PARALLELIZATION OF VIDEO ENCODING ALGORITHMS ON MULTI-CORE PROCESSORS

A DISSERTATION

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

of

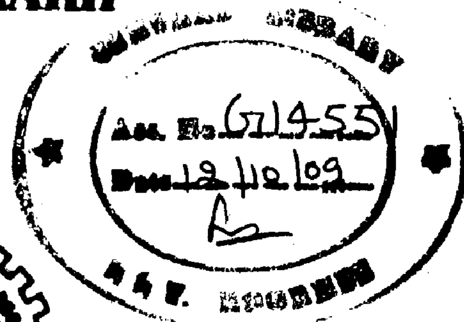
MASTER OF TECHNOLOGY

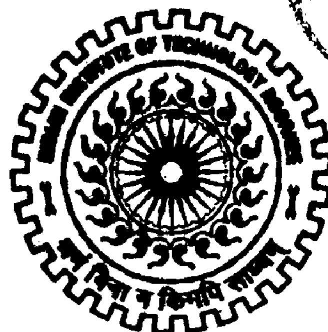
in

COMPUTER SCIENCE AND ENGINEERING

By

NITYAM PARAKH





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

JUNE, 2009

Candidate's Declaration

I hereby declare that the work being presented in the dissertation report titled "**Parallelization of Video Encoding Algorithms on Multicore Processors**" in partial fulfillment of the requirement 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, is an authentic record of my own work carried out under the guidance of Dr. Ankush Mittal, Associate Professor and Dr. Rajdeep Niyogi, Assistant Professor in 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:

Place: IIT Roorkee

(Nityam Parakh)

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.

Dr. Ankush Mittal.
Associate Professor.
Department of Electronics
& Computer Engineering,
IIT Roorkee, Roorkee,
247667 (India).

Dr. Rajdeep Niyogi
Assistant Professor.
Department of Electronics
& Computer Engineering,
IIT Roorkee, Rooekee.
247667 (India)

ACKNOWLEDGEMENTS

I am thankful to Indian Institute of Technology Roorkee for giving me this opportunity. It is my privilege to express thanks and my profound gratitude to my supervisor Dr. Ankush Mittal, Associate Professor for his invaluable guidance and constant motivation throughout the dissertation. I was able to complete this dissertation in time due to the constant motivation and support received from him.

I am also grateful to Dr. Rajdeep Niyogi, Assistant Professor for his continuous encouragement. His valuable help and constant support proved immensely beneficial for my work so did his ability to motivate me. I am grateful to Mr. Khalil Sawant, Mr. Kshitiz Gupta, Mr. Salil Shirish Sahasrabudhe, Mr. Tarun Kumar and Mr. Payas Goyal, my colleagues, for being excellent peers and creating a congenial environment for work. I am also thankful to all my friends who helped me directly and indirectly in completing this dissertation.

Most importantly, I would like to extend my deepest appreciation to my family for their love, encouragement and moral support.

(Nityam Parakh)

ABSTRACT

Scope of high quality videos is not just limited to entertainment industry, but they are used widely in E-learning and health care applications. To reduce the space and bandwidth requirements of these videos MPEG standards are widely used. As the task of encoding videos in MPEG standards is computationally intensive and time consuming, it cannot be achieved in real time. In this thesis, we present parallel implementation of video encoding algorithms by using economical processing model i.e. multicore processors.

In this work we have explained how the IBM Cell B. E. and NVidia CUDA architecture can be exploited to attain a fast video encoding system. In this thesis, we also explain various approaches that can be used to parallelize the MPEG encoder and address various problems faced during implementation.

The encoder discussed using Cell B. E. is a real time MPEG encoder for a frame size up to 384 * 288. The encoding rate of the encoder is above 26 frames per seconds.

Table of Contents

Candidate's declaration........i
Acknowledgements........ii
Abstract........iii
Contents........iv
List of figures........vi
List of tables........vii

1. Introduction 1
   1.1 Motivation 2
   1.2 Problem Statement 2
   1.3 Organization of Report 3

2. Video Encoding Basics 5
   2.1 Evolution of MPEG 5
   2.2 Video Fundamentals 5
   2.3 Bit Rate Reduction Principles 6
   2.4 MPEG – 2 details 9
   2.5 Motion JPEG 11

3. Multi-core Processors 12
   3.1 The CELL B. E. Processor 12
   3.2 CUDA (Compute Unified Device Architecture) 19

4. Parallel Implementation of MPEG 23
   4.1 Overview of MPEG Encoding process 23
   4.2 Parallelization of MPEG Encoder 24

4.3 Porting MPEG Encoder on CELL B. E. 28
4.4 Implementation of MPEG Encoder on CUDA 32
4.5 Implementation of MJPEG Encoder on Cell B. E. 34

5. Results 36
6. Conclusion and Future Work 40
References 41

List of Figures

Fig.2.1 Frame dependencies in MPEG encoding....................10
Fig 2.2 Organization of frames in a GOP....................10
Fig 3.1 Architecture of the Cell B. E. Processor....................13
Fig 3.2 Implementation of double buffering....................18
Fig 3.3 Transistor division in CPU and GPU....................20
Fig 3.4 GPU architecture....................21
Fig 4.1 MPEG encoder....................23
Fig 4.2 Profiling of MPEG-2 encoder....................24
Fig 4.3 Architecture of Parallel MPEG encoder....................25
Fig 4.4 Architecture implementing frame level parallelism....................26
Fig 4.5 Motion estimation in MPEG encoding....................27
Fig 4.6 Implementation of MPEG encoder on Cell B. E.....................29
Fig 4.7 Implementation of motion estimation routine using GPUs....................33
Fig 4.8 MJPEG frame encoding....................34
Fig 4.9 MJPEG parallel encoding architecture....................35
Fig 5.1 Comparison between performances of encoder over various platforms.......39

List of Tables

Table 5.1 Comparison between Intel processor and Cell processor......36
Table 5.2 Comparison between Intel processor and Cell processor for MPEG encoder......37
Table 5.3 Comparison between Intel processor and Cell processor for MJPEG encoder......37
Table 5.4 Comparison for motion estimation routine between GPU using and non GPU using Intel machine......38
Table 5.5 Comparison between GPU using and non GPU using Intel machine......38
Table 5.6 Comparison between GPU and Cell B. E. machine......39

Multimedia applications have changed the way computers were used. People prefer using these applications as they empower them to present their views in a variety of formats. They also offer them the flexibility to address different types of audiences in addition to their being easy to use. Recent advancements in the field of digital imaging and video compression have enabled people to use this highly effective means of communication in a more cost effective way. Digital video technology has its scope not only in the entertainment industry but also in areas such as health care and e-learning having the potential to bring a revolutionary change in the way these services are offered presently. However, high quality video requires more storage space and communication bandwidth than traditional data [1]. This fact alone act as a bottleneck which has prevented the widespread use of video technology in different fields.

To deal with this problem, most of the digital video encoding techniques use a compression scheme [2]. The MPEG committee has defined widely used standards for digital video encoding that provides high quality images and high compression rates. MPEG encoding, however, also demands high computational power. As a result, the compression of such data on traditional sequential machines requires lot of time. Thus, various applications that require the video to be compressed at a fast rate either cannot be deployed altogether, or if deployed are unable to yield satisfactory performance. There is thus a need to come up with methods or techniques which can render faster video compression to meet the demand posed by these applications.

Video compression techniques can either be hardware-based or software-based [2]. A software solution is more flexible, and thus allows algorithmic improvements. However, there are various hardware devices that can be used for this purpose, but they are far expensive besides being obsolete. Although there are other approaches of achieving a fast encoding, like using multiprocessing systems, grids etc. but, these options introduce problems such as communication overhead, besides being expansive.

Thus, we focus on addressing the problem and generating an efficient and cost effective solution by the use of new trend growing in the market, i.e. the multicore processors. These processors have multiple computing cores that work simultaneously. The clock frequency of the cores might be less but the overall throughput that can be generated by using the cores properly is quite high. These processors are comparatively cheap and easily affordable. The performance factor that can be achieved by using such a parallel processing model depends more on how different processing units are used i.e. quality of the software is responsible to draw maximum performance from the architecture.

In order to attain maximum performance from multicore processor we need to develop architecture centric applications. In this work, we discuss how video compression can be implemented on Cell B.E. and CUDA architecture. We have achieved significant performance improvement in MPEG 2 and MJPEG compression routine on Cell B.E. processor and GPUs. The focus of the work is to parallelize the encoder in such a way that it suits the underlying architecture, as there are different programming and implementation constraints with these architectures which restricts simple implementation of applications.

1.1 Motivation

Video compression may vastly enhance the bandwidth utilization over a network. In most of the developing countries, network bandwidth is a huge obstacle in deploying applications that require transmission of high quality video. Even if networking resources are improved to transmit compressed video, real time video encoding is a major challenge. Solutions that are available in the market are either quite expansive or they require networking infrastructure. Another problem with hardware solutions is that they become worthless with a slight change in standard. Although there is a need for software based video encoder that uses a low cost resource and can achieve video encoding at a better rate, yet not much have been done in this field after 1998. All the system developed in past 10 years capable of fast video encoding uses special hardware for this purpose.

Multicore architecture seems to be a good option for porting such applications. When comparison is made on basis of cost multicore systems are cheaper than multiprocessing systems or even grids. They consume less power and can be used for a wide variety of applications. CUDA devices can be used as add on in most of existing systems and can perform various task. Thus, to make use of such devices and boost up performance of multimedia applications is highly beneficial.

1.2 Problem Statement

In this dissertation work we propose and implement a model that improves a computationally intensive application i.e. video encoders system using parallel processing architectures such as Cell Broadband Engine and NVidia Graphics Processing Units. The objective of the model is to achieve a significant improvement in performance over the linear version.

1.3 Organization of the Report

This dissertation proposes a model for implementing Video Encoding model on Multicore architectures. The organization of the report is as follows:

Chapter 2 discusses the background of MPEG-2 and MJPEG standards and their evolution, working principle and bit rate reduction process.

Chapter 3 discusses the hardware architecture of the STI Cell Broadband Engine and CUDA. It also describes the usage, advantages, constraints and limitations of these platforms.

Chapter 4 describes parallel implementation of MPEG and MJPEG encoders on different multicore architectures and discusses different approaches in their parallelization. We also address issues and challenges in these approaches and how various problems can be dealt with in such application. A novel method of parallelization of Motion Estimation routine in MPEG encoding is also described in this chapter.

Chapter 5 compares the performance of the MPEG encoder on Cell Broadband Engine, and NVidia GTX 280 card, it also presents the performance improvement of MJPEG encoder on Cell processor.

Chapter 6 concludes the dissertation work and gives suggestions for future work.

2.1 Evolution of MPEG

With growing rate in technology usage compressed video has created a widespread scope. Thus, a need of standardization was very important to allow the use of common compression methods in new services. Standardization not only facilitates the development of new products and services but also it creates a platform where different services can interoperate. To establish digital video standards MPEG (Moving Picture Experts Group) was started in 1988. MPEG's first project was MPEG-1 and was published in 1993. The standard itself can be divided in to three parts defining audio and video compression methods and multiplexing techniques so that they can be played together.

A need for another standard was felt by MPEG group in order to encode formats of higher data rate as MPEG1 was limited to 1.5 Mbps. The MPEG-2 standard [2] is capable of coding standard-definition television at bit rates from about 3-15 Mbit/s and high-definition television at 15-30 Mbit/s. MPEG-2 aims to be a generic video coding system supporting a diverse range of applications. To implement all the features of the standard in all decoders is unnecessarily complex and a waste of bandwidth, so a small number of subsets of the full standard, known as profiles and levels, have been defined. A profile is a subset of algorithmic tools and a level identifies a set of constraints on parameter values (such as picture size and bit rate). A decoder which supports a particular profile and level is only required to support the corresponding subset of the full standard and set of parameter constraints.

