DESIGNING IMAGE PROCESSING ALGORITHMS ON CELL BROADBAND ENGINE: PROGRAMMING STRATEGIES AND PERFORMANCE ANALYSIS

BY

VIJAY P BHATT

07MCE023

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

MAY 2009

DESIGNING IMAGE PROCESSING ALGORITHMS ON CELL BROADBAND ENGINE: PROGRAMMING STRATEGIES AND PERFORMANCE ANALYSIS

Major Project

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science and Engineering

By

VIJAY P BHATT

07MCE023

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

May 2009

Certificate

This is to certify that the Major Project entitled "Designing Image Processing Algorithms on Cell Broadband Engine: Programming Strategies and Performance Analysis" submitted by Mr. Vijay P Bhatt (07MCE023), 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 him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results 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 degree or diploma.

Dr. S. N. Pradhan
Guide and Professor,
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. K. Kotecha
Director,
Institute of Technology,
Nirma University, Ahmedabad

Abstract

The multicore processors, which are widely available in recent times, have great potential to achieve significant performance improvement of the Digital Image Processing Algorithms. Due to the availability of inherent parallelism, large number of resources and functionality on different cores of such multicore architecture, significant performance gain can be obtained by implementing applications on it. But the major challenge lies in programming and designing applications on such multicore architecture to exploit the available resources and functionality to its fullest. These challenges are addressed by developing two image processing algorithms with different computational load on Cell Broadband Engine - a hybrid multicore processor from Sony, IBM and Toshiba. The performance analysis of different versions of two algorithms is shown. The programming approach, designing strategies and drawbacks in development of image processing algorithm on Cell Broadband Engine are addressed.

Acknowledgements

It gives me immense pleasure in expressing my sincere thanks and profound gratitude to Dr. S N Pradhan, Guide and P.G. Coordinator, Department of Computer Science & Engineering, Institute of Technology, Nirma University, Ahmedabad, for his valuable guidance and continual encouragement throughout the Major Project. I am heartily thankful to him for his time to time suggestions and the clarity of the concepts of the topic that helped me a lot during this study. I also thank Prof. Vijay Ukani, Assistant Professor, Department of Computer Science & Engineering, Institute of Technology, Nirma University, Ahmedabad, for his support and valuable guidance in writing this Thesis using Latex.

I like to give my special thanks to Prof. D J Patel, Professor and Head, Department of Computer Science & Engineering, Institute of Technology, Nirma University, Ahmedabad, for his continual kind words of encouragement and motivation throughout the Major Project. I am also thankful to Dr. K Kotecha, Director, Institute of Technology, Ahmedabad, for his kind support in all respect during my study.

I am thankful to all faculty members of Department of Computer Science & Engineering, Nirma University, Ahmedabad, for their special attention and suggestions towards the project work.

The blessings of God and my family members are always there with me to show me the right path. I am very much grateful to them.

The friends, who always bear and motivate me throughout this course, I am thankful to them. Special thanks to Mr. Viral Rathod for his generous help in creation and editing of images in this thesis.

Finally I want to thank the colleagues and experts of Cell BE Forums and bloggers of other Fedora Forums for providing me with solutions when there was any confusion during my entire study.

- Vijay P Bhatt

(07MCE023)

Contents

List of Tables

List of Figures

Abbreviations

ASIC	Application Specific Integrated Circuit
CBE	Cell Broadband Engine
CBEA	Cell Broadband Engine Architecture
DSP	Digital Signal Processing
EIB	Element Interconnect Bus
FPGA	Field Programmable Gate Array
LS	Local Store
MFC	Memory Flow Controller
MMIO	Memory Mapped I/O
PPE	PowerPC Processor Element
SIMD	Single Instruction Multiple Data
SPE	Synergistic Processor Element

Chapter 1

Introduction

1.1 Background

Digital Image Processing is a major bottleneck in applications involving imaging. Hence, improving the performance of Digital Image Processing Algorithms has always been an interesting problem among researchers. In past substantial performance improvement has been achieved by offloading the Digital Image Processing task to specialized hardware consisting of ASICs, DSP Boards, FPGAs, and their various combinations thereof [1],[2]. With the recent advent of latest multicore processors at comparable cost, and with their fast task distribution and inherent parallelism capability on its cores, multicore architectures has presented itself as an obvious solution to improve the performance of Digital Image Processing Algorithms.

Nowadays most of the commercially available PCs and workstations are coming with state-of-the-art multicore processor chips. The multicore architecture can achieve parallelism in two ways: (1) thread level parallelism by running different threads on different cores and (2) fine grain parallelism with the help of architecture specific strategies like Single Instruction Multiple Data (SIMD) parallelization, instruction pipelining, etc. In addition to performance enhancement, development

on multicore processor leads to less development and design cycle time than that on FPGAs or other specialized hardware. Generalized algorithms can be implemented on multicore processors and the multicore processor can do multiple tasks at the same time in a way similar to traditional general purpose processors. Hence use of multicore processors is more viable solution to improve performance of image processing algorithms than traditional general purpose processors with low performance and specialized hardware with high development time and low flexibility.

1.2 Motivation

There are many challenges to be addressed to achieve the best performance on multicore architectures [3]. The major challenge is in designing applications which can efficiently utilize the inherent parallelism and resources of the different cores of the chip. This study is motivated by the fact that, as of now, no commercially available compilers can produce optimal code from traditional scalar application programs, which can fully exploit the functionality of the multicore processors. The software development kits available for programming these multicore processors are in primitive stage and also very few standard libraries are available for developing applications on these multicore processors. As a result, there is a lot of pressure on the application designers to reap the resources and fully exploit the functionality available in different cores built on such multicore processor. Hence the major focus of this study is on development of image processing algorithms on hybrid multicore architecture from Sony, IBM and Toshiba called Cell Broadband Engine (CBE).

1.3 Objective of this Study

The objectives of this study are:

a. The development of standard image processing algorithms on Cell Broadband Engine.

b. To address challenges in designing image processing algorithms for Cell Broadband Engine and programming it.

c. To evaluate the shortcomings and drawbacks in implementing image processing algorithms on Cell Broadband Engine.

d. To present hybrid multicore architecture of Cell Broadband Engine as viable solution to achieve performance enhancement of image processing algorithms.

1.4 General Outline

Wide varieties of applications have been developed on Cell Broadband Engine to achieve performance enhancement.Alias systems have developed Alias Cloth Technology Demonstration for the Cell Processor [4]. Barry Minor et al at of IBM Corporation Systems and Technology Group has developed Terrain Rendering Engine with the help of Cell Broadband Engine [5]. Fabrizio Petrini et al has studied the parallelization process of a scientific application, the radiation transport code Sweep3D, on Cell Broadband Engine [6]. Lurng-Kuo Liu et al has developed Digital Media Indexing on Cell Broadband Engine [7]. Filip Blagojevic et al has studied Dynamic Multigrain Parallelization on the Cell Broadband Engine [8]. As far as image processing is concerned, the work on Fast Fourier Algorithm for Cell Broadband Engine has been done by Alex Chungen Chow et al [9], David Bader et al [10] and Long Chen et al [11]. Also libraries for histogram calculation and convolution have been developed by IBM for Cell Broadband Engine and provided with Cell SDK 3.0 [12]. The parallel processing approaches to Image correlation using SIMD algorithms has been presented by Leah J Siegel et al [13].

In this study, two image processing algorithms - Simple Averaging Filter Algorithm and Mean Normalized Digital Cross Correlation Algorithm are developed and optimized step by step on Cell Broadband Engine processor. The algorithms are

restructured and designed in such a way to fully exploit the SIMD architecture and available resources of the Cell Broadband Engine. The commercially available gaming console - Sony Play Station 3 which comes with 9 cores Cell Broadband Engine Processor is used to port the algorithms on the chip. The operating system running on Cell Broadband Engine is Yellow Dog Linux 5. Cell SDK 3.0 developed by IBM is used for development of algorithms. Cell SDK 3.0 is installed on Fedora Core 7 Operating System and the algorithms are cross compiled with the help of GNU Tool Chain of compilers [14] available with Cell SDK 3.0 for Cell Broadband Engine.

1.5 Scope of the Work

The algorithms developed on Cell Broadband Engine achieved significant performance gain than that on Intel Core2 Duo Processors. The application designing and programming strategies are presented along with the shortcomings and drawbacks of application development on Cell Broadband Engine. The scope of this work covers the development of standard image processing libraries which are optimized and tuned to achieve maximum performance on Cell Broadband Engine. The development of these standard image processing libraries will eventually lead to rapid application development on this hybrid multicore architecture so as to fully exploit the functionality of multiple cores.

1.6 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2, *Overview of Cell Broadband Engine*, describes the Architecture and Programming Environment of Cell Broadband Engine. The concept of SIMD Vectorization is also explained in this chapter.

Chapter 3, *Overview of Cell SDK 3.0*, highlights the important features of Cell SDK

3.0. It also explains the simulation environment of IBM Full-System Simulator.

In **Chapter 4**, *Implementation*, the development of different versions of Simple Averaging Filter and Mean Normalized Digital Cross Correlation Algorithm is explained. The Double Buffering Optimization technique for Mean Normalized Digital Cross Correlation Algorithm is also covered in this chapter.

Chapter 5, *Results and Discussions*, shows the results obtained by implementation of Simple Averaging Filter and Mean Normalized Digital Cross Correlation Algorithm. The programming notes for each algorithm is mentioned. The suggested strategies and drawbacks in implementing Image Processing Algorithms on Cell Broadband Engine are evaluated.

Finally, in **Chapter 6** concluding remarks and scope for future work is presented.

Chapter 2

Overview of Cell Broadband Engine

The first generation Cell Broadband Engine is the first incarnation of a new family of microprocessors conforming to the Cell Broadband Processor Architecture (CBEA). The CBEA is a new architecture that extends the 64-bit PowerPC Architecture. The CBEA and the Cell Broadband Engine are the result of a collaboration between Sony, Toshiba, and IBM, known as STI, formally started in early 2001.

Although the Cell Broadband Engine is initially intended for application in game consoles and media-rich consumer-electronics devices such as high-definition televisions, the architecture and the Cell Broadband Engine implementation have been designed to enable fundamental advances in processor performance. A much broader use of the architecture is envisioned.

