace.eas.asu.edu/yuv/. we perform different type of analysis based on its values. The evaluation of the sample can be done in several ways:

- **The speedup rate analysis :** Each implementation outputs the pure processing timing for input/output and motion estimation. The speedup rate can be measured as the ratio of timings of reference CPU implementation and of CUDA implementation.

- **Consistency Checking:** The consistency checking is the assurance that both CPU and GPU (CUDA) implementations of the same approach produce the same output given the same input. To compare the performance of implementation by considering both estimation quality and computation time, we use, for each test sequence, the average value of the obtained PSNR (given in 3.3.5). we checked for PSNR value of CPU implementation and of CUDA implementation, by using MSU Video Quality Measurement Tool, we achieved approximate same PSNR value of both implementation.

Processor	Intel i3-550
Clock Speed	3.2GHz
System Type	64 bit OS
RAM	6GB
Cores	2
OS	Ubuntu 11.10

Table 5.1: CPU Configuration

Processor	Tesla C2070
CUDA Driver Version	CUDA4.0
Global Memory	4096 MB
Shared Memory per Threadblock	40152Bytes
Number of Thread per Threadblock	1024
GPU Clock Speed	1.15GHz
CUDA Cores	448
Multiprocessors	14(each have 32 cores)
OS	Ubuntu 11.10

Table 5.2: GPU Configuration

Table 5.1 and 5.2 shows CPU and GPU configuration, we used to test application. In this section, we present evaluation results of the proposed CUDA based implementation in comparison with CPU based implementations.

5.2 Speedup Rate Analysis

We have calculated the processing time for input/output and computation time on GPU also compute total processing time including memory transfer time for CPU and GPU version. At least, we have a look at how fast the different parts of the motion estimation are running on different CPU implementation and also on the GPU (CUDA) implementation.

Platform	Execution Time	Speed Up
CPU	3.00s	-
GPU_One Kernel Approach	1.43s	2.1
GPU_Three Kernel Approach	1.32s	2.3

Table 5.3: Performance of Full Search Block Matching Algorithm

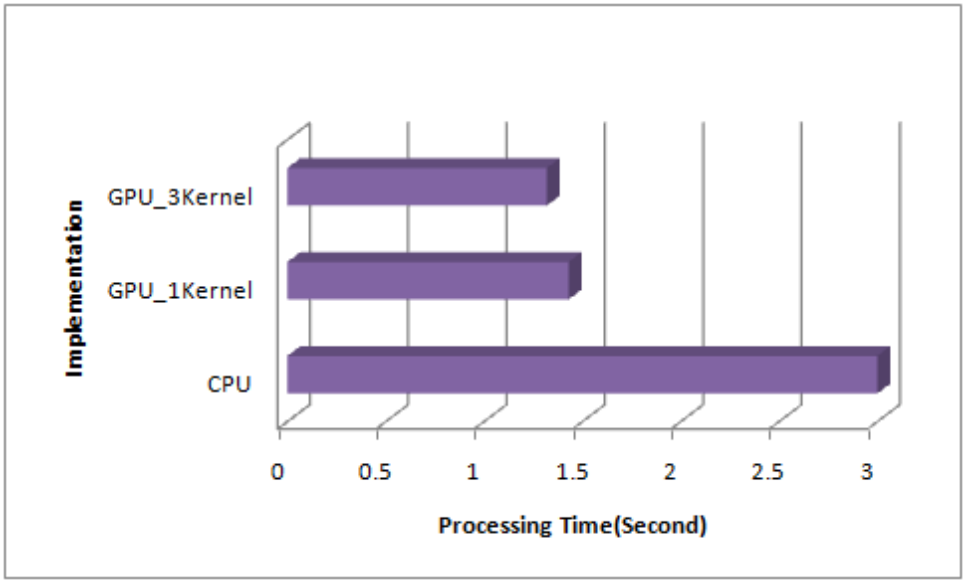

Figure 5.1: Performance Chart of Full Search Block Matching Algorithm

Performance of Full Search Block Matching Algorithm are given on table 5.3. performance is measured with video frame of size 1280×720. Naive sequential implementation of the algorithm is executed on a mid-range CPU, configuration given in table 5.1. naive serial implementation extended with CUDA, the usage of the texture cache, parallel control, the grouping of pixels and the usage of shared memory. parallel CUDA implementation of the algorithm is executed on a high range GPU,configuration given in table 5.2. by using one_kernel approach we achieved 2.1 speed up as compared to CPU. and by using three_kernel approach we achieved 2.3 speed up as compared to CPU.