2.2. Video Fundamentals

A video stream is a sequence of video frames. Each frame is a still image. A video player displays one frame after another, usually at a rate close to 30 frames per second. Frames are digitized in a standard RGB format, 24 bits per pixel (8 bits each for Red, Green, and Blue). The MPEG algorithm operates on images represented in YUV color space (Y Cr Cb). If an image is stored in RGB format, it must first be converted to YUV format.

Television services in Europe currently broadcast video at a frame rate of 25 Hz. Each frame consists of two interlaced fields, giving a field rate of 50 Hz. American television is similarly interlaced but with a frame rate of just 30 Hz. In video systems other than television, non-interlaced video is commonplace (for example, most computers output non-interlaced video). In non-interlaced video, all the lines of a frame are sampled at the same instant in time. Non-interlaced video is also termed 'progressively scanned' or 'sequentially scanned' video. The red, green and blue (RGB) signals coming from a color television camera can be equivalently expressed as luminance (Y) and chrominance (UV) components. The chrominance bandwidth may be reduced relative to the luminance without significantly affecting the picture quality. For standard definition video, MPEG-2 defines how the component (YUV) video signals can be sampled and digitized to form discrete pixels. Using 8 bits for each Y, U or V pixel, the uncompressed bit rates for 4:2:2 and 4:2:0 signals are therefore:

4:2:2: 720 x 576 x 25 x 8 + 360 x 576 x 25 x (8 + 8) = 166 Mbit/s

4:2:0: 720 x 576 x 25 x 8 + 360 x 288 x 25 x (8 + 8) = 124 Mbit/s

MPEG-2 is capable of compressing the bit rate of standard-definition 4:2:0 video down to about 3-15 Mbit/s. At the lower bit rates in this range, the impairments introduced by the MPEG-2 coding and decoding process become increasingly objectionable. For digital terrestrial television broadcasting of standard-definition video, a bit rate of around 6 Mbit/s is thought to be a good compromise between picture quality and transmission bandwidth efficiency.

2.3 Bit Rate Reduction Principles

A bit rate reduction system works on the principle of removing redundant information from the frames at the coder before transmission and re-inserting it at the decoder. A coder and decoder pair is referred to as a 'codec'. In video signals, two distinct kinds of redundancy can be identified.

2.3.1 Spatial and Temporal Redundancy: Pixel values are not independent, but are correlated with their neighbour both within the same frame and across frames. So, to some extent, the value of a pixel is predictable given the values of neighbouring pixels.

2.3.2 Psycho Visual Redundancy: The human eye has a limited response to fine spatial detail, and is less sensitive to detail near object edges or around shot-changes. Thus, a little distortion introduced into the decoded picture by the bit rate reduction process should not be visible to a human observer.

Two key techniques employed in an MPEG codec are intra-frame Discrete Cosine Transform (DCT) coding and motion-compensated inter-frame prediction.

2.3.3 Intra-Frame DCT Coding

DCT is performed on small blocks (16 pixels * 16 lines) of each component of the picture to produce blocks of DCT coefficients. The magnitude of each DCT coefficient indicates the contribution of a particular combination of horizontal and vertical spatial frequencies to the original picture block [4]. The coefficient corresponding to zero horizontal and vertical frequency is called the DCT coefficient. Equation 2.1 is used to calculate DCT of a block.

$$F(x,y) = \frac{2}{N} C(u)C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) \cos\left(\frac{(2x+1)u\pi}{2N}\right) \cos\left(\frac{(2x+1)v\pi}{2N}\right)$$

$$C(u), C(v) = \begin{cases} \frac{1}{\sqrt{2}} \; for\, u,v = 0 \\ 1 \; otherwise \end{cases}$$

Eq 2.1

The DCT doesn't directly reduce the number of bits required to represent the block. In fact for an 8x8 block of 8 bit pixels, the DCT produces an 8x8 block of 11 bit coefficients (the range of coefficient values is larger than the range of pixel values.) The reduction in the number of bits follows from the observation that, for typical blocks from natural images, the distribution of coefficients is non-uniform. The transform tends to concentrate the energy into the low-frequency coefficients and many of the other coefficients are near-zero. The bit rate reduction is achieved by not transmitting the near-zero coefficients and by quantizing and coding the remaining coefficients as described below. The non-uniform coefficient distribution is a result of the spatial redundancy present in the original image block. MPEG-2 video compression distribution is a result of the spatial redundancy present in the original image block.

2.3.4 Quantization

The function of the coder is to transmit the DCT block to the decoder, in a bit rate efficient manner, so that it can perform the inverse transform to reconstruct the image. It has been observed that the numerical precision of the DCT coefficients may be reduced while still maintaining good image quality at the decoder. Quantization is used to reduce the number of possible values to be transmitted, reducing the required number of bits. The degree of quantization applied to each coefficient is weighted according to the visibility of the resulting quantization noise to a human observer. In practice, this results in the high-frequency coefficients being more coarsely quantized than the low-frequency coefficients. The quantization noise introduced by the coder is not reversible in the decoder, making the coding and decoding process 'lossy'.

2.3.5 Motion Compensated Inter-Frame Prediction

This technique makes use of temporal redundancy by attempting to predict the frame to be coded from a previous 'reference' frame [4]. The prediction cannot be based on a source picture because the prediction has to be repeatable in the decoder, where the source pictures are not available. Consequently, the coder contains a local decoder which reconstructs pictures exactly as they would be in the decoder, from which predictions can be formed. The simplest inter-frame prediction of the block being coded is that which takes the co-sited (i.e. the same spatial position) block from the reference picture. Naturally this makes a good prediction for stationary regions of the image, but is poor in moving areas.

A more sophisticated method, known as motion-compensated inter-frame prediction, is to offset any translational motion which has occurred between the block being coded and the reference frame and to use a shifted block from the reference frame as the prediction. One method of determining the motion that has occurred between the block being coded and the reference frame is a 'block-matching' search in which a large number of trial offsets are tested by the coder using the luminance component of the picture. The 'best' offset is selected on the basis of minimum number of errors between the block being coded and the prediction. The bit rate overhead of using motion-compensated prediction is the need to convey the motion vectors required to predict each block to the decoder. For example, using MPEG-2 to compress standard

definition video to 6 Mbit/s, the motion vector overhead could account for about 2 Mbit/s during a picture making heavy use of motion-compensated prediction.

2.4 MPEG-2 Details

2.4.1 Picture Types

In MPEG-2, three 'picture types' are defined. The picture type defines which prediction modes may be used to code each block. 'Intra' pictures (I-pictures) are coded without reference to other pictures. Moderate compression is achieved by reducing spatial redundancy, but not temporal redundancy. They can be used periodically to provide access points in the bit stream where decoding can begin.

'Predictive' pictures (P-pictures) can use the previous I- or P- picture for motion compensation and may be used as a reference for further prediction. Each block in a P-picture can either be predicted or intra-coded. By reducing spatial and temporal redundancy, P-pictures offer increased compression compared to I-pictures.

'Bi-directionally-predictive' pictures (B-pictures) can use the previous and next I- or P-pictures for motion-compensation, and offer the highest degree of compression. Each block in a B-picture can be forward, backward or bi-directionally predicted or intra-coded [5]. To enable backward prediction from a future frame, the coder reorders the pictures from natural 'display' order to 'bit stream' order so that the B-picture is transmitted after the previous and next pictures it references. This introduces a reordering delay dependent on the number of consecutive B-pictures. The different picture types typically occur in a repeating sequence, termed a 'Group of Pictures' or GOP.

A typical GOP in display order is:
B1 B2 I3 B4 B5 P6 B7 B8 P9 B10 B11 P12

The corresponding bit stream order is:
I3 B1 B2 P6 B4 B5 P9 B7 B8 P12 B10 B11

A regular GOP structure can be described with two parameters: *N*, which is the number of pictures in the GOP, and *M*, which is the spacing of P-pictures. The GOP given here is described as *N*=12 and *M*=3. MPEG-2 does not insist on a regular GOP structure. For example, a P-picture following a shot-change may be badly predicted since the reference picture for prediction is completely different from the picture being predicted. Thus, it may be beneficial to code it as an I-picture instead.

2.4.2 Frame Dependencies in a GOP

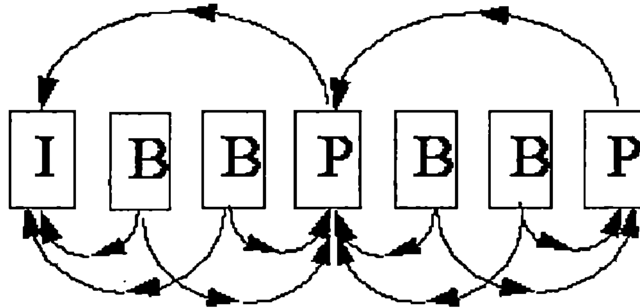


Figure 2.1 Frame dependencies in MPEG encoding

Figure 2.1 shows the dependencies among various types of frames in a GOP. The arrows represent the inter-frame dependencies. Frames do not need to follow a static IPB pattern. Each individual frame can be of any type. Often, however, a fixed IPB sequence is used throughout the entire video stream for simplicity. I frame is encoded independently, whereas P frames are encoded on basis of previous I and P frame. B frames are encoded on basis of previous and next P frame and I frame.

2.4.3 Frame Structure in a GOP

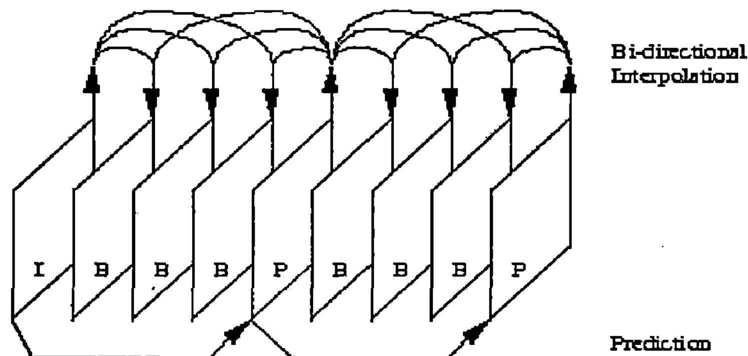


Figure 2.2 Organization of frames in a GOP

Figure 2.2 shows a typical GOP giving a basic idea of how frames are sequenced in a GOP. Each GOP starts with I frame then a fixed number of B frames are inserted followed by a P frame. This structure is repeated again and again but instead of I frame P frame is inserted. There is only one I frame in a GOP.

2.5 Motion JPEG

MPEG standards are quite optimistic when it comes to reduction of video file size, but if there is a requirement for real time video transmission over an unreliable network the standard will not hold its utility as loss in a single reference frame will pause the playback of complete GOP. Also, a real time processing of MPEG videos is very difficult as MPEG decoding is also a computationally intensive process.

To overcome these problems a new standard using intra-frame coding technology that is very similar in technology to the I-frame part of MPEG standard, but does not use inter-frame prediction. The lack of use of inter-frame prediction results in a loss of compression capability, but eases video editing, since simple edits can be performed at any frame when all frames are I-frames [6]. Video coding formats such as MPEG-2 can also be used in such an I-frame only fashion to provide similar compression capability and similar ease of editing features. As each frame is an I frame any packet loss in transmission will not hamper the playback of the whole video.

Using only intra-frame coding technology also makes the degree of compression capability independent of the amount of motion in the scene, since temporal prediction is not being used. However, although the bit rate of Motion JPEG is substantially better than completely uncompressed video, it is substantially worse than that of video standards which use inter-frame motion compensation.

In this thesis we also implement a model capable of encoding raw frames to Motion JPEG format using parallel Cell B. E. architecture.