The Cell Broadband Engine is a single-chip multiprocessor with nine processors operating on a shared, coherent memory. In this respect, it extends current trends in PC and server processors. The most distinguishing feature of the Cell Broadband Engine is that, although all processors share main storage (the effective-address space

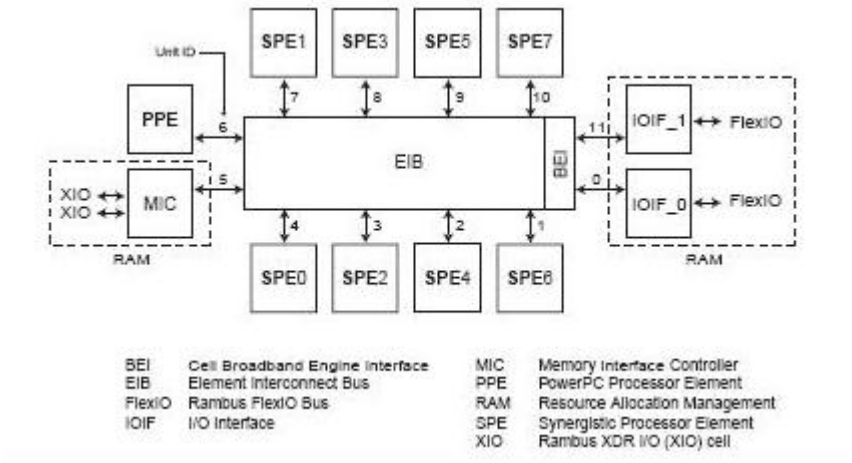

Figure 2.1: Architecture of Cell Broadband Engine

that includes main memory), their function is specialized into two types:

- the PowerPC Processor Element (PPE),
- the Synergistic Processor Element (SPE).

2.1 Cell Broadband Engine Architectural Overview

Figure 2.1 shows the block diagram of Cell Broadband Engine Architecture. The Cell Broadband Engine on Sony PlayStation 3 is a hybrid multicore architecture with 9 cores - 1 Power Processor Element (PPE) and 8 Synergistic Processor Element (SPE) each running at 3.2 GHz frequency. Only 6 out of 8 SPEs are available for user applications on CBE of Sony PlayStation 3. All processor elements are connected to each other and to the on-chip memory and I/O controllers by the memory-coherent element interconnect bus (EIB) [15].

The PPE consists of a 64-bit, dual-thread PowerPC Architecture RISC core and supports a PowerPC virtual-memory subsystem. The PPE contains 32 KB level-1 (L1) instruction and data caches and a 512 KB level-2 (L2) unified (instruction and data) cache [16]. The PPE is designed primarily for control processing, running

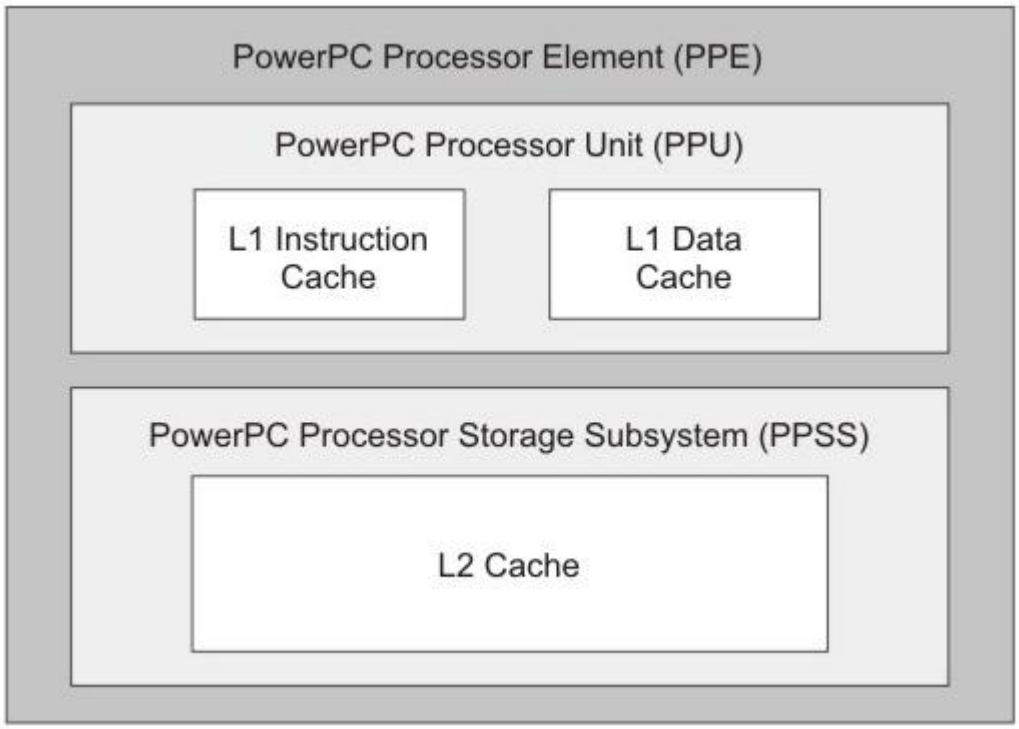

Figure 2.2: PowerPC Processor Element (PPE) block diagram

operating systems, managing system resources, and managing SPE threads. The instruction set for the PPE includes the vector/SIMD multimedia extensions and associated C/C++ intrinsic extensions in addition to PowerPC instruction set.

As shown in Figure 2.2, the PPE consists of two main units:

- The Power Processor Unit (PPU).
- The Power Processor Storage Subsystem (PPSS).

The PPE is responsible for overall control of the system. It runs the operating systems for all applications running on the Cell Broadband Engine.
The PPU deals with instruction control and execution. It includes:

- the full set of 64-bit PowerPC registers,
- 32 128-bit vector registers,

- a 32-KB level 1 (L1) instruction cache,
- a 32-KB level 1 (L1) data cache,
- an instruction-control unit,
- a load and store unit,
- a fixed-point integer unit,
- a floating-point unit,
- a vector unit,
- a branch unit,
- a virtual-memory management unit.

The SPEs of the CBE are single-instruction, multiple-data (SIMD) processor elements that are optimized for data-rich operations allocated to them by the PPE. Each SPE contains a RISC core, 256 KB software-controlled LS (Local Storage) for instructions and data, and a 128-bit, 128-entry unified register file [17]. A special SIMD instruction set-the Synergistic Processor Unit Instruction Set Architecture-and a unique set of commands for managing DMA transfers and interprocessor messaging and control are supported by the SPEs. SPE DMA transfers access main storage using PowerPC effective addresses. The SPEs are not intended to run an operating system. An SPE controls DMA transfers and communicates with the system by means of channels that are implemented in and managed by the SPE's memory flow controller (MFC). The channels are unidirectional message-passing interfaces. The PPE and other devices in the system, including other SPEs, can also access this MFC state through the MFC's memory-mapped I/O (MMIO) registers and queues, which are visible to software in the main-storage address space. Maximum DMA Transfer size is 16 KB.

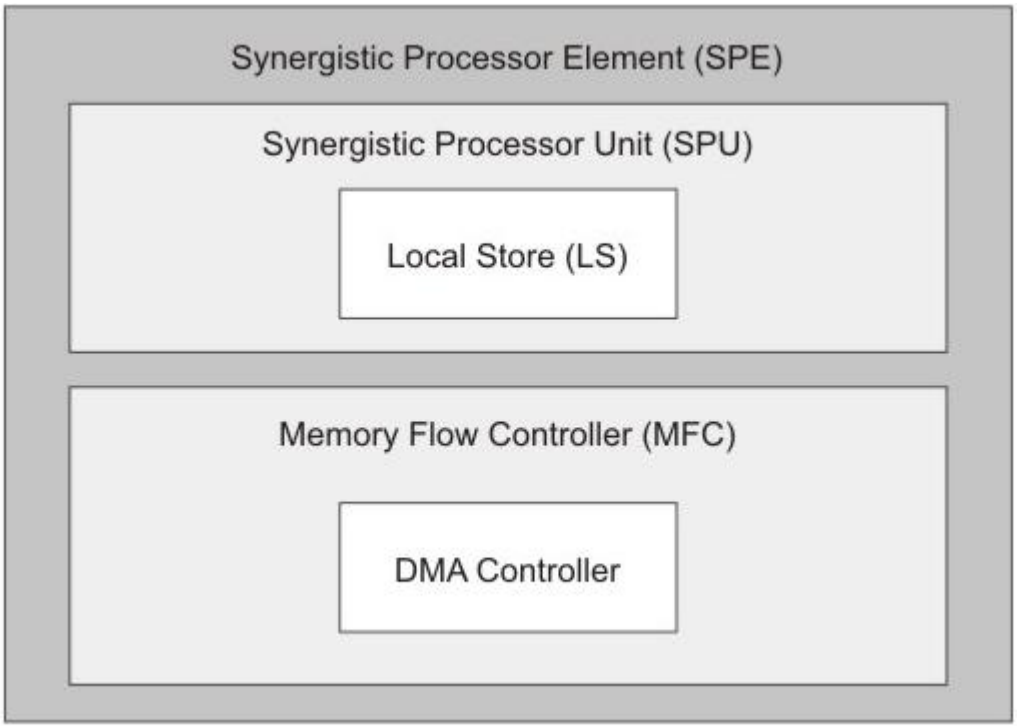

Figure 2.3: Synergistic Processor Element (SPE) block diagram

As shown in Figure 2.3, each SPE consists of two main units:

- The Synergistic Processor Unit (SPU).
- The Memory Flow Controller (MFC).

The SPU deals with instruction control and execution. It includes a single register file with 128 registers (each one 128 bits wide), a unified (instructions and data) 256-KB local store (LS), an instruction-control unit, a load and store unit, two fixed-point units, a floating-point unit, and a channel-and-DMA interface. The SPU implements a new SIMD instruction set, the *SPU Instruction Set Architecture*, that is specific to the Broadband Processor Architecture.

Each SPU is an independent processor with its own program counter and is optimized to run SPE threads spawned by the PPE. The SPU fetches instructions from its own LS, and it loads and stores data from and to its own LS. With respect to accesses

by its SPU, the LS is unprotected and un-translated storage. The MFC contains a DMA controller that supports DMA transfers. Programs running on the SPU, the PPE, or another SPU, use the MFCs DMA transfers to move instructions and data between the SPUs LS and main storage. (Main storage is the effective-address space that includes main memory, other SPEs LS, and memory-mapped registers such as memory-mapped I/O [MMIO] registers.) The MFC interfaces the SPU to the EIB, implements bus bandwidth-reservation features, and synchronizes operations between the SPU and all other processors in the system.

The SPEs provide a deterministic operating environment. They do not have caches, so cache misses are not a factor in their performance. Pipeline-scheduling rules are simple, so it is easy to statically determine the performance of code. Although the LS is shared between DMA read and write operations, load and store operations, and instruction prefetch, DMA operations are accumulated and can only access the LS for at most one of every eight cycles. Instruction prefetch delivers at least 17 instructions sequentially from the branch target. Thus, the impact of DMA operations on loads and stores and program-execution times is, by design, limited.

A significant difference between the PPE and SPEs is how they access memory:

- The PPE accesses main storage (the effective-address space that includes main memory) with load and store instructions that go between a private register file and main storage (which may be cached).

- The SPEs access main storage with direct memory access (DMA) commands that go between main storage and a private local memory used to store both instructions and data. SPE instruction-fetches and load and store instructions access this private local store, rather than shared main storage. This 3-level organization of storage (register file, local store, main storage), with asynchronous DMA transfers between local store and main storage, is a radical break with

conventional architecture and programming models, because it explicitly parallelizes computation and the transfers of data and instructions.