Platform	Execution Time	Speed Up
CPU	0.180s	-
GPU_Two Kernel Approach	0.055s	3.36

Table 5.4: Performance of Diamond Search Block Matching Algorithm

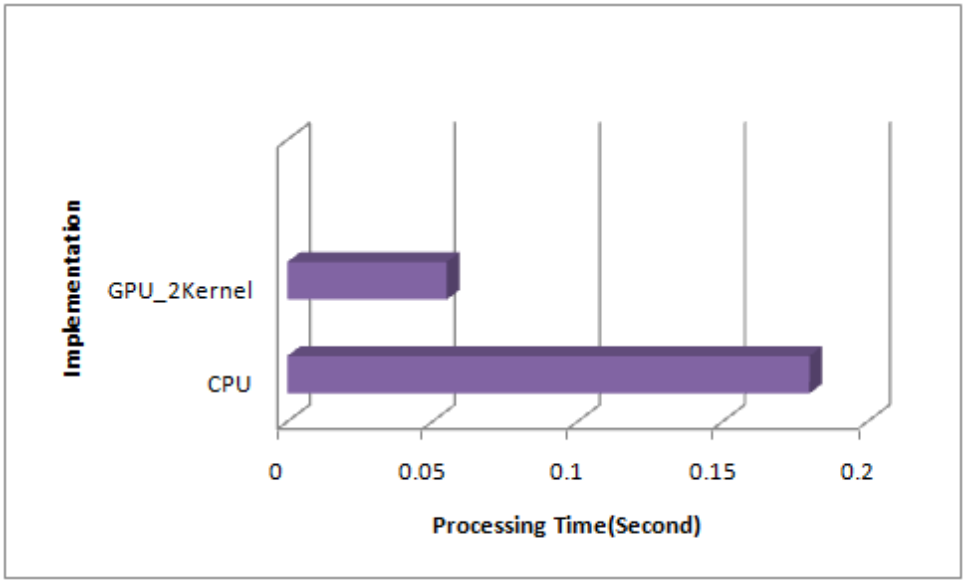

Figure 5.2: Performance Chart of Diamond Search Block Matching Algorithm

Performance of Diamond Search Block Matching Algorithm are given on table 5.4. performance is measured with video frame of size 1280×720. Naive sequential implementation of the algorithm is executed on a mid-range CPU, configuration given in table 5.1. naive serial implementation extended with CUDA, the usage of the texture cache, parallel control, the grouping of pixels and the usage of shared memory. Parallel CUDA implementation of the algorithm is executed on a high range GPU,configuration given in table 5.2. by using two_kernel approach we achieved 3.36 speed up as compared to CPU.

Video Frame Size	CPU Time	GPU Time	Speed Up
1280 × 720	0.185s	0.055s	3.36
512 × 512	0.06s	0.019s	3.15
176 × 144	0.0056s	0.0036s	1.55

Table 5.5: Performance comparison of Diamond Search with different Video Frame size

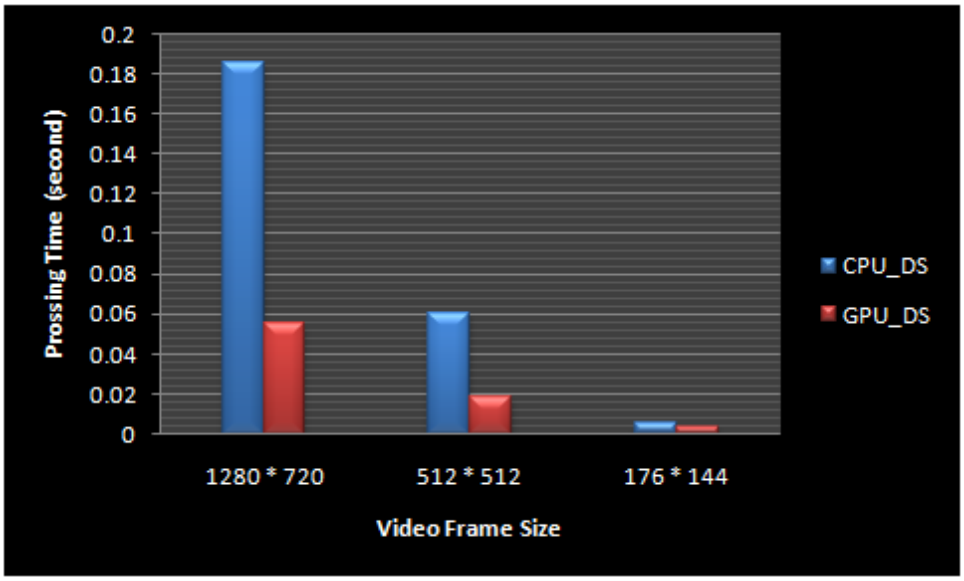

Figure 5.3: Performance chart of Diamond Search with different Video Frame size

Performance of Diamond Search Block Matching Algorithm with different video frame size are given on table 5.5. performance is measured with large video frame of size 1280 × 720, medium video frame size 512 × 512 and small video frame size 176 × 144. with small frame size we achieved 1.55 speed up, while for medium frame size achieved 3.15 and for large video frame size achieved 3.36 speedup.