Multi-core technology is a reality of today. The era of the single processor system has passed; multi-core is real as applications can no longer count on increased processor clock speeds to improve performance. Multicore architecture, as said before is the solution for the ever growing demand of more and more computational power. As the trend is growing in the market more and more varieties of multicore chips are developed. Examples of some such multicore architecture are

- Intel Dual core, Xeon quad core
- Cell B. E.
- NVidia CUDA

In this work we have used Cell B. E. and NVidia CUDA architecture to attain performance improvement over sequential machines. Let us see these architecture in detail.

3.1 The Cell B. E. Processor

The Cell architecture addresses three major bottlenecks of modern microprocessors, namely power wall, memory wall, and frequency wall as discussed in [7]. The Cell broadband engine architecture addresses these problems by heterogeneous multiprocessing. The cell architecture has one Power Processor Element (PPE) and eight Synergistic Processor Elements (SPEs). The PPE is specialized for control-intensive codes while the SPEs are specialized for computationally intensive codes. The SPEs have very simple hardware logic and not use complicated logic for branch prediction, out-of-order execution and register renaming. Therefore the SPEs can operate at higher frequency without much of power dissipation. The SPE area is only 14.5 mm^2 and dissipates only a few Watts even when operating at 3.2 GHz [7]. The Cell architecture addresses the problem of memory wall by making programmers manage the SPEs local memory using explicit direct memory access (DMA) transfers. Each SPE can give a request for 16 DMA transfers simultaneously, and hence a total of 128 simultaneous transfers could be issued between the SPEs and main memory. This asynchronous DMA transfer can be used by programmers to overlap memory latency time with computation. Memory latency in the SPEs is further reduced by

larger register size. The following section describes the different components of the Cell broadband engine architecture.

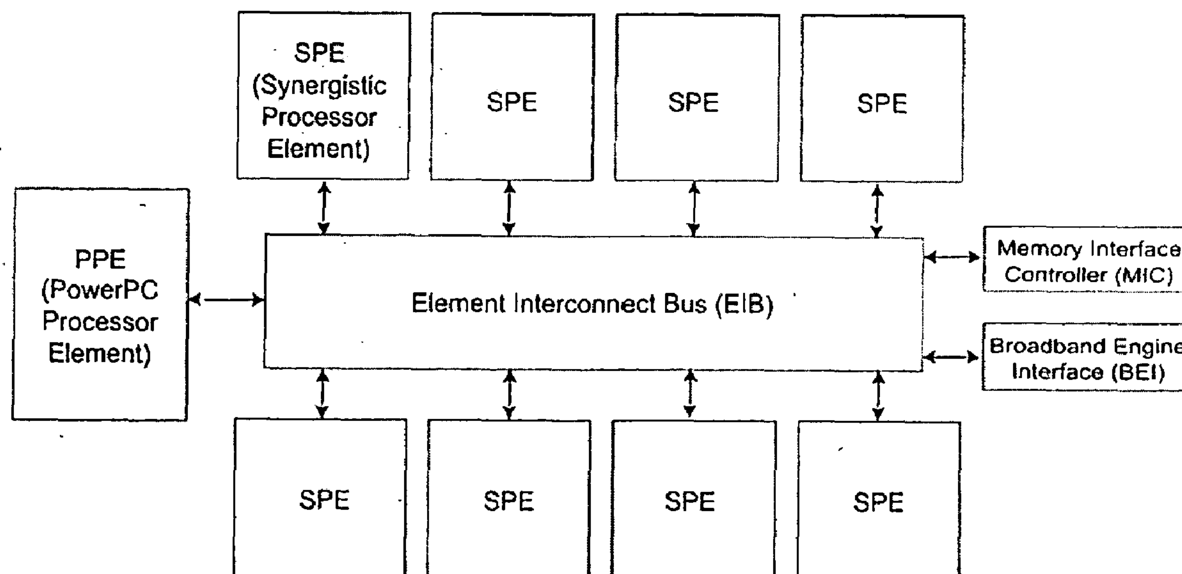


Figure 3.1 Architecture of Cell B. E. processor

3.1.1 Power Processing Element (PPE)

The PPE implements a 64-bit, dual-thread PowerPC architecture. In addition to the general PowerPC floating point units (FPU), the PPE is also equipped with a small SIMD engine to perform fast vector operations. The PPE's register set includes 32 64-bit general purpose registers, 32 64-bit floating-point registers and 32 128-bit vector registers. The PPE operates at 3.2 GHz and can perform four matrix-add operations in a single clock cycle resulting in 8 floating point operations per cycle delivering a peak performance of 25.6 GFLOPS. The PPE is mostly used for running the operating system and for managing the SPE threads and system resources. The primary function of the PPE is the management and allocation of tasks for SPEs in a system. When data enters the PPE, this element then distributes it among SPEs, schedules them to be processed on one or more of the SPEs, controls and synchronizes them. PPE can also handle task of user interaction, resource utilization etc. Mailboxes and signals are the two mechanisms that can be used for synchronization of the PPE and the SPEs. Mailboxes and signals are explained below.

3.1.2 Synergistic Processing Elements (SPEs)

The SPEs are single-instruction, multiple-data (SIMD) processor elements. Each SPE contains a 256 KB software-controlled private memory referred to as the local store

(LS) and is shared between instructions and data. The local store memory is managed by software through explicit DMA transfers. The SPEs register set includes a 128-bit, 128-entry unified registers. The SPEs implements a new instruction set architecture (ISA) optimized for power and high performance on compute-intensive applications. These instructions can be performed on 128 bits vectors of eight floating-point values. The vector registers can even contain variables of 8, 16, 32 and 64 bits and thus can operate simultaneously on two 64-bit double precision values, four 32-bit single precision values, eight 16-bit integer values or sixteen 8-bit character values. The SPE does not have separate register support for scalar operands or addresses and thus the scalar operations are executed by pushing the scalar values to the preferred slot in the vector register. The SPEs can execute four fused multiply-add (single-precision) instructions per cycle resulting in eight floating point operations per cycle. Operating at 3.2 GHz, the SPE delivers a maximum performance of 25.6 GFLOPS. With eight processors the peak performance of the Cell processor is 204.8 GFLOPS. Techniques to get maximum performance from the SPEs will be discussed in the next sections.

3.1.3 Memory Flow Controller

Each SPE can execute code residing in its local store and operate on data residing in its local store. Explicit DMA commands are needed to transfer data and code between the local store and the main memory. Other than this, the SPEs need to communicate with other processing units and the I/O controllers to do synchronization. Both of these purposes are fulfilled by the memory flow controller (MFC). Each SPE is connected to the main memory, other processing units and the I/O devices by means of the element interconnect bus (EIB) through its MFC. The main function of the MFC is to interface the SPEs' local store with the main memory. The SPEs communicate explicitly with other entities (main memory, I/O Device, other SPEs and PPE) in the system using the following three primary communication mechanisms provided by the MFC of an SPE. An MFC supports naturally aligned DMA transfers with sizes 1, 2, 4, 8, and 16 bytes and multiples of 16 bytes.

3.1.3.1 DMA Transfers

DMA transfers are used to move data and instructions between main memory and local stores. Transfers can be initiated by the SPEs, or the PPE. SPE-initiated DMA is fastest and is mostly used. DMA transfers can be done in two ways. First, DMA

transfers move up to 16 KB of data between a local store of SPE and continuous address space of main memory. Second, a DMA list is a sequence of eight-byte list elements, stored in an SPE's local store, each of which describes a DMA transfer. Each entry in the DMA list species the main memory address space location and the amount of data to be transferred. A DMA list can have 2048 entries with each entry capable of issuing a transfer command of 16 KB in length. Thus, 2048 * 16KB = 32MB of data can be transferred through a DMA list, which is much larger than the local store size of 256KB. One of the biggest advantages of DMA list is that the data can be gathered from different address regions in the main memory and can be accumulated in a continuous address range in the local store.

3.1.3.2 Mailboxes

Mailboxes are queues which are used to send and buffer 32-bit messages between an SPE and other processing units. Each SPE has 3 mailbox registers connected through SPE's MFC. Two one-entry mailboxes namely, SPU Write Outbound Mailbox and SPU Write Outbound Interrupt Mailbox, is used for sending messages from the SPEs to other processing unit. One four entry SPU Read Inbound Mailbox is used for sending messages from PPE or other SPEs to an SPE.

3.1.3.3 Signal Notification

An SPE has two 32-bit signal-notification registers, each of which has the corresponding MMIO (Memory mapped I/O) registers that can be written with signal notification data by all SPEs and the PPE. There are two SPE signal-notification channels, namely, SPE Signal Notification 1 and SPE Signal Notification 2, corresponding to each of the signal notification registers. An SPE's read operation from a signal-notification channel will be stalled if no signal is pending at the time of read until a signal is sent. Reading the pending signals automatically resets the register to 0. Signal-notification registers can operate either in OR mode or overwrite mode. In OR mode a new value added is ORed with the old one while in overwrite mode, the old value is overwritten.

3.1.4 Element Interconnect Bus (EIB)

All the processing units, the memory controllers and the I/O controllers are connected to each other with the coherent on-chip element interconnect bus (EIB). EIB operates

at 1.6 GHz. The EIB consists of four 16-byte-wide unidirectional data rings (two on either direction). Each ring transfers 128 bytes (one PPE cache line) at a time. Each ring at maximum concurrency can have three active transactions, which result in EIB's internal maximum bandwidth of 96-bytes per processor-clock cycle. Each processor element and I/O controller is connected to the EIB with one 16 byte read port and one 16 byte write port. SPEs can compute and receive data simultaneously.

3.1.5 Memory Interface Controller and Interface Unit

The memory interface controller provides the interface between EIB and main memory. The IBM Blade Center supports one or two RAMBUS XDR memory interface. Two 32-bit channels together over a data transfer bandwidth of 25.6 GB/s. Memory accesses on each interface are of size 1 to 8, 16, 32, 64, or 128 bytes. A maximum of 64 reads and 64 writes can be queued.

3.1.6 Optimization Techniques

The Cell architecture, as explained above, can offer a maximum of 204.6 GLOPS. One of the biggest advantages of the Cell architecture is that it is very much possible to get close to the maximum performance offered by the Cell. However, to get good performance on the Cell, one has to be aware of the architectural details of Cell processor. This chapter describes some of the optimization techniques that should be used to design and implement high-performance algorithms on the cell.

3.1.6.1 Huge Pages

If the program uses large datasets, it is recommended to use huge pages to avoid TLB misses. In the SPEs TLB misses occur when the DMA transfer involves a page that is not present in the TLB. If huge pages are used, they will never be paged-out of the TLB. DMA transfers involving an address in the huge pages will enjoy TLB hits and will decrease the DMA time significantly.

3.1.6.2 Uniform Memory Access across Memory Banks

If accesses to the main memory banks are not uniformly distributed a large number of conflicts can lead to very poor performance of the DMA operations. The Cell processor's memory subsystem has 16 banks and the banks are interleaved at the cache block boundaries of size 128 bytes. Addresses that are 2 KB apart access the

same bank and hence if accessed simultaneously will result in bank conflicts. To reduce this problem, all the memory banks should be used uniformly so that bank conflict bottleneck can be reduced to increase performance.

3.1.7 SPE Optimization Techniques

The Cell's computational power resides in the SPEs. The end-performance of any application on the Cell processor is determined by the efficiency of the program executing on the SPEs. To efficiently utilize the power of the SPEs, it is required to understand the strength and weakness of the SPEs. In the following, we discuss some of the optimization techniques for SPE programming.

3.1.7.1 SIMD Programming

SIMD stands for single instruction multiple data. The SPEs are vector processors and have very efficient SIMD engines. The SPEs can execute four single precision floating-point operations in one cycle. SPE instruction set also introduces vector operations which perform two operations in single cycle. For example multiplication of two vectors and its addition with the third can be done by a single instruction. But, the SPEs are not as efficient in performing scalar operations. Scalar operations are executed by shifting scalar values to the preferred slot in the vector and then shifting it back to its original location after the operation is completed, which introduce costly overhead. Therefore, our goal while programming the Cell should be to eliminate as many scalar operations as possible and replace them by vector operations.