The PPE and SPEs communicate coherently with each other and with main storage and I/O through the EIB. The EIB is a 4-ring structure (two clockwise and two counterclockwise) for data, and a tree structure for commands. The EIBs internal bandwidth is 96 bytes per cycle [18], and it can support more than 100 outstanding DMA memory requests between main storage and the SPEs.

The memory-coherent EIB has two external interfaces, as shown in Figure 2.1:

- *The Memory Interface Controller (MIC)* provides the interface between the EIB and main storage. It supports two Rambus Extreme Data Rate (XDR) I/O (XIO) memory channels and memory accesses on each channel of 1-8, 16, 32, 64, or 128 bytes.

- The *Cell Broadband Engine Interface (BEI)* manages data transfers between the EIB and I/O devices. It provides address translation, command processing, an internal interrupt controller, and bus interfacing. It supports two Rambus FlexIO external I/O channels. One channel supports only non-coherent I/O devices. The other channel can be configured to support either non-coherent transfers or coherent transfers that extend the logical EIB to another compatible external device, such as another Cell Broadband Engine.

2.1.1 Cell Broadband Engine and Power, Memory, and Frequency Limitations

The CBE processor overcomes three important limitations of contemporary microprocessor performancepower use, memory use, and clock frequency.

Microprocessor performance is approaching limits of power dissipation rather than integratedcircuit resources (transistors and wires). The only way to significantly increase the performance of microprocessors in this environment is to improve power efficiency at approximately the same rate as the performance increase. The CBE processor does this by differentiating between the PPE, optimized to run an operating system and control-intensive code, and the eight SPEs, optimized to run compute-intensive applications. The control-plane PPE leaves the eight data-plane SPEs free to compute data-rich applications.

On todays symmetric multiprocessorseven those with integrated memory controllerslatency to DRAM memory is approaching 1,000 cycles. As a result, program performance is dominated by moving data between main storage and the processor. Compilers and application writers must manage this data movement explicitly, even though the hardware cache mechanisms are supposed to relieve them of this task. In contrast, the CBE processors mechanisms for dealing with memory latenciesthe 3-level SPE memory structure (main storage, local stores, and large register files), and asynchronous DMA transfersenable programmers to schedule simultaneous data and code transfers to cover long memory latencies. At 16 simultaneous transfers per SPE, the CBE processor can support up to 128 simultaneous transfers between the SPE local stores and main storage. This surpasses the bandwidth of conventional processors by a factor of almost twenty.

Conventional processors require increasingly deeper instruction pipelines to achieve higher operating frequencies. This technique has reached a point of diminishing returnsand even negative returns if power is taken into account. By specializing the PPE for control-intensive tasks and the SPEs for compute-intensive tasks, these processing elements run at high frequencies without excessive overhead. The PPE achieves efficiency by executing two threads simultaneously rather than by optimizing single-thread performance. Each SPE achieves efficiency by using a large register file that

supports many simultaneous in-flight instructions without the overhead of register-renaming or out-of-order processing, and asynchronous DMA transfers, that support many concurrent memory operations without the overhead of speculation.

By distinguishing and separately optimizing control-plane and data-plane processor elements, the CBE processor mitigates the problems posed by power, memory, and frequency limitations. The net result is a multiprocessor that, at the power budget of a conventional PC processor, can provide approximately ten-fold the peak performance of a conventional processor. Of course, actual application performance varies. Some applications may benefit little from the SPEs, whereas others show a performance increase well in excess of ten-fold. In general, computeintensive applications that use 32-bit or smaller data formats (such as single-precision floatingpoint and integer) are excellent candidates for the CBE processor.

2.2 Cell Broadband Engine Programming Overview

The Cell Broadband Engine looks like a 9-way coherent multiprocessor to an application programmer. The PPE is more adept in control-intensive tasks and quicker at task switching while the SPEs are more adept in compute-intensive tasks and slower at task switching. However, either processor element is capable of doing both types of functions. This specialization is a significant factor accounting for the order-of-magnitude improvement in peak computational performance and chip-area-and-power efficiency that the CBE processor achieves over conventional PC processors [18].

CBE-processor software development is done in the C/C++ language and supported by a rich set of language extensions that define C/C++ data types for SIMD operations and map C/C++ intrinsics (which are commands, in the form of function calls) to one or more assembly instructions. These language extensions give C/C++ programmers great control over code performance, without the need for assembly

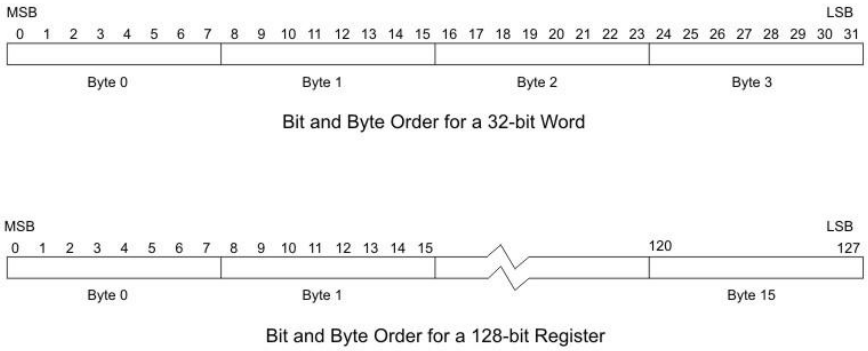

Figure 2.4: Big-endian byte and bit ordering

language programming.

The instruction set for the PPE is an extended version of the PowerPC Architecture instruction set. The extensions consist of the vector/SIMD multimedia extensions, a few additions and changes to PowerPC Architecture instructions, and C/C++ intrinsics for the vector/SIMD multimedia extensions.

The instruction set for the SPEs is a new SIMD instruction set, the Synergistic Processor Unit Instruction Set Architecture, with accompanying C/C++ intrinsics, and a unique set of commands for managing DMA transfer, external events, interprocessor messaging, and other functions. The instruction set for the SPEs is similar to that of the PPE's vector/SIMD multimedia extensions, in the sense that they operate on SIMD vectors. However, the two vector instruction sets are distinct, and programs for the PPE and SPEs are often compiled by different compilers.

Storage of data and instructions in the CBE processor uses big-endian ordering, which has the following characteristics:

- Most-significant byte stored at the lowest address, and least-significant byte stored at the highest address.

- Bit numbering within a byte goes from most-significant bit (bit 0) to least-significant bit (bit n). This differs from some other big-endian processors [15].

Figure 2.4 shows a summary of the byte-ordering and bit-ordering in memory, as well as the bit-numbering conventions.

While developing application on CBE, the usual practice is to run a main program on the PPE that allocates threads to the SPEs. In such an application, the main thread is said to spawn one or more CBE tasks. A CBE task has one or more main threads associated with it, along with some number of SPE threads. An SPE thread is a thread that is spawned to run on an available SPE. An SPE thread has its own 128 x 128-bit register file, program counter and MFC-DMA command queues [18].

A main thread can interact directly with an SPE thread through the SPE's LS. It can interact indirectly through the main-storage space. A thread can poll or sleep, waiting for SPE threads. The operating system defines the mechanism and policy for selecting an available SPE. It must prioritize among all the CBE applications in the system, and it must schedule SPE execution independently from regular main threads. The operating system is also responsible for runtime loading, passing parameters to SPE programs, notification of SPE events and errors, and debugger support.

2.2.1 PPE Registers

The PPE problem-state (user) registers are shown in Figure 2.5. All computational instructions operate on registers; no computational instructions modify main storage. To use a storage operand in a computation and then modify the same or another storage location, the contents of the storage operand must be loaded into a register, modified, and then stored back to the target location.

The PPE registers include:

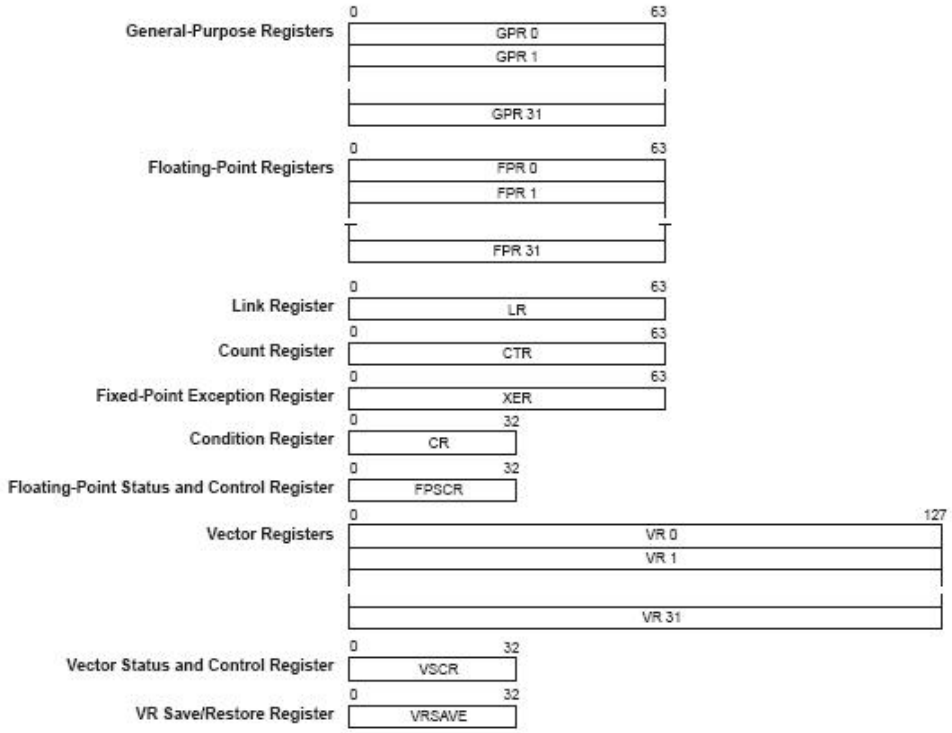

Figure 2.5: PPE User Register Set

- *General-Purpose Registers (GPRs)* Fixed-point instructions operate on the full 64-bit width of the GPRs, of which there are 32. The instructions are mode-independent, except that in 32-bit mode, the processor uses only the low-order 32 bits for determination of a memory address and the carry, overflow, and record status bits.

- *Floating-Point Registers (FPRs)* The 32 FPRs are 64 bits wide. The internal format of floating-point data is the IEEE 754 double-precision format. Single-precision results are maintained internally in the double-precision format.

- *Link Register (LR)* The 64-bit LR can be used to hold the effective address of a branch target. Branch instructions with the link bit (LK) set to 1 (that is, subroutine-call instructions) copy the next instruction address into the LR. A Move To Special-Purpose Register instruction can copy the contents of a GPR into the LR.

- *Count Register (CTR)* The 64-bit CTR can be used to hold either a loop counter or the effective address of a branch target. Some conditional-branch instruction forms decrement the CTR and test it for a zero value. A Move To Special-Purpose Register instruction can copy the contents of a GPR into the CTR.