Chapter 6

Conclusion and Future Scope

6.1 Conclusion

Motion estimation constitutes one of the most time consuming and compute intensive parts of video compression. For motion estimation require lots of SAD calculation and compression. as part of dissertation we implemented Full search and Diamond search block matching algorithms,in full search algorithm whole full-pixel motion estimation for a MB is integrated in a one CUDA kernels and three CUDA kernels, parallel calculating and comparing variable block-size SAD values. While in diamond search algorithm motion estimation for a MB is integrated in a two CUDA kernels. At the same time, this method takes full advantage of high-speed on-chip memory of GPU, such as registers and shared memory to minimize the amount of device DRAM access. Experimental results show that the proposed full search algorithm achieved 2.3 speedup and diamond search algorithm achieved 3.36 speed up as compare to traditional CPU implementation.

6.2 Future Work

In the future we intend to explore other block based motion estimation algorithms like TSS, 4SS, on the GPU architecture using CUDA and compare the portability of all the algorithms on the GPU architecture. also we intend to explore 3D video compression standard in CUDA.

Appendix A

List of Publication

1. Chhaya patel, **Block Matching Algorithm on CUDA Multicore Processors** of the 1st National Conference on Emerging Vistas on Technology in 21st century NCEVT12, organized by Gujarat Technological University, Ahmedabad, India, 14-15 April 2012. published in the special issue of IJTE and ISTE website http://www.isteonline.in/. ISSN No: 0971-3034.

    - Awarded the Best Research Paper at First Position in CSE/IT track.
    - Awarded the Best Presentation during Conference at First Position in CSE/IT track.

2. Chhaya patel, **FullSearch Motion Estimation on Multicore CUDA Architecture** of the 1st National Conference on Advances in Engineering and Technology NCAVT12, organized by Kalol Institute of Technology & Research Centre, Kalol, India, 09-10 March 2012.

    - Awarded the Best Research Paper at First Position in CSE/IT track.

References

[1] Ang, P.H.; Ruetz, P.A.; Auld, D.Video compression makes big gains Spectrum, IEEE Volume 28, Issue 10, Oct. 1991 Page(s):16 - 19

[2] Iain E G Richardson, H.264 / MPEG-4 Video Compression Video Coding for Next-generation Multimedia:The Robert Gordon University, Aberdeen, UK; British Library Cataloguing in Publication Data ;ISBN 0-470-84837-5

[3] Karsten Shring. H.264/AVC Software Coordination. http://iphome.hhi.de/suehring/tml/.

[4] W.-c. Feng, D. Manocha. "High-performance computing using accelerators". Parallel Computing,vol 33, n. 10-11, pp 645-647, November 2007.

[5] NVIDA, CUDA Compute Unified Device Architecture-Programming Guide, Version 4.0, August 2009.

[6] Shane Ryoo , Christopher I. Rodrigues , Sara S. Baghsorkhi , Sam S. Stone , David B. Kirk , Wen-mei W. Hwu, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, February 20-23, 2008, Salt Lake City, UT, USA.

[7] Javier Setoain1, Christian Tenllado1, Manuel Arenaz, and Manuel Prieto1, "Towards Automatic Code Generation for GPU architectures", Computer Architec- ture Group, Department of Electronics and Systems, University of A Coruna, Spain.

[8] B. R. Neha Patil, "Fast and parallel implementation of image processing algo- rithm using cuda technology on gpu hardware", tech. rep., Department of Elec- trical and Computer and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY 12180-3590.

[9] D. L. N. Research, "nvidia gpu architecture & implications", NVIDIA Corporation 2007.

[10] N. P. Karunadasa & D. N. Ranasinghe, "On the Comparative Performance of Parallel Algorithms on Small GPU/CUDA Clusters", University of Colombo School of Computing, Sri Lanka,2008

[11] JAshwin Aji, Mayank Daga, and Wuchun Feng. Bounding the Effect of Partition Camping in GPU Kernels. In ACM International Conference on Computing Frontiers (To appear), 2011.