3.1.7.2 Loop Unrolling

In SPEs, branches are very expensive, and when mispredicted, results in a loss of 18 cycles. The branches can be reduced by unrolling loops. By unrolling, a long stretch of instructions can be executed on the SPEs without any branch instruction. The SPEs have sufficient number of registers to allow deep unrolling.

3.1.7.3 Efficient Utilization of Local Store Memory

One of the biggest challenges of Cell programming is due to the limited local store memory. The local store LS is only 256 KB, shared between code and data. Optimization techniques discussed like loop unrolling and inline functions increases the code size significantly. With small memory available it may not be possible to do

all the optimizations. Code partitioning techniques as proposed in [8] can be used to reduce the impact of local store limitations by partitioning the code in multiple partitions. Partitions are created such that the SPEs execution could be possible by keeping a small set of partitions in the SPE, keeping the rest of the partitions in the main memory. Programmers can explicitly specify the program partitioning strategy using code overlays.

3.1.7.4 Double Buffering

Figure 3.2 shows how double and triple buffering are implemented for a simple problem.

Computation	Communication
No Computation	Initialize three Buffers Read Data in Buffer[1]
Process Buffer[1] i = 2	Read Data in Buffer[2]
if (i<N)	
Process Buffer[i mod 3] i = i+1	Write Buffer [(i-1) mod 3] Read Data in Buffer[(i+1)mod3]
Process Buffer[N mod 3] i = i+1	Write Buffer [(i-1) mod 3]
No Computation	Write Buffer[N mod 3]

Figure 3.2 Implementation of double buffering

Using the asynchronous DMA transfers, the SPEs can inject outstanding DMA transfer commands. The SPEs can receive data from main memory and at the same time can do computation thus overlapping communication time with the computation. Double buffer is generally needed when data is only read from main memory. In double buffering while one buffer of data is read from the memory, the other buffer is used for computation. But if the buffer of data being read also needs to be written

back to the memory, triple buffering is used. In triple buffering while the ith buffer is being used for computation, the $(i - 1)^{th}$ buffer is written back into the memory and the $(i+1)^{th}$ buffer is read from the main memory.

3.2 CUDA (Compute Unified Device Architecture)

NVidia is a name known mainly for developing powerful graphics cards. These cards are used as add on computational device in desktop computing. Primarily the focus of such cards was to provide accelerated graphics to end users by using GPUs or Graphics Processing Units. As the use of computer has been redefined and the demand for processing capabilities has raised and hardware industries have shifted to multiple processors on a chip, the power of Graphics Cards have also been raised by similar method.

Modern graphics cards are built with multiple processing units. The number of cores is not limited in order of 10s but they have crossed the order of 100s. This scenario gives rise to a highly capable parallel processing unit having hundreds of cores working simultaneously. Having such a powerful device is not enough to achieve performance improvement as the challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to many core GPUs with widely varying numbers of cores [9].

CUDA is a parallel programming model and software environment designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C.

3.2.1 CPU versus GPU

GPUs are highly parallel, multithreaded multicore processor with tremendous computing power and high memory bandwidth. They are designed to handle computationally intensive highly parallel task. Thus, it contains more dedicated registers for data processing rather than data caching and flow control. More specifically, the GPU is well suited to address problems that can be expressed as data parallel computations [9]. Multiple threads are created that works on different data elements in parallel. As the same work has to done on different data there is a lower

requirement for complicated flow control. Thus more registers can be dedicated to data processing. Let us compare the number of dedicated transistors in a GPU and a CPU. This comparison is shown in figure 4.3.

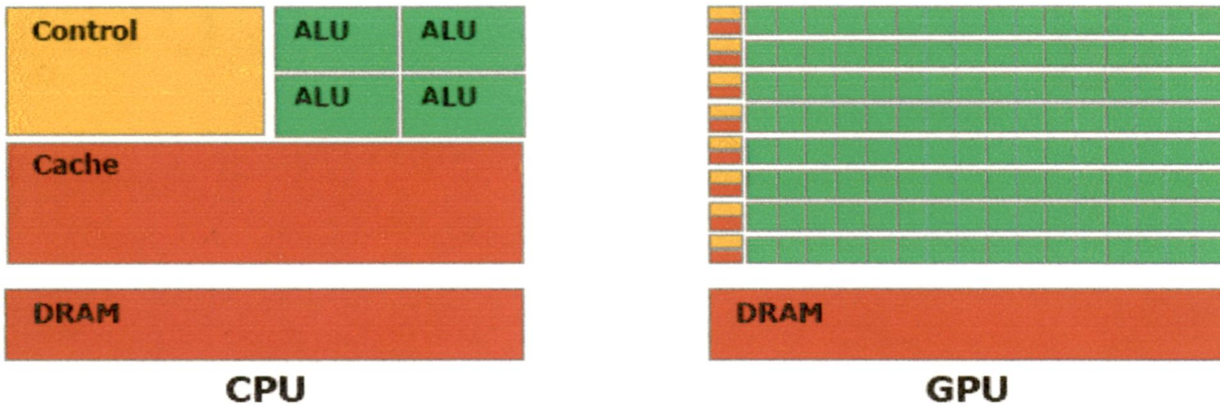
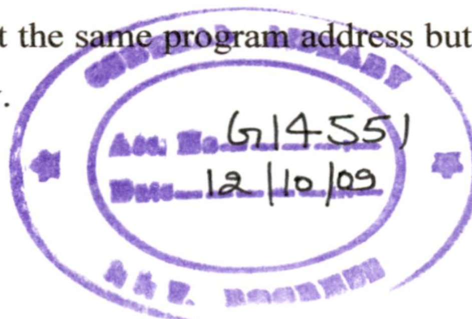


Figure 3.3 Transistor division in CPU and GPU

3.2.2 GPU architecture

A GPU consists of a number of multiprocessors each containing 8 cores each. The multiprocessor creates, manages, and executes concurrent threads in hardware with zero scheduling overhead.

To manage hundreds of threads running several different programs, the multiprocessor employs a new architecture we call SIMT (single-instruction, multiple-thread). The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state. The multiprocessor SIMT unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a SIMT warp start together at the same program address but are otherwise free to branch and execute independently.



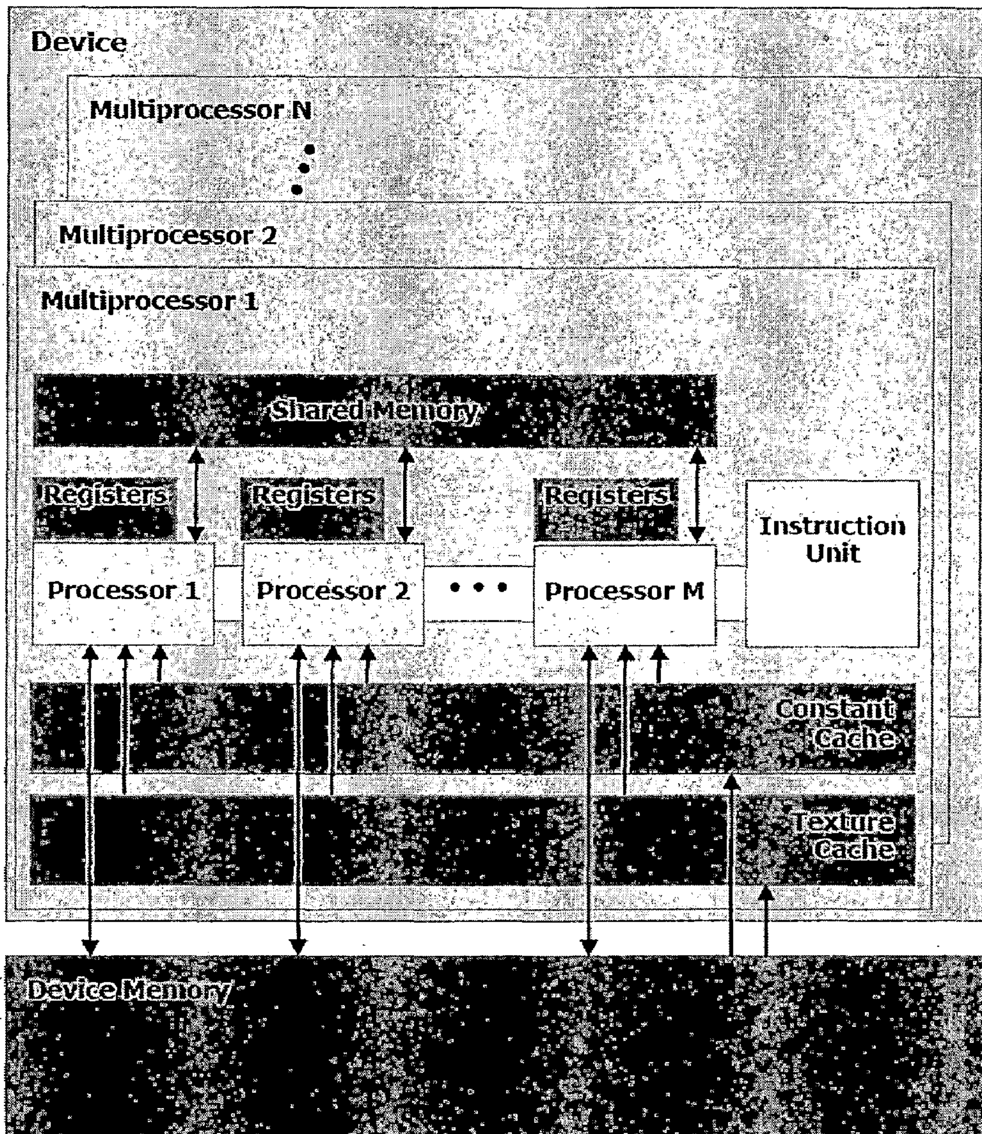


Figure 3.4 GPU architecture

When a multiprocessor is given one or more thread blocks to execute, it splits them into warps that get scheduled by the SIMT unit. 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. Every instruction issue time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. 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. Branch divergence occurs only within a warp; different warps execute independently

regardless of whether they are executing common or disjointed code paths. The GPU hardware model is shown in figure 3.4.

CUDA threads may access data from multiple memory spaces during their execution. Each thread has a private local memory, which is akin to local variable declaration for any normal CPU code. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same global memory. CUDA assumes that both the host and the device maintain their own DRAM, referred to as host memory and device memory respectively. The global memory is persistent across kernel launches by the same application and is allocated in the device memory.

Memory management at runtime on the GPU RAM is done using CUDA API equivalents. The general procedure is to allocate memory on both host and device RAM, using cudaMalloc function call for the device memory. The data contents are copied from host memory to device memory using cudaMemcpy function. Writing data directly onto device memory from CPU code is not possible. The kernel calls are then made to do appropriate processing on the data. The processed data contents are copied back from the device to the host using cudaMemcpy function.

4.1 Overview of MPEG Encoding Process

As discussed in the previous chapters MPEG encoding is a computationally intensive process. Figure 4.1 shows the architecture of a MPEG encoder.