- *Fixed-Point Exception Register (XER)* The 64-bit XER contains the carry and overflow bits and the byte count for the move-assist instructions. Most arithmetic operations have instruction forms for setting the carry and overflow bit.

- *Condition Register (CR)* Conditional comparisons are performed by first setting a condition code in the 32-bit CR with a compare instruction or with a recording instruction. The condition code is then available as a value or can be tested by a branch instruction to control program flow. The CR consists of eight independent 4-bit fields grouped together for convenient save or restore during a context switch. Each field can hold status information from a comparison, arithmetic, or logical operation. The compiler can schedule CR fields to avoid data hazards in the same way that it schedules the use of GPRs. Writes to the CR occur only for instructions that explicitly request them; most operations have recording and non-recording instruction forms.

- *Floating-Point Status and Control Register (FPSCR)* The processor updates the 32-bit FPSCR after every floating-point operation to record information about the result and any associated exceptions. The status information required by IEEE 754 is included, plus some additional information for exception handling.

- *Vector Registers (VRs)* There are 32 128-bit-wide VRs. They serve as source and destination registers for all vector instructions.

- *Vector Status and Control Register (VSCR)* The 32-bit VSCR is read and writ-

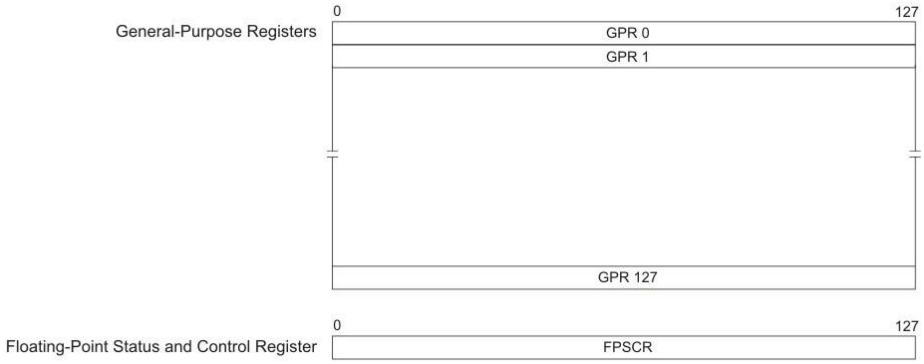

Figure 2.6: SPE user-register set

ten in a manner similar to the FPSCR. It has 2 defined bits, a non-Java mode bit and a saturation bit; the remaining bits are reserved. Special instructions are provided to move the VSCR to a VR register.

- *Vector Save Register (VRSAVE)* The 32-bit VRSAVE register assists user and privileged software in saving and restoring the architectural state across context switches.

2.2.2 SPE Registers

The complete set of SPE user registers is shown in Figure 2.6. All computational instructions operate only on registersthere are no computational instructions that modify storage. The SPE registers include:

- *General-Purpose Registers (GPRs)* All data types can be stored in the 128-bit GPRs, of which there are 128.

- *Floating-Point Status and Control Register (FPSCR)* The processor updates the 128-bit FPSCR after every floating-point operation to record information about the result and any associated exceptions.

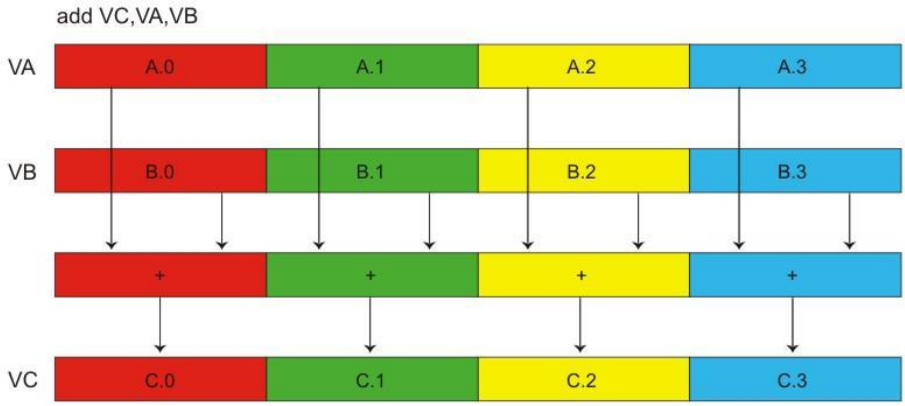

Figure 2.7: Vector Addition Operation

2.3 SIMD Vectorization

A vector is an instruction operand containing a set of data elements packed into a one-dimensional array. The elements can be integer or floating-point values. Most Vector/SIMD Multimedia Extension and Synergistic Processing Unit (SPU) instructions operate on vector operands [18]. Vectors are also called SIMD operands or packed operands.

SIMD processing exploits data-level parallelism. Data-level parallelism means that the operations required to transform a set of vector elements can be performed on all elements of the vector at the same time. That is, a single instruction can be applied to multiple data elements in parallel.

Support for SIMD operations is pervasive in the Cell Broadband Engine. In the PPE, they are supported by the Vector/SIMD Multimedia Extension instruction set. In the SPEs, they are supported by the SPU instruction set. In both the PPE and SPEs, vector registers hold multiple data elements as a single vector. The data paths and registers supporting SIMD operations are 128 bits wide, corresponding to four full 32-bit words. This means that four 32-bit words can be loaded into a single reg-

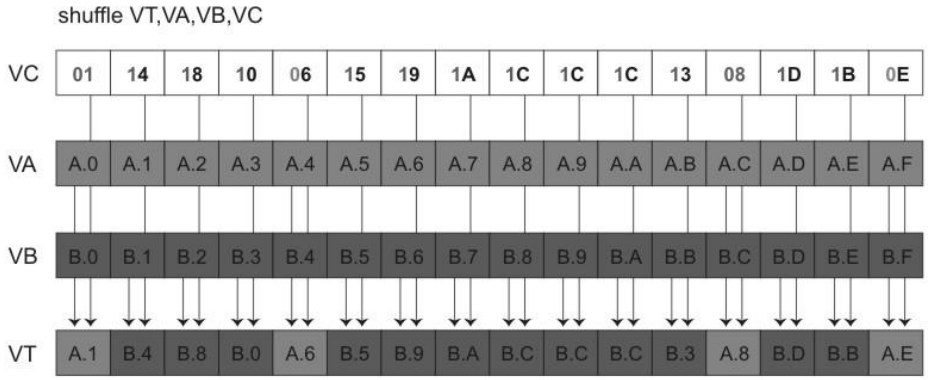

Figure 2.8: Byte-Shuffle (Permute) Operation

ister, and, for example, added to four other words in a different register in a single operation. Figure 2.7 shows such an operation. Similar operations can be performed on vector operands containing 16 bytes, 8 halfwords, or 2 doublewords.

The process of preparing a program for use on a vector processor is called *vectorization* or *SIMDization*. It can be done manually by the programmer, or it can be done by a compiler that does *auto-vectorization*.

Figure 2.8 shows another example of an SIMD operation in this case, a byte-shuffle operation. Here, the bytes selected for the shuffle from the source registers, VA and VB, are based on byte entries in the control vector, VC, in which a 0 specifies VA and a 1 specifies VB. The result of the shuffle is placed in register VT.

Chapter 3

Overview of Cell SDK 3.0

The IBM Software Development Kit for Multicore Acceleration Version 3.0 (SDK 3.0) is a complete package of tools to enable you to program applications for the Cell Broadband Engine (Cell BE) processor. The SDK 3.0 is composed of development tool chains, software libraries and sample source files, a system simulator, and a Linux kernel, all of which fully support the capabilities of the Cell BE.

Cell BE applications can be developed on the following platforms:

- x86
- X86-64
- 64-bit PowerPC (PPC64)
- BladeCenter QS20
- BladeCenter QS21

The supported languages for developing Cell BE applications are as follows:

- C/C++
- Assembler

- Fortran
- ADA (Power Processing Element (PPE) Only)

Although C++ and Fortran are supported, take care when you write code for the Synergistic Processing Units (SPUs) because many of the C++ and Fortran libraries are too large for the 256 KB local storage memory available [14].

The SDK consists of numerous components including the following:

- The IBM Full System Simulator for the Cell Broadband Engine, systemsim
- system root image containing Linux execution environment for use within systemsim.
- GNU tools including C and C++ compilers, linkers, assemblers and binary utilities for both PPU and SPU.
- IBM xlc (C and C++) compiler for both PPU and SPU.
- IBM xlf (Fortran) compiler for both PPU and SPU.
- newlib for the SPU. newlib is a C standard library designed for use on embedded systems.
- gdb debuggers for both PPU and SPU with support for remote gdbserver debugging. The PPU debugger also provides combined, PPU and SPU, debugging.
- PPC64 Linux with CBE enhancements.
- SPE Runtime Management Library providing a standardized, low-level application programming interface for application access to the SPEs.
- Libraries to assist in the development and execution of parallel applications, including the:

  - Accelerated Library Framework library (ALF) support SM, and the
  - Data Communication and Synchronization (DaCS) library.

- Performance tools including:
  - oprofile is a system-wide profiler for Linux,
  - CellPerfCount is a low level tool to configure and access HW performance counters,
  - FDPR-Pro is a tool for gather information for feedback directed optimization,
  - CodeAnalyzer examines executable files and displays detailed information about functions, basic blocks, and assembly instructions, and
  - Visual Performance Analyzer (VPA) is an Eclipse-based performance visualization toolkit.
  - spu_timing is a static timing analysis timing tool that instruments assembly source (either compiler or programmer generated) with expected, linear, instruction timing details.
  - PDT is a performance debugging tool which provides a tracing infrastructure for application timing analysis.

- An Eclipse-based Integrated Development Environment (IDE) to improve programmer productivity and integration of development tools.

- Standardized SIMD math libraries for the PPUs Vector/SIMD Multimedia Extension and the SPU.

- Mathematical Acceleration Subsystem (MASS) libraries supporting both long and short (SIMD) vectors.

- Cell optimized domain-specific application libraries, including Basic Linear Algebra Subprograms (BLAS) library, Fast Fourier Transform (FFT) library, and Monte Carlo Random Number Generator library.