[12] Francesc Massanes, Marie Cadennes and Jovan G. Brankov, "CUDA implementation of a block-matching algorithm for Multiple GPU cards", Illinois Institute of Technology, Medical Imaging Research Center, Chicago IL 60616, USA

[13] Jiao Liangbao, Zhou Jing, Shu Xiao,"Parallelized Block Match Algorithm on Multicore Processors",International Journal of Advancements in Computing Technology Volume 3, Number 6, July 2011

[14] Y. Lin, P. Li, C. Chang, C. Wu, Y. Tsao, and S. Chien. "Multi-Pass algorithm of motion estimation in video encoding for generic GPU", In Proceedings of IEEE International Symposium.

[15] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S.Stone, David B. Kirk, and Wen-mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda", In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, New York, NY, USA, 2008. ACM., pp. 73-82.

[16] Dishant Ailawadi, Milan Kumar Mohapatra, Ankush Mittal, "Frame-Based Parallelization of MPEG-4 on Compute Unified Device Architecture (CUDA)", Department of Electronics & Computer Engineering, Indian Institute of Technology Roorkee, India.

[17] J. R. Jain, A. K. Jain, Displacement Measurement and Its Application in Interframe Image Coding, IEEE Trans. Communications, vol. COM-29, no. 12, pp. 1799-1808, Dec. 1981.

[18] Y.-C. Lin, S.-C. Tai, Fast Full-Search Block-Matching Algorithm for Motion-Compensated Video Compression, IEEE Trans. Communications, vol. 45, no. 5, pp. 527-531, May 1997.

[19] Yun Q.Shi and Huifang Sun, Image and Video Compression for Multimedia Engineering: Fundamentals, Algorithms, and Standards, CRC press, 2000.

[20] T. Ha, S. Lee, and J. Kim, Motion compnesated frame interpoaltion by new block-based motion estimation algorithm, IEEE Trans. Consum. Electron., vol. 50, no. 2, pp. 752759, May 2004.

[21] B.Liu and A.Zaccarin, New fast algorithms for the estimation of block motion vectors, IEEE Trans. Circuits Syst. Video technology, Vol.3, pp.440445,Dec 1995.

[22] R. Li, B. Zeng, and M. L. Liou, A new three-step search algorithm for block motion estimation, IEEE Trans. Circuits Syst. Video Technol., vol. 4, pp. 438442, Aug. 1994.

[23] S. Zhu and K.-K. Ma, A new diamond search algorithm for fast blockmatchingmotion estimation, in Proc. 1997 Int. Conf. Information Communication and Signal Processing (ICICS), vol. 1, Sept. 9-12, 1997,pp.292-296.

[24] L. M. Po and W. C. Ma, A novel four-step search algorithm for fast block motion estimation,IEEE Trans. Circuits Syst. Video Technol., vol. 6, pp. 313317, June 1996.

[25] A. Barjatya, "Block Matching Algorithms for Motion Estimation", in Technical Report, Utah State University (2004).

[26] F. BENBOUBKER, F. ABDI and A. AHAITOUF, "Shape-Adaptive Motion Estimation Algorithm for MPEG-4 Video Coding", IJCSI International Journal of Computer Science Issues, Vol. 7, Issue 1, No. 2, January 2010

List Of Useful Web-sites

[1] developer.nvidia.com/cuda-downloads

[2] http://ixbtlabs.com/articles3/video/cuda-1-p1.html

[3] http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf

[4] http://www.nvidia.com/object/cuda_get.html http://www.nvidia.com

[5] http://forums.nvidia.com/index.php?showtopic=181472

[6] http://en.wikipedia.org/wiki/Inter_frame

[7] http://gpucoder.livejournal.com/990.html

[8] https://sites.google.com/site/x264cuda/

[9] http://developer.nvidia.com/gpu-computing-webinars

[10] http://en.wikipedia.org/wiki/Inter_frame

[11] http://llpanorama.wordpress.com/2008/05/21/my-first-cuda-program/

[12] http://www.ece.cmu.edu/~ee899/project/deepak_mid.htm

[13] https://github.com/cancan101/h.264-cuda/blob/master/encoder/pyramid/cuda/cuda-me.cu

[14] http://www.r-tutor.com/gpu-computing/cuda-installation/cuda4.0-ubuntu

[15] http://www.ubuntugeek.com/howto-install-nvidia-drivers-manually-on-ubuntu-10-04-l
html

[16] developer.nvidia.com/cuda-toolkit-40

[17] https://trace.eas.asu.edu/yuv/