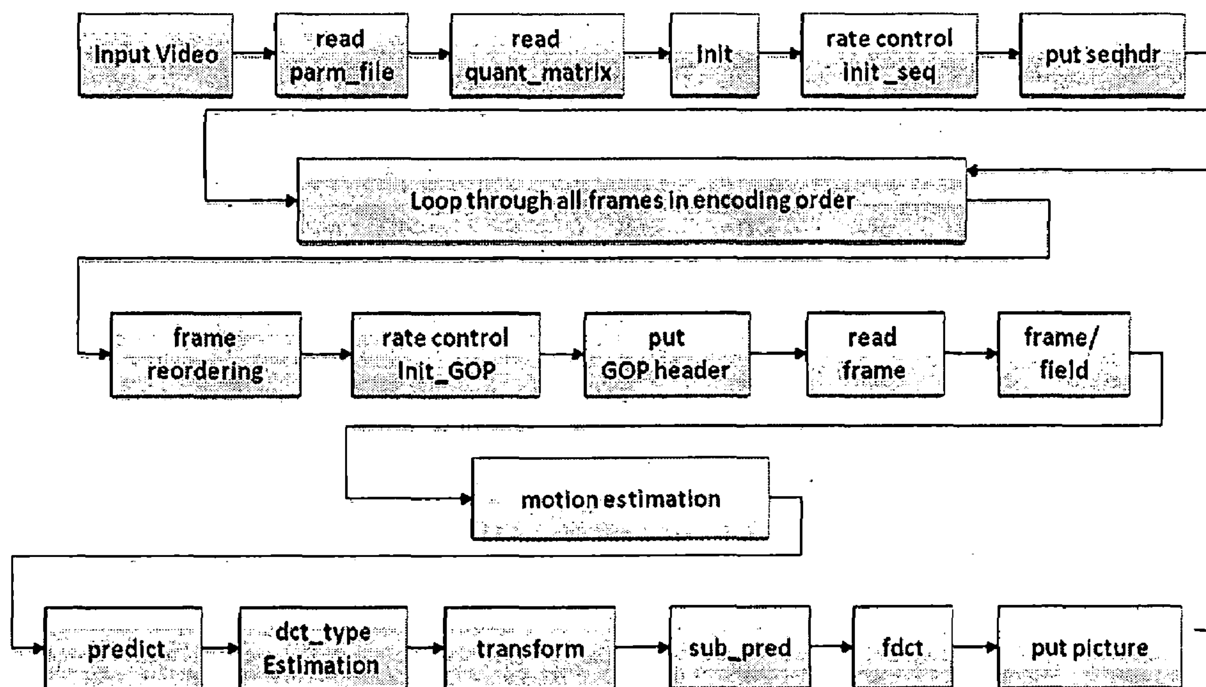


Figure 4.1 MPEG encoder

The process starts with reading parameter file from the hard disk. The parameter file tells the encoder about the resolution of the frame, profiles to be used for encoding, input frame type, number of frames in the GOP etc. After which different parameters such as quantization matrix, frame and GOP headers are initialized. Once the parameters are initialized a loop is executes for all the frames. In this loop, each frame is passed through different stages of encoding, such as motion estimation, prediction, discrete cosine transformation etc. Once a reference is encoded the reference frames are passed through inverse quantization process and the reconstructed frames are used as reference frames, because original frames will not be available at the decoder only reconstructed frames will be available. Signal to noise ratio is calculated which specifies the output quality.

All the routines in this encoding process contribute to the time required for the encoding process. If we compute the time required by the different stages of the process i.e. we conduct profiling of the encoder, we obtain results shown in figure 4.2. The figure shows different modules of the encoding process along with their percentage of total time that each module requires.

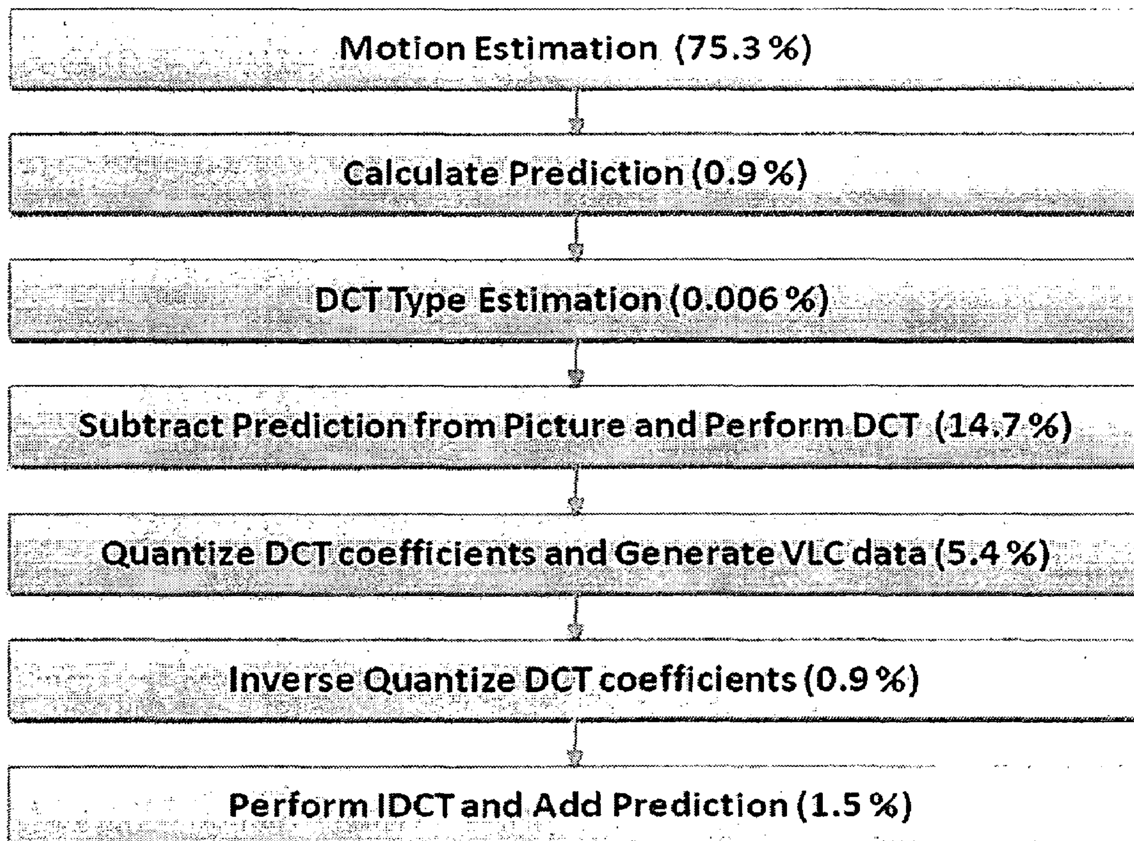


Figure 4.2 Profiling of MPEG encoder

Thus, from the figure it is clear that calculation of motion vectors is the most lengthy or time consuming part of the algorithm. Therefore our focus for optimization is to calculate the motion vectors in parallel and save the overall computation time.

4.2 Parallelization of MPEG encoder

To achieve parallelism in any algorithm data dependencies have to be resolved. For video encoding, data dependencies turn in to frame dependencies. Data parallelism in an MPEG encoder can be exploited at various levels:

1. Data per GOP
2. Data per Frame

3. Data per Macro block

The above levels can be achieved in one of the following ways:

- Encoding GOPs in parallel.
- Encoding different frames of a GOP in parallel.
- Encoding macro blocks in a frame in parallel.

These levels can be compared if we have a look at the profiling of the encoder. The simplest approach for parallelization is the one wherein the frames in a GOP can be encoded in parallel. This approach has been used in most of the previous parallel architecture work to attain considerable speed up. A simple phenomenon is to exploit the number of processing units and assign them totally independent work that can be executed in parallel. As the task size is large enough various distributed architectures can also be used. Even network transmission delay is compensated if the degree of parallelization is high. Instead of using distributed environment such as ATM networks or clusters, parallel processing models such as multiprocessing systems can also be used with this level of parallelization. Figure 4.3 shows the architecture of such a encoder which can be deployed on any parallel processing environment.

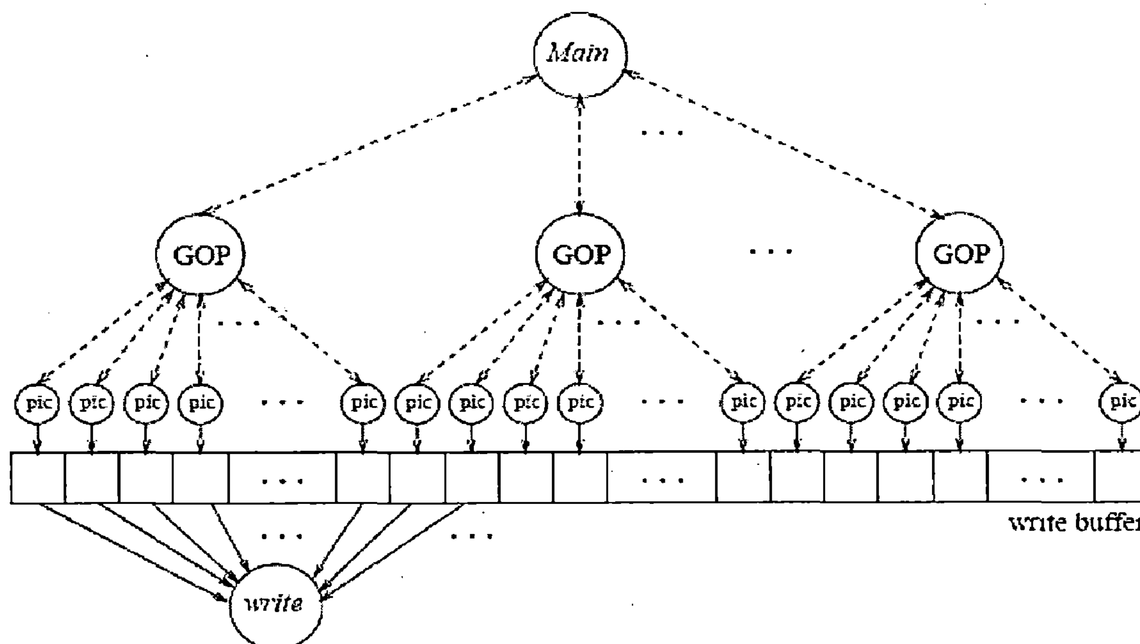


Figure 4.3 Architecture of a parallel MPEG encoder

At second level of parallelism, multiple processing units can be assigned encoding of different frames. Figure 4.4 describes how the encoder can be implemented exploiting

this type of parallelism. First the server or the main program is assigned the responsibility to encode the reference frame and then it distributes the frames to be encoded among the processing units. A major problem with this approach arises if the number of processing units is more. This might sound a bit unusual but the problem arises due to the frame structure in a GOP. We have explained earlier about the dependencies of the frames in a GOP. To enable a processing unit to encode a frame it should be supplied with all reference frames are required, but in a typical GOP structure a P frame is inserted after every 3 – 4 frames. Thus, in order to be able to encode any frame reference frames need to be encoded first. This will create a huge problem as more number of processing will not be able to work together because due to absence of reference frames. However, with limited no of processing units working some speedup is possible but synchronization issues will again depreciate the performance.