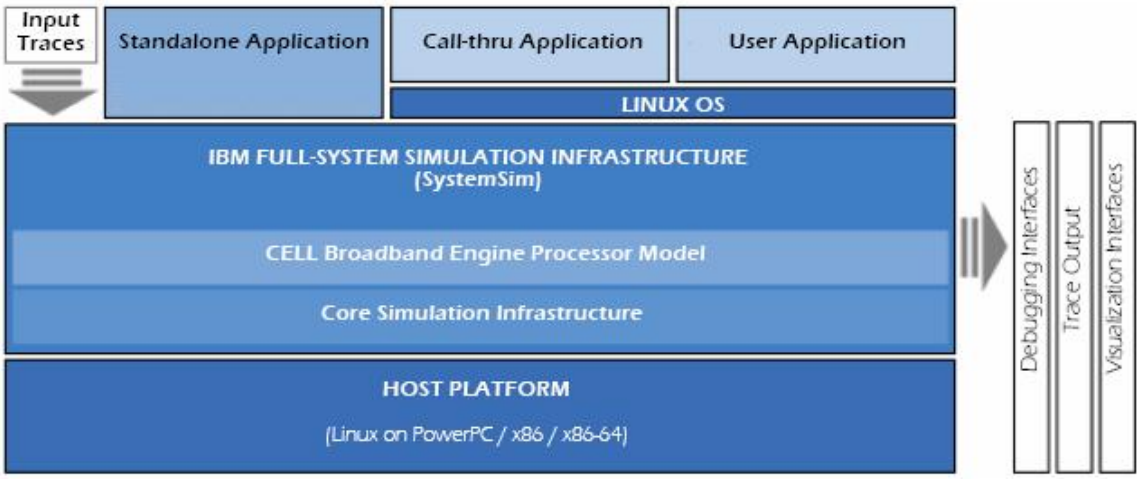

Figure 3.1: Simulator Stack for the Cell Broadband Engine

3.1 IBM Full-System Simulator

The IBM Full-System Simulator for the Cell Broadband Engine is a generalized simulator that can be configured to simulate a broad range of full-system configurations. It supports functional simulation of complete systems based on the Cell Broadband Engine processor, including simulation of the PPE, SPUs, MFCs, memory, disk, network, and system console. The SDK, however, provides a ready-made configuration of the simulator for Cell Broadband Engine system development and analysis. The simulator also includes support for performance simulation (or timing simulation) of certain components to allow users to analyze performance of Cell Broadband Engine applications. It can simulate and capture many levels of operational details on instruction execution, cache and memory subsystems, interrupt subsystems, communications, and other important system functions. Figure 3.1 shows the simulation stack.

The IBM Full-System Simulator provides different simulation modes, ranging from functional simulation of processor instructions to performance simulation of an entire system. In most cases, the simulation mode can be changed dynamically at any point in the simulation. However, certain warm-up effects may affect the results of performance simulation for some portion of the simulation following a change to cycle mode.

Simple (functional-only) mode models the effects of instructions, without attempting to accurately model the time required to execute the instructions. In simple mode, a fixed latency is assigned to each instruction; the latency can be arbitrarily altered by the user. Since latency is fixed, it does not account for processor implementation and resource conflict effects that cause instruction latencies to vary. Functional-only mode assumes that memory accesses are synchronous and instantaneous. This mode is useful for software development and debugging, when a precise measure of execution time is not required.

Fast mode is similar to functional-only mode in that it fully models the effects of instructions while making no attempt to accurately model execution time. In addition, fast mode bypasses many of the standard analysis features provided in functional-only mode, such as statistics collection, triggers, and emitter record generation. Fast mode simulation is intended to be used to quickly advance the simulation through uninteresting portions of program execution to a point where detailed analysis is to be performed.

Cycle (performance) mode models not only functional accuracy but also timing. It considers internal execution and timing policies as well as the mechanisms of system components, such as arbiters, queues, and pipelines. Operations may take several cycles to complete, accounting for both processing time and resource constraints.

The IBM Full System Simulator for the Cell Broadband Engine is started with a graphics user interface by following command:
systemsim -g

Two new windows will appear on the screen. The first is a command-line/console window labeled mysim in the windows title bar. The second is the simulator graphical

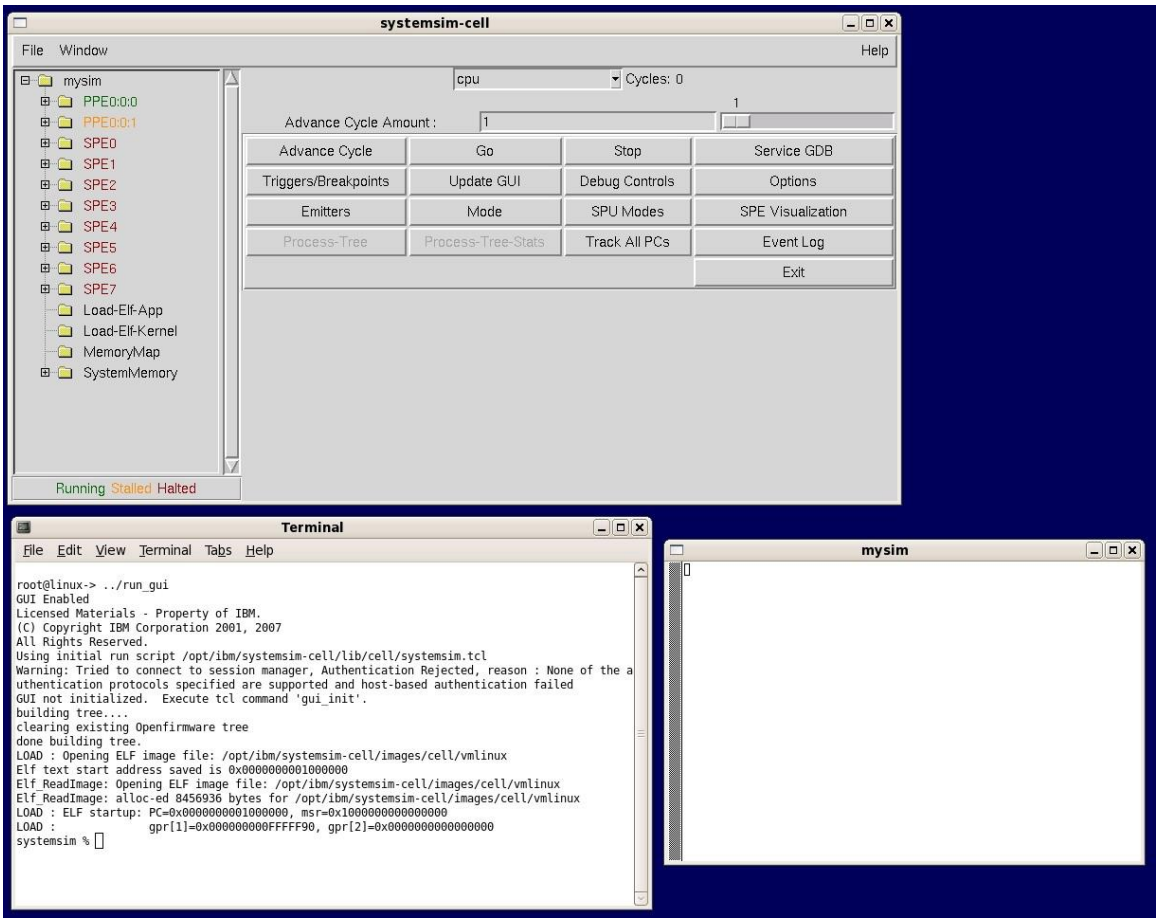

Figure 3.2: Windows visible after starting the simulator GUI

user interface (GUI) window. These windows are shown in Figure 3.2. The window labeled mysim is an uart window that, when Linux boots, it becomes a Linux console window. When the console window first appears, it is empty and there is no user prompt, because Linux has not yet been booted on the simulated system.The window in which the simulator was started (systemsim -g) is the simulator command-line window.

3.1.1 The Callthru Utility

The callthru utility allows you to copy files between the host system and the simulated system while it is running. This utility runs within the simulated system and accesses

files in the host system using special callthru functions of the simulator. The source code for this utility is provided with the simulator in the sample/callthru directory as a sample of the use of the simulator callthru functions. In the Cell SDK, the callthru utility is installed as a binary application in the simulator system root image in the /usr/bin directory. The callthru utility supports the following options:

- To write standard input into ¡filename¿ on the host system, issue
  callthru sink <filename>
- To write the contents of <filename> on the host system to standard output, issue
  callthru source <filename>

Redirecting appropriately lets you copy files between the host and simulated system. For example, to copy the /tmp/matrix_mul application from the host into the simulated system and then run it, issue the following commands in the console window of the simulated system:
callthru source /tmp/matrix_mul > matrix_mul
chmod +x matrix_mul
./matrix_mul

Another commonly used feature of the callthru utility is the exit option, which will stop the simulation, similar to the stop button of the GUI, but initiated by the callthru utility inside the simulator rather than through user interaction. This is especially useful for constructing scripted executions of the simulator that involve alternating steps in the simulator and the simulated system.

3.2 Compiling and linking with the GNU tool chain

The GNU tool chain available with Cell SDK 3.0 includes a GCC compiler and utilities that optimize code for the Cell BE processor. These are:

- The spu-gcc compiler for creating an SPU binary
- The ppu32-embedspu tool
- The ppu-gcc compiler
- The ppu-embedspu tool which enables an SPU binary to be linked with a PPU binary into a single executable program
- The ppu32-gcc compiler for compiling the PPU binary and linking it with the SPU binary

Chapter 4

Implementation

To achieve the performance gain on CBE, the application should partition the work on the available cores of the processor. Various considerations like processing-load distribution, program structure, program data flow and data access patterns, cost, in time and complexity of code movement and data movement among processors, and cost of loading the bus and bus attachments must be taken into account while distributing the workload and data.

The model chosen in this study for partitioning an application on CBE is PPE-Centric. In the PPE-centric model, the main application runs on the PPE, and individual tasks are off-loaded to the SPEs. The PPE then waits for, and coordinates, the results returning from the SPEs. The SPEs can be used in three ways in PPE-Centric model - the multistage pipeline model, the parallel stages model and the services model [18]. Out of these three models, the parallel stages model as shown in Figure 4.1, has been chosen to implement the simple averaging filter and mean normalized digital cross correlation algorithm. The selected image processing algorithms have large amount of data that can be partitioned and acted on at the same time. Hence, it typically make sense to use SPEs to process different portions of that data in parallel and thus the PPE-centric Parallel Stages Model is used for implementation.

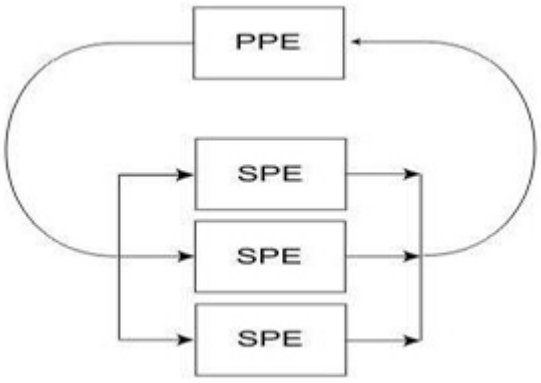

Figure 4.1: Parallel Stages Model

The programming approach in implementing the algorithms is to develop different versions of each algorithm step-by-step, gradually exploiting the functionally of the CBE to its maximum. In first step, the scalar version of the algorithm is developed on Intel Core2 Duo Processor running at 2.20 GHz frequency. In second step, the scalar version of algorithm is developed to run on only the PPE core of the CBE. In third step, the scalar version of algorithm is parallelized to run on the PPE and the multiple SPEs of the CBE. In fourth step, the SIMD Vectorization of the algorithm is done on only the PPE core of the CBE. Finally, in the fifth step, the SIMD Vectorized algorithm is parallelized to run on the PPE and multiple SPEs of the CBE. At each step, the functional correctness of the algorithm is asserted by comparing the results obtained with the simulation results of the algorithm developed in MATLAB using MATLAB's standard functions. And finally the performance results are compared by measuring the time using C language's gettimeofday() and clock() functions. But the clock() function does not count the clocks of the portion of the program running on SPEs. Hence the results shown in this study are that obtained by using gettimeofday() function.

The Simple Averaging Filter algorithm and mean normalized digital cross correlation algorithm are chosen due to their simplicity in application partitioning. The image data can be easily partitioned in such a way that the algorithm can work on

different portions of the image data in parallel. This helped in putting the entire focus on the application designing strategies for the Cell Broadband Engine and not worrying about the complexities in partitioning the data for parallelization of the algorithm.

For both the scalar and SIMD version of the algorithms on PPE and multiple SPEs, the main thread running on the PPE reads the input image from the disk. Then the PPE thread spawns the SPE program which implements the algorithm on the given number of the SPEs i.e. from 1 SPE to 6 SPEs. After this, the PPE thread waits for the SPE threads to finish their tasks. The running SPE threads then initiates the DMA transfer for their part of the image data from the main memory to their respective Local Store and perform computation on the image data in their Local Store and finally send back the results to the main memory from their Local Store by again initiating DMA transfer. The PPE thread resumes when all the SPE threads have finished their work and write back the filtered image back to the disk and exit. The task division is done by partitioning the image data into num_spe equal portions, where num_spe is the given number of SPEs. Each SPE gets the (m/num_spe rows) of the total m rows of the m x n image data for computation.

4.1 Simple Averaging Filter

Simple Averaging Filter is a simplest image filtering technique in spatial domain. Simple Averaging Filter is widely used for removing uniform type of noise as well as Gaussian noise. The filtering in the spatial domain involves convolving the noise image with a mask or window of size w x w [19], where this w is normally odd integer such as 3, 5, or 7. Rarely does it get larger than these sizes. Figure 4.2 explains the filtering the image in spatial domain with mask size of 3 x 3.

At the beginning when the mask is overlaid onto the image,w_5 (center of the filter mask) will be aligned with z_5 (pixel in the image to be filtered). The value for z_5 will

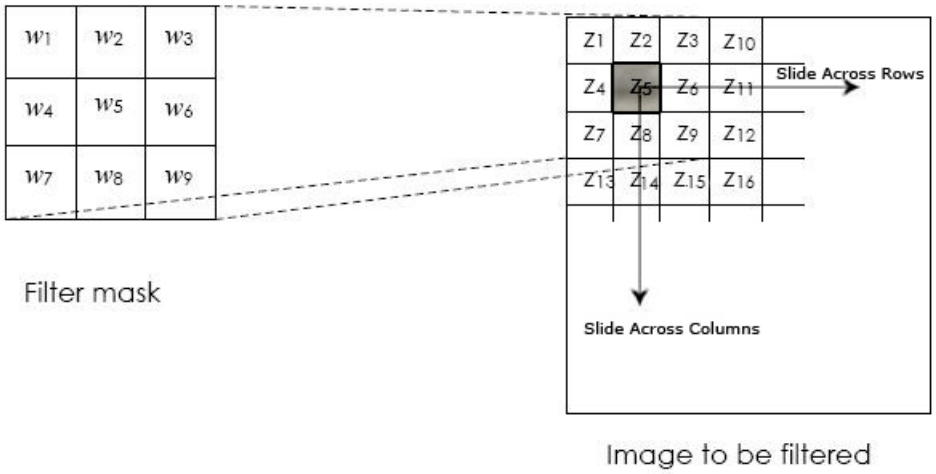

Figure 4.2: Filtering in Spatial Domain

be replaced by a value which is computed as shown in (4.1).

$$\begin{aligned} R = &(w_1 * z_1) + (w_2 * z_2) + (w_3 * z_3) \\ &+ (w_4 * z_4) + (w_5 * z_5) + (w_6 * z_6) \\ &+ (w_7 * z_7) + (w_8 * z_8) + (w_9 * z_9) \end{aligned} \tag{4.1}$$

Once this is done, the filter mask will be shifted to the left so that its center, w_5, is now aligned with z_6 and similarly the value of z_6 will be computed. This process will be repeated by moving the mask across the whole image and thus computing the new pixel values of the filtered image in a different array.

The filter mask for simple averaging filter is as shown in Figure 4.3. The kernel of this filter consists of constant 1. The scaling factor of $\frac{1}{9}$ is to guarantee that the result of summing up the point-wise multiplication will not run off the allowable dynamic range of the intensity level. For example for an 8-bit image, the scaling factor will ensure that the resulting intensity will not exceed 255. As the filter mask consists of

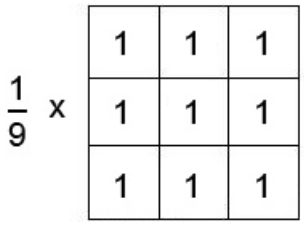

Figure 4.3: 3 x 3 Mask of Simple Averaging Filter

all 1s, the multiplication between 1s of the filter mask and image pixels is not done. Hence in this study, the simple averaging filter only does the addition of the image pixels according the mask size specified.

The simple averaging filter implemented in this study keep the boundary pixels of the original image unchanged where the filter mask crosses past the boundary of the original image. The scalar version of the simple averaging filter is developed on Intel Core 2 Duo Processor with Fedora Core 7 operating system. The code is written in C Language and compiled without any optimization with standard gcc compiler available with the Fedora Core 7 operating system. The scalar version and SIMDized version of the simple averaging filter for CBE PPE only is written in C and compiled with ppu-gcc compiler [14] available with Cell SDK 3.0. To implement the SIMDized version of simple averaging filter for CBE PPE only, the algorithm is restructured in such a manner so that it can take advantage of SIMD architecture of the processor by using Vector/SIMD Multimedia Extension Instruction Set of the PPE.

The important steps in SIMDization of the Simple Averaging Filter are highlighted in Figure 4.4 for image size of m x n and mask size of w x w. The m x n image is first restructured into 8 element vectors of type vector unsigned short int. The 8 elements with uniform shade in Figure 4.4 indicate one vector each. The pointer called *ptr* of type vector unsigned short int * points to the first vector of the image which contains 8 elements with indices from (0, 0) to (0, 7). Then the pointer $(ptr + 1)$ points to the

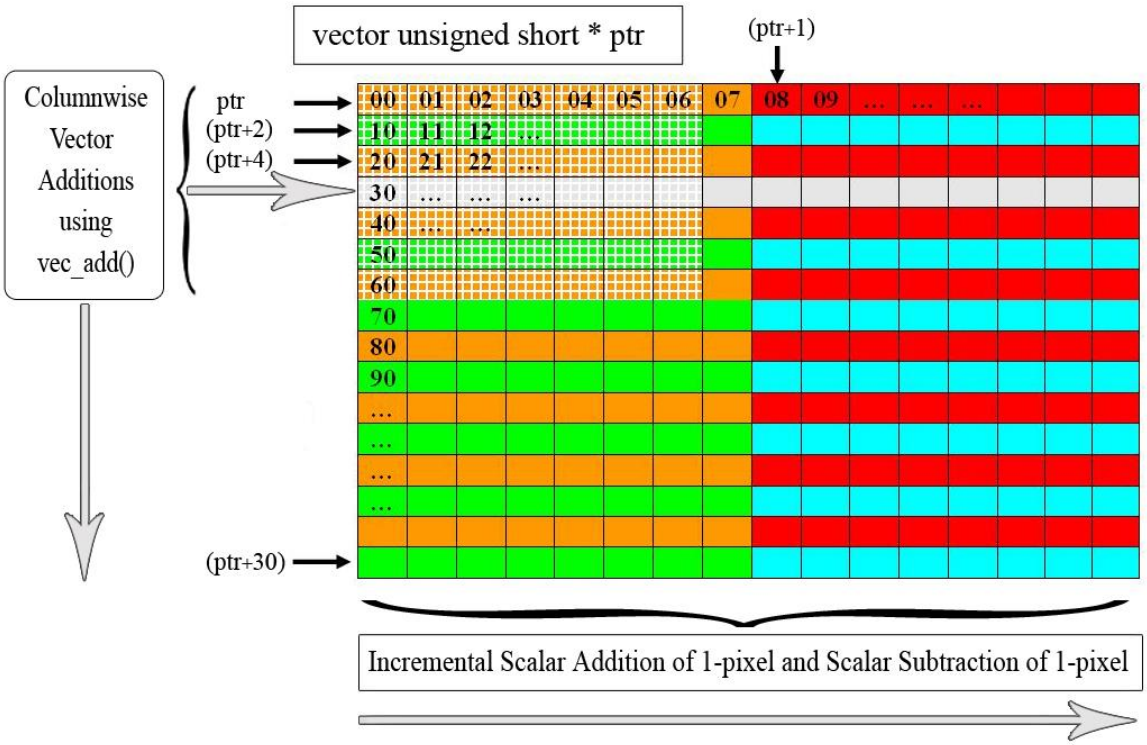

Figure 4.4: SIMDization of Simple Averaging Filter

second vector of the image which contains 8 elements with indices from (0, 8) to (0, 15). For example, a 16 x 16 image can be represented by 32 vectors of type vector unsigned short int so that all the pixels of the image can be accessed by offsetting the pointer *ptr* from $(ptr + 0)$ to $(ptr + 31)$. If the total number of columns - n in an image is not a multiple of 8, then appropriate numbers of columns are zero padded so that each starting pixel of the row will be the first element of 8 element vector. This constraint is added intentionally to avoid use of vector permutation and shuffle operation to manually align the image pixels while performing vector operations on it. Then as shown in Figure 4.4, for mask size of w x w, first w vectors of the first vector column of the image are added using vec_add() instruction. After that first w vectors of the second vector column, third vector column and so on up to the last vector column are added using vec_add() instruction. The result of these vector additions are stored at their respective column positions at the center row of the mask and image overlap area. The vec_add(a,b) instruction is a SIMD vector addition instruction that

(m x n Image, w x w Mask)	Scalar Additions	Vector Additions
Scalar Version	$(m-w+1)$ $*(n-w+1)$ $*(w*w)$	—
SIMDized Version	$(2n-w)$ $*(m-w+1)$	$w*(m-w+1)*(n/no.\ of$ $elements\ in\ single\ vector)$

Table I: Formulated Computation Statistics of Simple Averaging Filter

can perform 8 scalar additions in parallel on vector elements of unsigned short int vectors a and b [20]. Now, first w elements of the center row of the mask and image overlap area are added using scalar addition by extracting vector elements by using vec_extract() instruction and after that scaled by dividing the sum with $(w*w)$. This result is placed at the center column of the mask and image overlap area in the new filtered image array using vec_insert() instruction. After that the mask is moved right one column and the sum is computed by adding (w + 1)th element and subtracting 0^{th} element. The scaled result obtained by dividing the sum with $(w*w)$ is placed at the center of mask and image overlap area. In this way new pixel values are computed by adding 1 pixel value and subtracting 1 pixel value instead of performing seven vector additions at each step. When all the pixel values of the given row are computed, the above steps are repeated by moving the mask down one row. Table I shows the formulated computation statistics of the scalar version and restructured SIMDized version of simple averaging filter for image size of m x n and mask size of w x w. Table II shows the numerical values of computation statistics of the same for image size of 16 x 16 and mask size of 7 x 7.