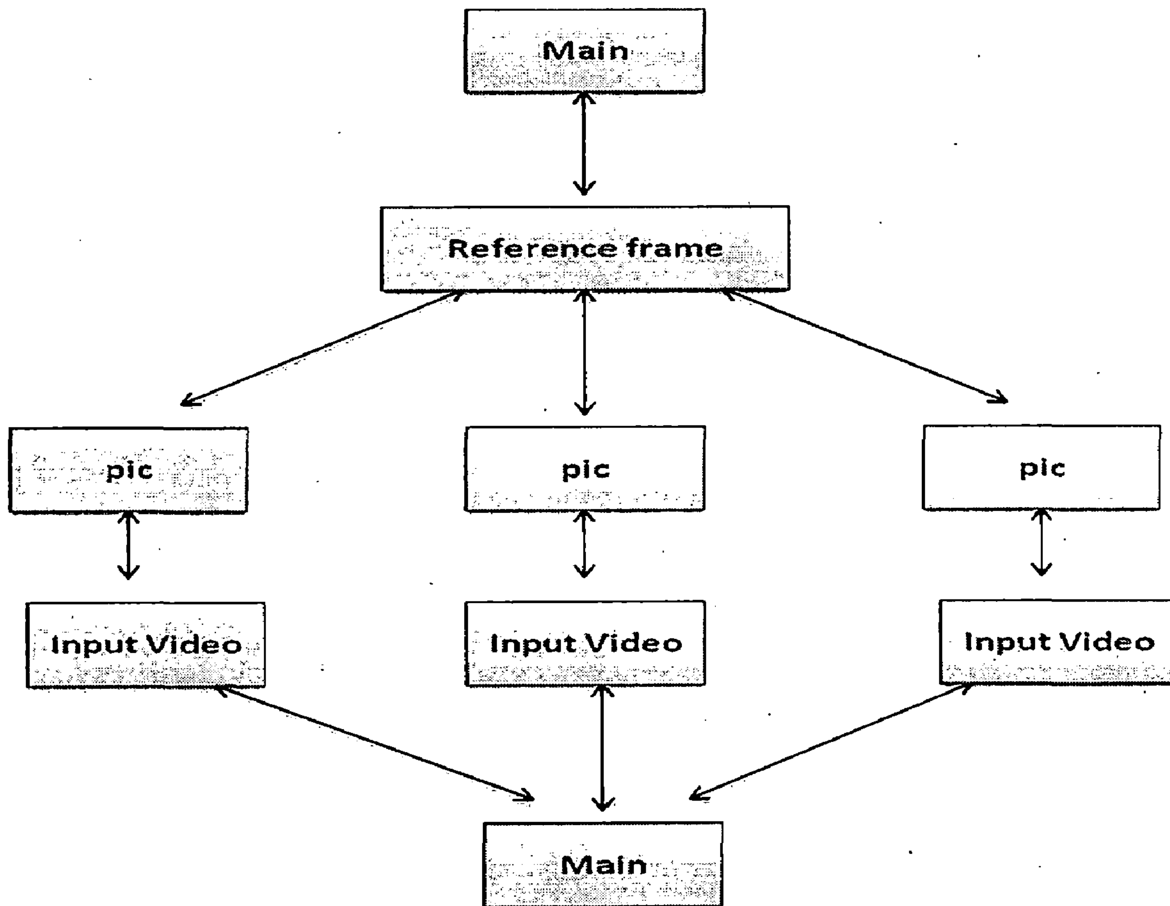


Figure 4.4 Architecture implementing frame level parallelism

If we have a careful look at the profiling of the encoder shown in figure 4.2 we see that maximum portion of the time required for encoding is consumed in finding motion vector, i.e. 75% of the time is consumed in motion estimation routine. Let us have a look at motion estimation process for a frame.

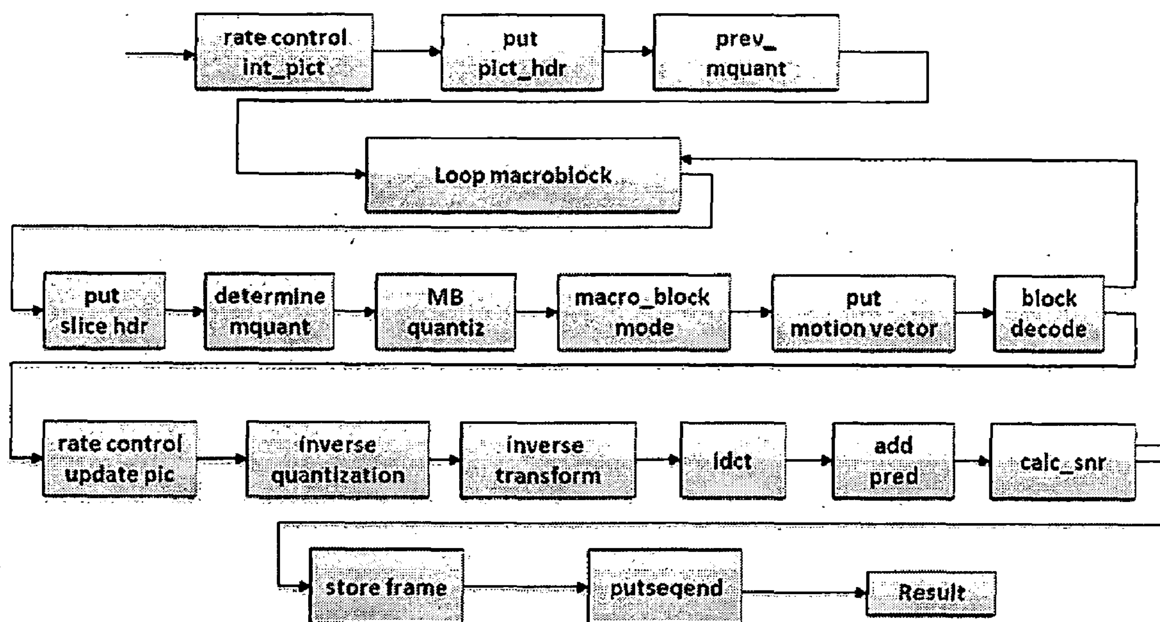


Figure 4.5 Motion estimation in MPEG encoding

From figure 4.5 it is clear that the computationally intensive part of the algorithm is the loop that calculates motion vector for each macro block. Hence, in order to exploit parallelism at macro block level the loop through all macro blocks should be split into different threads and different blocks can be assigned to different processing units. Although, this statement explains that there are enough components to be executed in parallel yet parallelism at this level has not been implemented so far. The prime reason why this level parallelism is not being extensively used is because the parallel processing models such as clusters or grids might not justify the cost of network communication. Although the size of a macro block is small still the time required for detection of motion vectors for a single a macro block takes less than a millisecond which is a small fraction of the transmission time required. Even in multiprocessing environment the cost of non uniform memory access is much expensive.

Thus it is clear that the best way by which maximum performance gain can be achieved by parallelization is by data parallelization at GOP level.

4.3 Porting MPEG Encoder on Cell B. E.

We have described earlier how MPEG encoder can be parallelized. In this section we focus on describing how the Cell B. E. architecture can be utilized in order to attain maximum performance improvement in MPEG encoding process.

4.3.1 Problems in Porting MPEG Encoder on Cell B. E.

Implementation of MPEG encoder on Cell B. E. becomes quite difficult due to certain limitations in the architecture. These limitations are:

- Small SPU local storage size does not permits storing all the reference frames and the encoded frames together
- Maximum possible DMA transfer in one go is 16 KB which disallows a frame to be read in the SPU local store once the size of frame surpasses 128 * 128
- Size of the SPU thread should be minimum because the thread are themselves stored in the SPU local store and the same memory is used by threads to store data
- If more number of branches are present in the SPU threads a penalty of 18 cycles has to be suffered, which minimizes the performance gain

4.3.2 Implementation of the Encoder Capable of Encoding Frames with Resolution 128*128

Implementation of encoder with a capability to encode frames of dimensions 128*128 strictly follows the design discussed in figure 4.3. Program starts with invoking 8 threads which are assigned the responsibility to encode a GOP each. Each thread is assigned a number that signifies which GOP of the sequence has to be encoded by it. The frames are read accordingly i.e. if a thread is assigned a number 4 and there are 12 frames in every GOP then the thread will encode frames 49 – 60.

The threads discussed in the above paragraph are created on PPU. These threads start the encoding process and when motion estimation has to be done they create a SPU thread which starts the motion estimation process and the PPU is made to wait after reading the next frame from the hard disk. The SPU then reads the frames from main memory and store the motion vectors for each macro block in the corresponding

memory once the procedure for motion estimation is complete the motion vectors are written back on to main memory and PPU is notified by mailbox that the motion vectors have been found. The PPU the notifies the SPU that next frame has been read and it can find motion vectors for that frame and then PPU starts DCT calculation, prediction with motion compensation and other routines mentioned above to complete the encoding of the frame. This process is followed until all the fames in the GOP are completely encoded. The above process will be clearer from the figure 4.6.

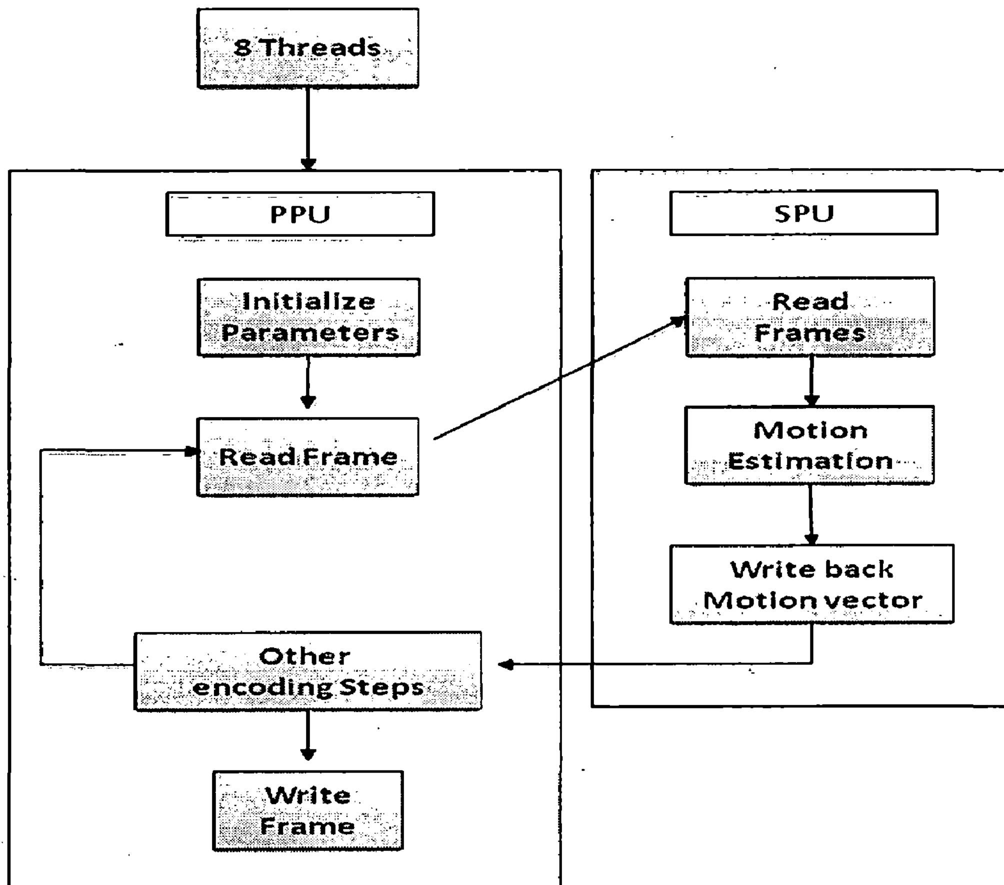


Figure 4.6: Implementation of MPEG encoder on Cell B. E.

Thus each thread utilizes 1 SPU at a time and hence all the SPUs are utilized. Also, other concepts specific to Cell B. E. architecture can be implemented. Double buffering is implemented by a simple method in which every time when motion estimation routine is invoked a request to read the next frame is also made. Thus, before detection of motion vectors of one frame next frame is already in memory. This architecture does not work when size of frames becomes large because SPU local store will not be able to store all the frames as discussed earlier. Higher resolution frames can be encoded as discussed in next section.