The Altivec library available with Cell SDK 3.0 is required to perform the vector operations on PPE. The Misc library available with Cell SDK 3.0 is required to

(16 x 16 Image, 7 x 7 Mask)	Scalar Additions	Vector Additions
Scalar Version	4900	—
SIMDized Version	250	140

Table II: Computation Statistics of Simple Averaging Filter

allocate and free memory aligned on 16-byte boundary. Two functions in the Misc library are used - malloc_align() function allocates memory on 16-byte boundary and free_align() function free the allocated memory using malloc_align() function [12]. This alignment of data on CBE is necessary on both PPE and SPE to perform vector operations.

The SIMDized version of simple averaging filter for CBE PPE and multiple SPEs also works in similar way as explained in Figure 4.4. But the computation is done by the SPE threads on the data in their respective Local Store. Hence the simple averaging filter is redesigned to handle the transfer of data from PPE's main memory to each Local Store and sending back the computed results to the main PPE thread. Two different programs are developed to run on the PPE and the SPEs respectively. PPE program is compiled with ppu-gcc. SPE program is compiled with spu-gcc. Then the ppu-embedspu tool is used to link the SPU binary with the Power Processor Unit (PPU) binary into a single executable program [14]. A PPE module starts an SPE module running by creating a thread on the SPE, using the spe_context_create, spe_program_load, and spe_context_run library calls, provided in the SPE runtime management library [18]. The threads are created by PPE using standard pthread library. The Misc library functions are used by PPE and SPE to allocate memory aligned at 16-byte boundary. The SPU Intrinsics library is used by the SPE program to support the vector operations on the SPEs. The Memory Flow Controller (MFC) input and output macros [18] are used by the SPE program to handle DMA transfers between main memory and the local store. The SPU vector instructions like

spu_add(), spu_extract() and spu_insert() are used to perform vector operations on SPE corresponding to the PPU vector instructions like vec_add(), vec_extract() and vec_insert() [20].

For mask size of w x w, each SPE requires $(w-1)$ extra rows to be transferred to their Local Store in order to compute results on their share of rows of image data. The Local Store holds only up to 256 KB for the program, stack, local data structures, and DMA buffers. Also the maximum DMA transfer can be up to 16 KB in size. Hence all the rows required to be processed by the SPE cannot be DMA transferred or stored in the Local Store together. Considering the above limitation and the algorithm structure, each SPE initiates DMA transfer of single vector at a time. Some memory is allocated by each SPE to store 1 row of temporary results and 1 row of computed results into the local store. The final results are transferred to the PPE one row at a time by initiating DMA transfer by the SPE.

4.2 Mean Normalized Digital Cross Correlation

Cross-correlation is the basic statistical approach for image matching. It is often used for template matching or pattern recognition in which the location and orientation of a template or pattern is found in an image. Cross-correlation is similarity measure or match metric, i.e. it gives a measure of the degree of similarity between an image and a template [21]. The cross correlation process for template matching is shown in Figure 4.5.

The cross correlation coefficient R between two windows - Search Area (SA) s of size M x M and Window Area (WA) w of size N x N, ($N \leq M$), has to be computed for $(M-N+1)^2$ shift positions. To take into account any spatially invariant linear, radiometric relationship between sample values of the WA and SA mean-normalized digital cross correlation is generally used. Then the cross correlation coefficient R is

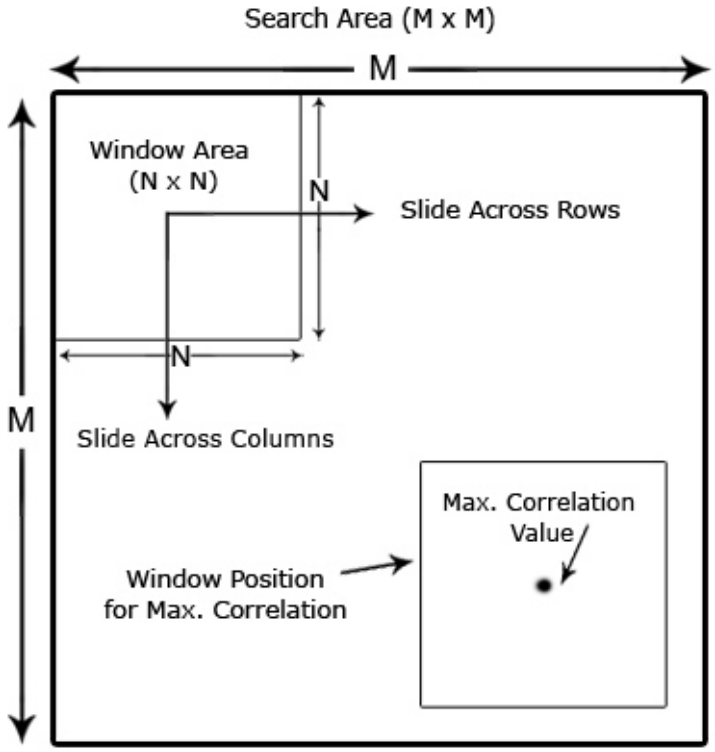

Figure 4.5: Digital Cross Correlation Process

defined by (4.2).

$$R(u,v) = \frac{\sum_{x=1}^{N}\sum_{y=1}^{N}(s(u+x,v+y)-\overline{s}_{uv})(w(x,y)-\overline{w})}{\sqrt[2]{[\sum_{x=1}^{N}\sum_{y=1}^{N}(s(u+x,v+y)-\overline{s}_{uv})^2\sum_{x=1}^{N}\sum_{y=1}^{N}(w(x,y)-\overline{w})^2]}} \quad (4.2)$$

$u, v = 0, 1, 2, \ldots, M - N$

The $s(x, y)$ and $w(x, y)$ are pixel values at location (x, y) of s and w respectively. The value of R in (4.2) lies in the range between -1 to +1, and the closer R is to +1, the more similar the two windows will be.

Mean normalized digital cross correlation algorithm is highly compute intensive

than simple averaging filter and requires large number of floating point multiplication, addition and subtraction operations on the image data. Hence this kind of algorithm is more suitable to gain programming and performance insight in developing image processing algorithm on Cell Broadband Engine.

All the scalar and SIMDized versions of mean normalized digital cross correlation algorithm for running on Intel Core 2 Duo Processor, CBE PPE only and CBE PPE and multiple SPEs, are developed and compiled in similar lines to the corresponding version of simple averaging filter. The standard math library is also used in addition to all the libraries used for the PPE and SPE programs of simple averaging filter. The Search Area is zero padded by appending columns so that the number of columns in the Search Area can be in multiple of the number of elements in the single vector operand used for computations. Similarly, the number of columns in the selected Window Area is also in multiple of number of elements in the single vector operand used for computation in order to avoid complexities in the SIMDized version of the algorithm. The mean of the Window Area is computed on the PPE and DMA transferred to each SPE for both scalar and SIMD version of algorithm to run of CBE PPE and multiple SPEs. This value is required by all the SPEs to compute the cross correlation coefficient. Computing the mean of Window Area on each SPE will incur redundant computation overhead in each SPE. For window size of N x N, each SPE requires $(N-1)$ extra rows to be transferred to their Local Store in order to compute results on their share of rows of image data. The DMA transfer of size of single row of Window Area is initiated to fetch the data of Window Area and Search Area into the local store from main memory for computation by the SPEs in both scalar and SIMD version to run on CBE PPE and multiple SPEs. The results are DMA transferred back to the main memory from Local Store to main memory one vector at a time by the SPE. Due to the limitation of the size of Local Store in SPEs, the Search Area data is DMA transferred twice to the SPE - first time to compute mean of the Search Area and second time to compute the cross correlation coefficient. The

spu_shuffle() instruction is used by SPE and vec_perm() [22] instruction is used by PPE to reorganize the data of the Search Area in SIMD version of algorithm to run on CBE PPE and multiple SPEs and CBE PPE only respectively. This is required as the Search Area data does not remain aligned on vector boundary as the window is moved over the Search Area. The other SIMD instructions used by the SPE program are spu_extract(), spu_insert(), spu_splats(), spu_sub() and spu_madd() . The corresponding SIMD instructions used by the PPE program are vec_extract(), vec_insert(), vec_splats(), vec_sub() and vec_madd() [20]. The spu_madd() and vec_madd() are multiply and accumulate vector instruction which gives significant performance gain on SIMD architectures. The other issues in implementation of the mean normalized cross correlation algorithm like aligned memory allocation, etc. are similar to the implementation issues of simple averaging filter.

4.3 Optimization of Mean Normalized Digital Cross Correlation Algorithm by Double Buffering Technique

The SIMD version of mean normalized digital cross correlation SPE program use DMA transfers to move large amount of data between main storage and the local store (LS) in the SPE. The following simple scheme is used to achieve that data transfer in SIMD version of algorithm without optimization:

a. Start a DMA data transfer from main storage to buffer B in the LS.

b. Wait for the transfer to complete.

c. Use the data in buffer B.

d. Repeat.

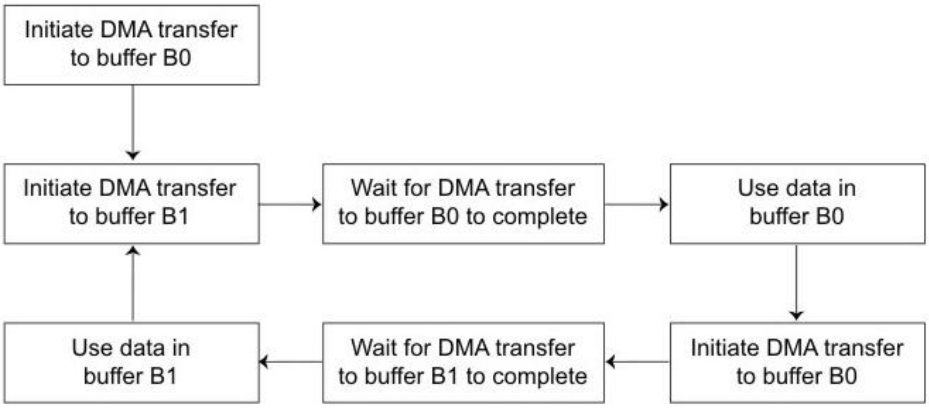

Figure 4.6: DMA transfers using a Double-Buffering Method

This method wastes a great deal of time waiting for DMA transfers to complete. In the optimized Double Buffered SIMD version of algorithm, the process is being speed up significantly by allocating two buffers, *B0* and *B1*, and overlapping computation on one buffer with data transfer in the other. This technique is called double buffering [18]. Figure 4.6 shows a flow diagram for this double buffering scheme.