4.3.3 Implementation of the Encoder Capable of Encoding Frames with Higher Resolution

Major problem in implementing the encoder is the small size of SPU local store, thus motion estimation should be done in such a way that there is no requirement of keeping the complete frame in the local store. For example let us consider a small example of the routine that calculates variance of a macro block that is required for encoding all the frames in each GOP.

```
int variance(unsigned char *p, int lx)
{
        int i, j;
        unsigned int v, s, s2;

        s = s2 = 0;
        for (j=0; j<16; j++)                        ......... (1)
        {
                for (i=0; i<16; i++)
                {
                        v = *p++;
                        s+= v;
                        s2+= v*v;
                }
                p+= lx-16;
        }
        return s2 - (s*s)/256;
}
```

This is a simple routine that calculates the variance in a macro block here p is the pointer to the first element of the macro block and lx specifies the width of the frame. Now, we consider the scenario as the frame is not in the SPU local store. First of all we align the frames with 128 bit alignment. This will enable us to read any line in a macro block as every line in a macro block contains 16 characters or 128 bits. Now, in the above routine we see that in the first loop (1), characters in every line of the macro

block are used in computation of the variance of the macro block. Thus, if the line of macro block is ready before next iteration of the loop, variance can be calculated successfully. Let us see the sample code for calculation of variance that can be executed on SPU.

```
int variance(unsigned char *p, int lx)
{
         int i, j;
         unsigned int v, s, s2;
        unsigned char *p1, * p2, *p3;
        int tag=1,tag_mask=1<<tag;
        p2 = malloc_align(16, 7);
        p3 = malloc_align(16, 7);
        s = s2 = 0;
        mfc_get(p2, (unsigned char)p, 16, tag, 0, 0);
        p1 = p2;

        for (j=0; j<16; j++)                                    ......... (1)
        {
                mfc_write_tag_mask(tag_mask);
                mfc_read_tag_status_any();
                if(j < 15)
                        mfc_get(p3, (unsigned char)p, 16, tag, 0, 0);
                for (i=0; i<16; i++)
                {
                        v = *p1++;
                        s+= v;
                        s2+= v*v;
                }
                p2 = p3;
                p3 = p1;
                p1 = p2;
                p+= lx-16;
        }
```

```
        return s2 - (s*s)/256;
}
```

This example shows how variance can be calculated from SPU even without the frames being present in the SPU local store, also this code shows how double buffering can be implemented in routine used to find variance in a block. In loop (I), every time when the content of one line is used to calculate the variance a request to read the next line has already been made. Thus, before the calculation starts next line is already in memory and thus, the time required for calculation and the time required to read the macro block line from main memory are interleaved. Hence, double buffering is implemented and an optimized routine is achieved.
Similarly, lines of macro block are read in SPU local store and other routines such as full search block matching and interpolation are also implemented.

4.4 Implementation of MPEG Encoder on CUDA

We have already seen how parallelism can be exploited at various levels while implementing MPEG encoder, but due to various architectural and programming constraints, same approach cannot be implemented on CUDA architecture. Various constraints that were encountered while implementation of the encoder on CUDA are:

- Size of threads that will run on GPUs cannot exceed 1 million instructions and as the motion estimation routine is lengthy enough it easily exceeds the length limit.
- Slower clock in CUDA architecture redefines how processing units should be used. The architecture suggests attaining performance gain by massive parallelization, but parallel encoding of a very large number of GOPs will not be useful as initialization of parameters in so many GOPs will make CPU so slow that CPU will not be able to make use all the processing units or cores present.
- Limited registers and cache space does not allows significant performance gain thus size of threads need to be very small and no of threads needs to be very large.

- No space for branch prediction table for any thread is present in CUDA architecture. Also, threads which opts different branches are blocked and cannot run in parallel.

To overcome these limitations we describe in next section how motion estimation routine can be implemented in order to attain speed up in CUDA architecture.

4.4.1 Implementation of Block Level Parallelism of MPEG Encoder on CUDA

As mentioned above size of the thread has to be kept as small as possible for successful implementation of an algorithm on CUDA. Thus, we choose the smallest unit which can be processed in parallel i.e. macro blocks.

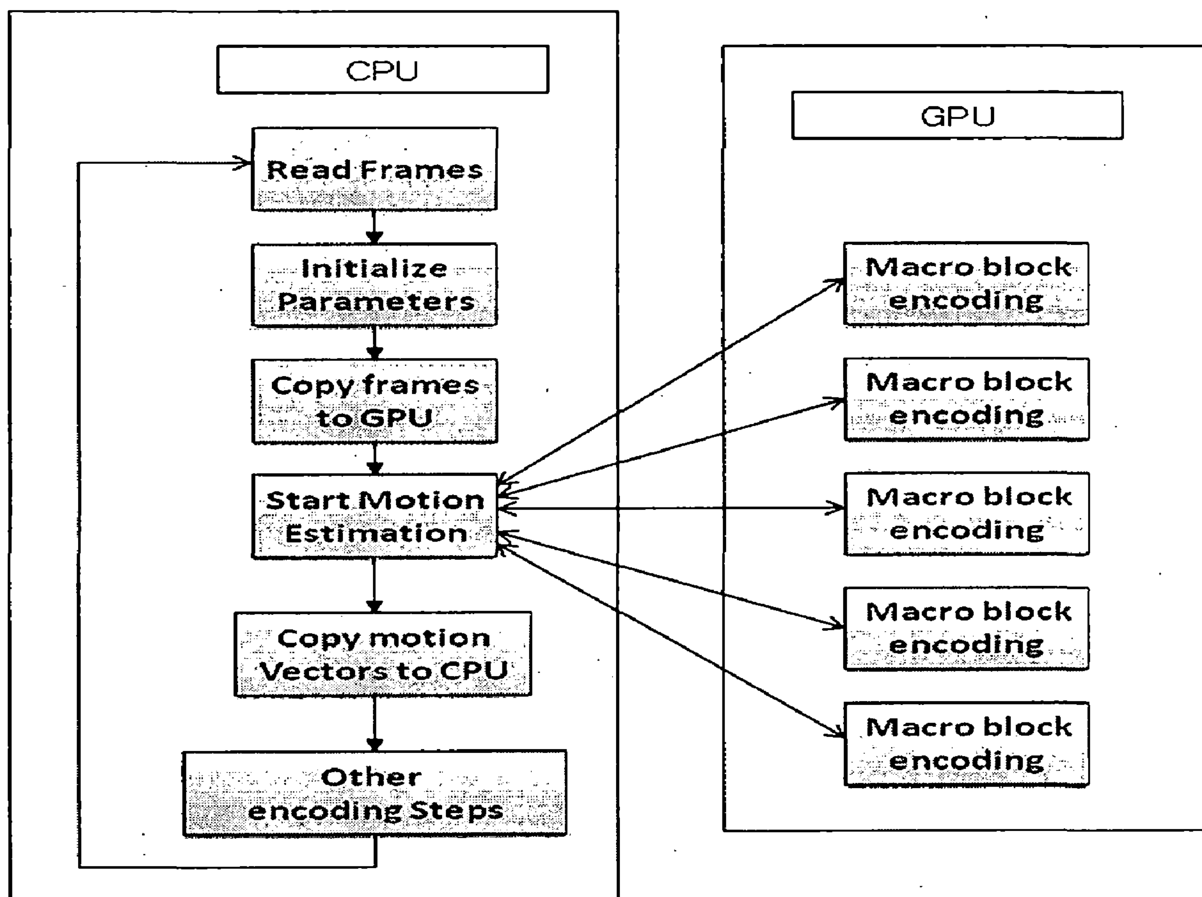


Figure 4.7 Implementation of motion estimation routine using GPUs

In this approach, we implement the third level of parallelism discussed, because the first two levels of parallelism do not suits the CUDA architecture. The motion estimation routine loops through all the macro blocks of a frame and calculate motion

vectors for each block. Instead of doing this process sequentially we create threads in such a way that each thread is assigned the responsibility to find out the motion vector for a macro block assigned. Therefore the number of threads to be created is same as the number of macro block present in the frame. For example frames with a resolution 128 * 128 will contain 128/16 * 128/16 = 64 macro blocks with a size 16 * 16 pixels.

The threads created runs independently on GPU and write the motion vectors on GPU memory. These vectors can be read back on to the CPU memory and the encoding procedure should be continued. Figure 4.7 gives a graphical representation of the approach discussed.

4.5 Implementation of MJPEG encoder on Cell B. E.

4.5.1 MJPEG Encoding Process

A major advantage with MJPEG standard is its flexibility in representation of frames. One of the commonly used representation of frames is similar I frame representation in MPEG encoding [6]. The encoding process of the frames if as follows:

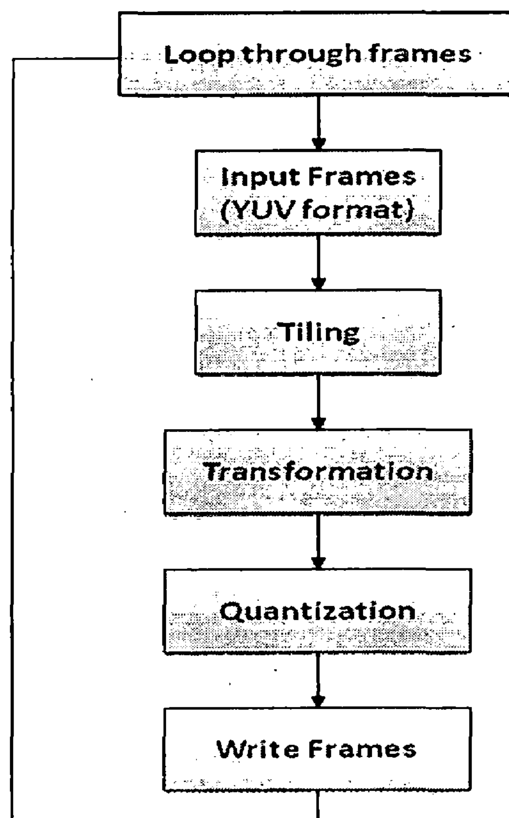


Figure 4.8 MJPEG frame encoding

Figure 4.8 shows how frames are encoded in MJPEG sequence. First of all a frame is divided in to tiles (macro block). Then they are passed through transformation process. Transformation may be either wavelet transformation or discrete cosine transformation. Once transformation is complete then the frames can be encoded after quantizing the obtained transformation coefficients.

4.5.2 Parallel Implementation of MJPEG Encoder

MJPEG encoder can be parallelized by using second level of parallelism as discussed in previous section. The simple way of parallelizing the encoder is to assign 1 frame to each SPU and encode 8 frames together. As each frame has to be encoded independently the implementation will not face any problem discussed above. Figure 4.9 shows the architecture of the encoder.

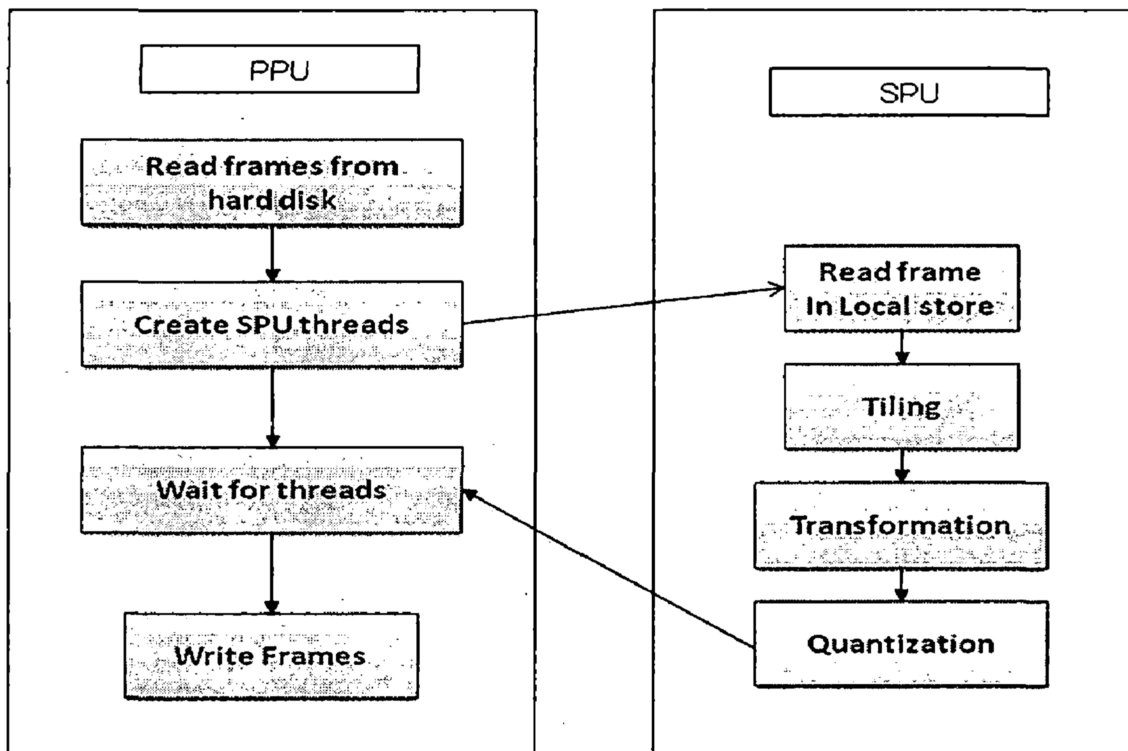


Figure 4.9 MJPEG parallel encoding architecture

The parallelization approach described in chapter 4 were implemented on Cell SDK 2.0 simulator running on windows based platform VMware and NVidia GTX 280 graphics card (having 30 multiprocessors and 8 cores per processor).

5.1 Results on Cell B. E. for Frame Size 128 * 128

First of all the program was executed to encode 8 GOPs having 6 frames per GOP with a resolution of 128 * 128 pixels. The programs were repeatedly executed on different platforms such as Intel P4 HT based Windows machine, Intel Xeon based LINUX machine and Cell SDK 2.0. The average timings obtained are shown in table 5.1.

Platform used	Processing time taken
Intel Pentium 4 HT based windows machine	1809 milliseconds
Intel Xeon based LINUX machine	1312 milliseconds
Simulated Time using Cell SDK 2.0	204 milliseconds
Speedup	8.8 X

Table 5.1: Comparison between Intel processor and Cell processor

Thus, a speed up of 9 times is obtained by implementing the architecture discussed in section 4.2.3.

5.2 Results on Cell B. E. for Frame Size 384 * 288

Now the program was executed to encode 8 GOPs having 6 frames per GOP with a resolution of 384 * 288 pixels. The programs were repeatedly executed on different platforms such as Intel P4 HT based Windows machine, Intel Xeon based LINUX machine and Cell SDK 2.0. The average timings obtained are shown in table 5.2.

Platform used	Processing time taken
Intel Pentium 4 HT based windows machine	12864 milliseconds
Intel Xeon based LINUX machine	9476 milliseconds
Simulated Time using Cell SDK 2.0	1893 milliseconds
Speedup	6.8 X

Table 5.2: Comparison between Intel processor and Cell processor

5.3 Results for Parallel MJPEG Encoding on Cell Processor

Parallelization of MJPEG encoder when implemented on Cell processor for encoding 48 frames with a resolution 384 * 288 yields results shown in table 5.3

Platform used	Processing time taken
Intel Pentium 4 HT based windows machine	1968 milliseconds
Intel Xeon based LINUX machine	1744 milliseconds
Simulated Time using Cell SDK 2.0	322 milliseconds
Speedup	6.1 X

Table 5.3: Comparison between Intel processor and Cell processor

5.4 Results for Parallel Motion Estimation Routine using NVidia GTX 280

The approach discussed in chapter 4 for implementation of motion estimation routine on CUDA architecture was implemented on Intel Xeon based Windows machine with NVidia GTX 280 card. The average time required for executing Motion Estimation routine for B frames are shown in table 5.4.

Platform used	Processing time taken for resolution 128*128	Processing time taken for resolution 384*288
Intel Pentium 4 HT based windows machine	48 milliseconds	380 milliseconds
Intel Xeon based LINUX machine	39 milliseconds	291 milliseconds
Intel Xeon using GTX 280	21 milliseconds	92 milliseconds
Speedup	2.3 X	4.1 X

Table 5.4: Comparison for motion estimation routine between GPU and non GPU using Intel machine

Table 5.5 shows the overall performance improvement after using GPUs for motion estimation on encoding 8 GOPs with 6 frames per GOP.

Platform used	Processing time taken for resolution 128*128	Processing time taken for resolution 384*288
Intel Pentium 4 HT based windows machine	1809 milliseconds	12846 milliseconds
Intel Xeon based LINUX machine	1312 milliseconds	9476 milliseconds
Intel Xeon using GTX 280	1064 milliseconds	5839 milliseconds
Speedup	1.7 X	2.2 X

Table 5.5: Comparison between GPU using and non GPU using Intel machine

5.5 Comparison on MPEG Encoding Time between Cell and CUDA Architecture

The encoders developed were executed to encode 8 GOPs having 6 frames each with resolution 128* 128 and 384 * 288. The results obtained are shown in table 5.6

Platform used	Frames / second for resolution 128*128	Frames / second for resolution 384*288
Intel Pentium 4 HT based windows machine	26.53	3.74
Intel Xeon based LINUX machine	36.59	5.07
Intel Xeon using GTX 280	45.11	8.22
Simulated Time using Cell SDK 2.0	235.49	26.10

Table 5.6: Comparison between GPU and Cell B. E. machine

The performance comparison between all the different approaches discussed and implemented on different architecture is shown by figure 5.1.

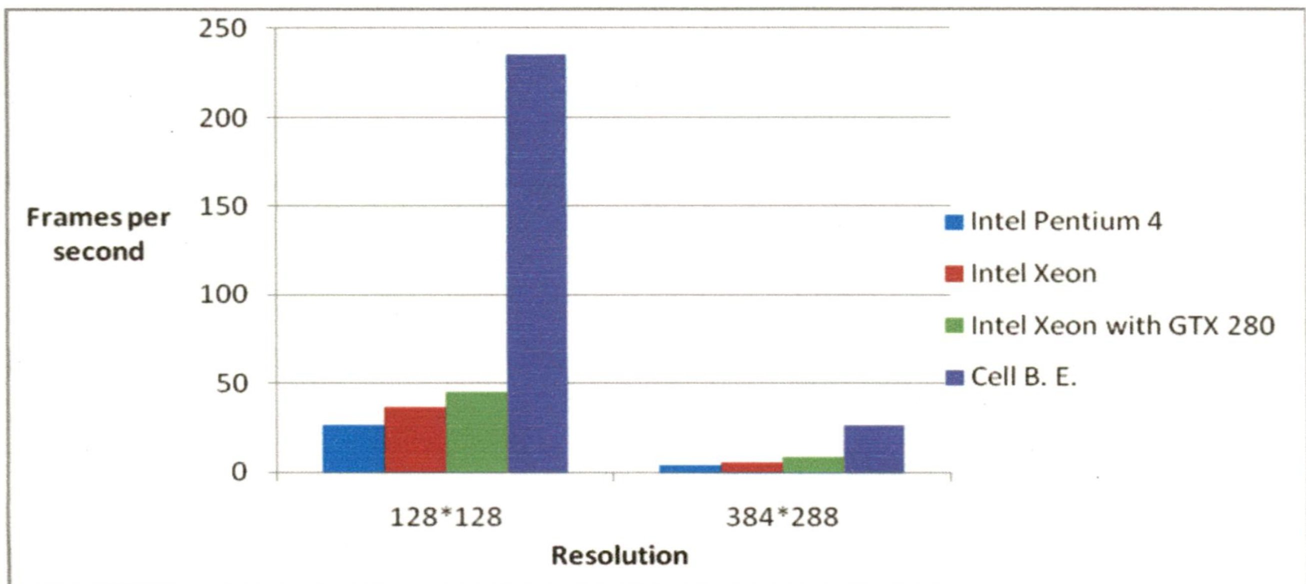


Figure 5.1 Performance comparison of MPEG encoder over various platforms

CHAPTER 6 Conclusions and Future Work

In this thesis we have shown how multicore processors can be used for solving computationally intensive multimedia problems. We have tried to highlight various programming and architectural constraints that do not allow proper utilization of the hardware available. We also focus on how various architectural limitations can be dealt with.

We presented various approaches, using which video encoders can be parallelized. We have also shown how parallelization approaches should be selected in order to utilize the underlying architecture to its potential. The encoders thus presented perform better than the encoders that run over sequential machine.

Advantages of multicore systems lie not only in performance improvement but also in terms of cost effectiveness and resource utilization. Thus, the trend of using multicore systems for solving computationally intensive problems can be viewed as a simple and highly beneficial means for performance improvement. With the use of video encoding on multicore processors various applications that require video encoding at a fast rate can be successfully deployed.

In future we suggest implementation of other video encoding standards such as H.264 and MPEG-4. The parallelization approach suggested in this thesis should be implemented over other parallel processing architecture in order to achieve a more beneficial video encoding system.

References

[1] D. M. Barbosa, J. P. Kitajima, W. Weira, "Parallelizing MPEG video encoding using multiprocessors", Computer Graphics and Image Processing, 1999. Proceedings. XII Brazilian Symposium on , vol., no., pp.215-222, 1999.

[2] S.M. Akramullah, I. Ahmad and M.L. Liou, "Performance of software-based MPEG-2 video encoder on parallel and distributed systems", IEEE Trans. Circuits Syst. Video Tech. 7 4 (1997), pp. 687–695.

[3] K. Shen, L. A. Rowe, E. J. Delp, "A Spatial-Temporal Parallel Approach for Real-time MPEG Video Compression", Proc. of 1996 International Conf. on Parallel Processing, vol. 2, 1996, pp. 100–107.

[4] Tudor, P.N., "MPEG-2 video compression", Electronics & Communication Engineering Journal, vol.7, no.6, pp. 257-264, Dec 1995.

[5] K. Shen, L. A. Rowe, E. J. Delp, "A Spatial-Temporal Parallel Approach for Real-time MPEG Video Compression", Proc. of 1996 International Conf. on Parallel Processing, vol. 2, 1996, pp. 100–107.

[6] T. Fukuhara, K. Katoh, S. Kimura, K. Hosaka, A. Leung, "Motion-JPEG2000 standardization and target market," 2000. Proc. of 2000 International Conference on Image Processing, vol.2, pages.57-60 vol.2, 2000.

[7] J. A. Kahle, " Introduction to the Cell Multiprocessor". In IBM Journal of Research and Development, 49(4): pages 589 - 604, July 2005.

[8] D. A., Shepherd, Z. Sura Z., A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault., Y. Gao, R. Koo. "Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture", IBM systems journal, 2006.

[9] CUDA programming Guide 2.0.

http://developer.download.nvidia.com/compute/cuda/2.1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf last accessed 10/6/09

[10] H. Zhong, S. A. Lieberman, and S. A. Mahlke "Extending multicore architectures to exploit hybrid parallelism in single-thread applications." In Intl.

Symp. on High-Performance Computer Architecture, Phoenix, Arizona, February", 2007.

[11] "Cell Broadband Engine - An Introduction", Cell Programming Workshop, IBM Systems and Technology Group, April 14-18, 2007.

[12] P. F. Gorder. "Multicore Processors for Science and Engineering", Computing in Science and Engineering (IEEE), Volume 9 Issue 2, Page(s): 3-7, March-April ", 2007.

[13] A. Douillet and G. R. Gao. "Software-pipelining on multi-core architectures." In Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'07), Brasov, Romania, September, 2007.

[14] P. Symes, "Video Compression", McGraw-Hill, 1998.

[15] I. Agi and R. Jagannathan, "A portable fault-tolerant parallel software MPEG-1 encoder," Multimedia Tools and Applic, vol. 2, pp. 183–197, 1996.

[16] http://homepages.inf.ed.ac.uk/rbf/CAVIARDATA1/ last accessed 10/6/09

[17] http://www.mpeg.org/pub_ftp/mpeg/mssg/mpeg2v12.zip last accessed 10/6/09

Publications

[1] N. Parakh, A. Mittal, R. Niyogi, "Optimization of MPEG-2 Encoder on Cell B. E. Processor", IEEE International Advance Computing Conference, Patiala, March 2009.