Double buffering is a form of multibuffering, which is the method of using multiple buffers in a circular queue to overlap processing and data transfer. The purpose of double buffering is to maximize the time spent in the compute phase of a program and minimize the time spent waiting for DMA transfers to complete. Let T_t represent the time required to transfer a buffer B, and let T_c represent the time required to compute on data contained in that buffer. In general, the higher the ratio (T_t/T_c), the more performance benefit an application will realize from a double-buffering scheme [18].

The optimized double buffered SIMD version of algorithm gives substantial performance gain over SIMD version of algorithm without any optimization. The double buffered SIMD version of algorithm is developed by modifying the earlier SIMD version of algorithm with very few programming efforts like using two dimensional arrays to store two rows each of the Search Area and the Window Area, and using unique

DMA tag IDs for each logical group of row buffers. Each logical group of row buffers contains one row of Search Area and one row of Window Area. Then the program is modified and implemented according to the double buffering technique shown in Figure 4.6.

Chapter 5

Results and Discussions

5.1 Simple Averaging Filter

The functional results of Simple Averaging Filer for Image Size of 256 x 256 and Mask Size of 7 x 7 are shown in Figure 5.1. The pixels at the border of the filtered image that are kept unchanged can be seen in the output image clearly at the bottom and at two sides in Figure 5.1.

The functional results of Simple Averaging Filter for Image Size of 1028 x 732 and Mask Size of 7 x 7 are shown in Figure 5.2. The Mask Size of 7 x 7 blurs the original significantly, but this Mask Size is intentionally chosen to increase the computation in Simple Averaging Filter.

The results of both the images have been verified and matched by using the imfilter() function of the Matlab, except at the border pixels that are kept unchanged in our implementation.

The performance of different versions of algorithm on Intel Core 2 Duo Processor and Cell Broadband Engine Processor is compared by measuring the time of execu-

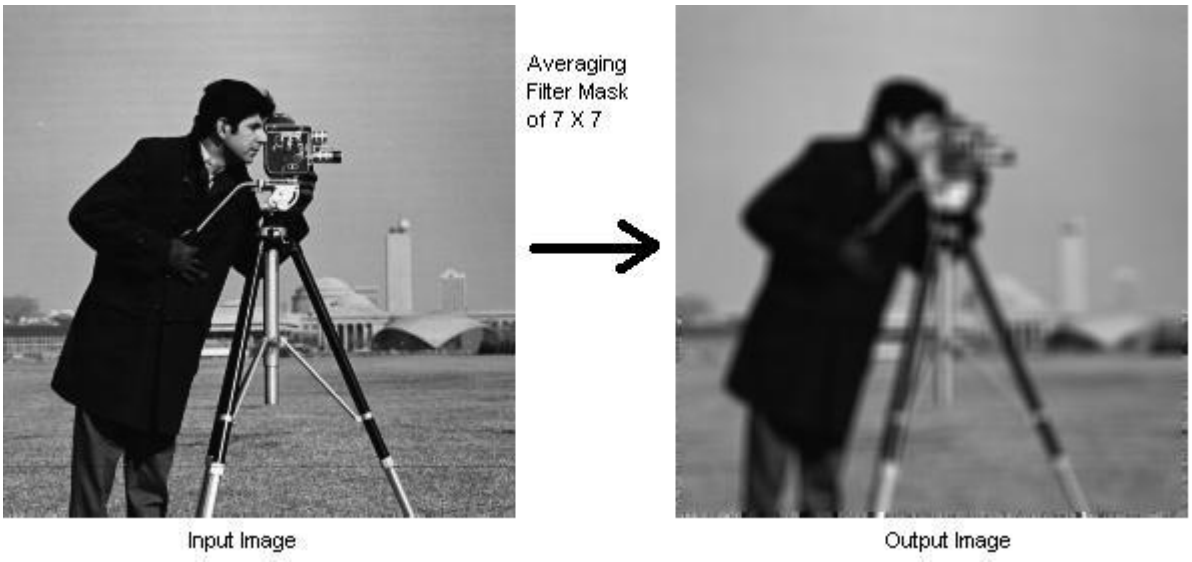

Figure 5.1: Functional Result of Simple Averaging Filter for Image Size of 256 x 256 and Mask Size of 7 x 7

tion of the program by using gettimeofday() function available in sys/time.h library of the C language. The time of reading the image from the disk and writing back the results to the disk are not included in the measured timings. Table I shows the measured execution time in micro seconds of different versions of simple averaging filter for image size of 256 x 256 and mask size of 7 x 7.

Figure 5.3 shows the graph of Performance Measure of Simple Averaging Filter for Image Size of 256 x 256 and Mask Size of 7 x 7. It can be clearly seen from Figure 5.3 that the maximum performance gain of 3X is obtained on SIMD version of algorithm running on CBE PPE only than Intel Core 2 Duo Processor. Also the SIMD version of algorithm running on CBE PPE only is 2.15 times faster than corresponding scalar version running on CBE PPE only. The important feature of CBE hybrid multicore architecture is highlighted in the performance results of SIMD version of algorithm running on CBE PPE and multiple SPEs. The maximum performance gain of 2.34X over Intel Core 2 Duo Processor was obtained by running the SIMD version of algorithm on CBE PPE and 3 SPE cores. Thereafter increasing the SPE cores leads

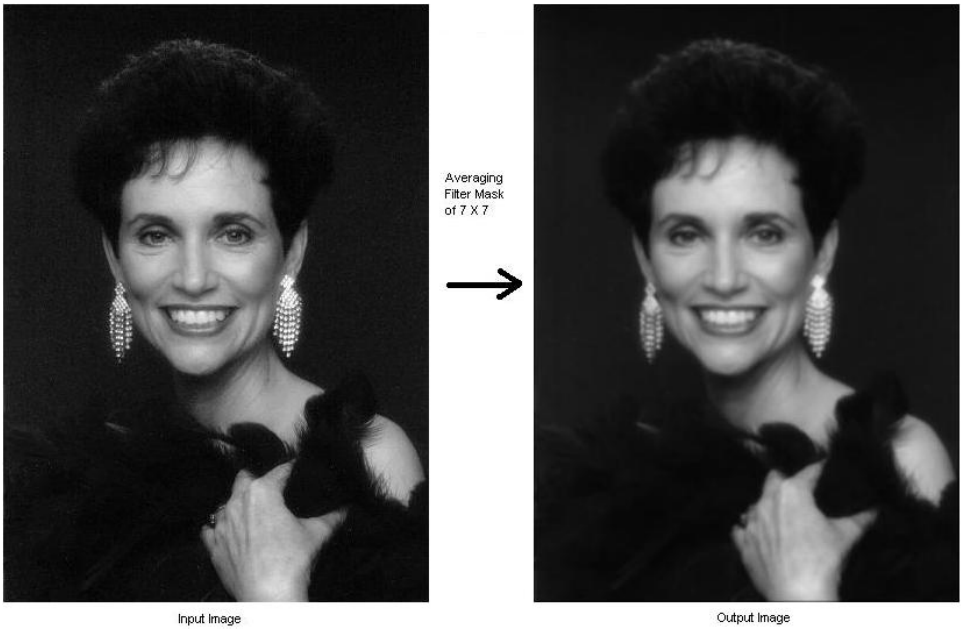

Figure 5.2: Functional Result of Simple Averaging Filter for Image Size of 1028 x 732 and Mask Size of 7 x 7

to decrease in performance gain gradually to 1.97X for algorithm running on CBE PPE and 6 SPEs. The reason for this decrease in performance gain by adding more SPE cores is that Simple Averaging Filter is less compute intensive and after the image data is divided on multiple SPE cores, each SPE core is left with very little computational load and then the data transfer between the main memory and the Local Store of the SPEs become the major bottleneck in performance.

Table II shows the execution time of different version of different versions of Simple Averaging Filter for Image Size of 1028 x 732 and Mask Size of 7 x 7.

Figure 5.4 shows the graph of performance measure of different versions of Simple Averaging Filter for Image Size of 1028 x 732 and Mask Size of 7 x 7. This image contains much larger number of rows and columns than previous image of size 256 x 256. Hence maximum performance gain is achieved when we run the algorithm on maximum number of SPE cores on CBE. This can be clearly seen in Figure 5.4. Also

Algorithm Version	*Time (Micro Seconds)*
Scalar Version on Intel Core2Duo	16994
Scalar Version on CBE PPE	12004
SIMD Version on CBE PPE	5570
SIMD Version on PPE + 1 SPE	12479
SIMD Version on PPE + 2 SPE	8150
SIMD Version on PPE + 3 SPE	7251
SIMD Version on PPE + 4 SPE	7288
SIMD Version on PPE + 5 SPE	7551
SIMD Version on PPE + 6 SPE	8592

Table I: Performance Measure of Simple Averaging Filter for Image Size: 256 x 256 and Mask Size: 7 x 7

Algorithm Version	*Time (Micro Seconds)*
Scalar Version on Intel Core2Duo	193681
Scalar Version on CBE PPE	145059
SIMD Version on CBE PPE	63987
SIMD Version on PPE + 1 SPE	126533
SIMD Version on PPE + 2 SPE	64690
SIMD Version on PPE + 3 SPE	45295
SIMD Version on PPE + 4 SPE	35352
SIMD Version on PPE + 5 SPE	30392
SIMD Version on PPE + 6 SPE	27396

Table II: Performance Measure of Simple Averaging Filter for Image Size: 1028 x 732 and Mask Size: 7 x 7

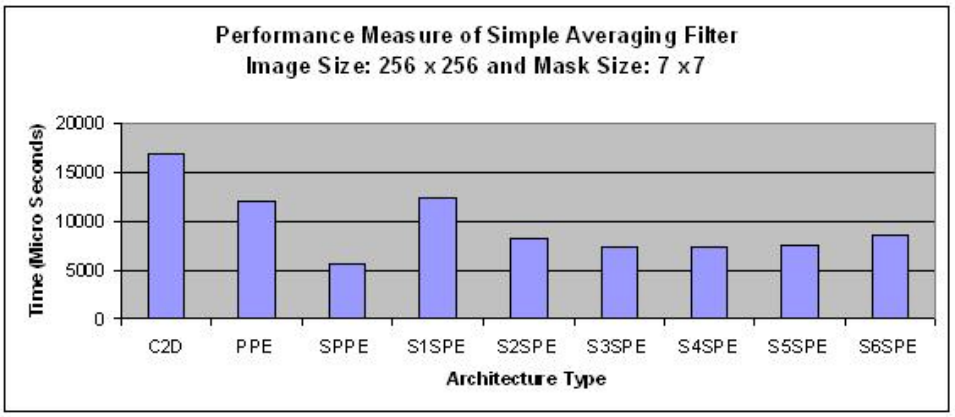

Figure 5.3: Performance Measure of Simple Averaging Filter for Image Size of 256 x 256 and Mask Size of 7 x 7

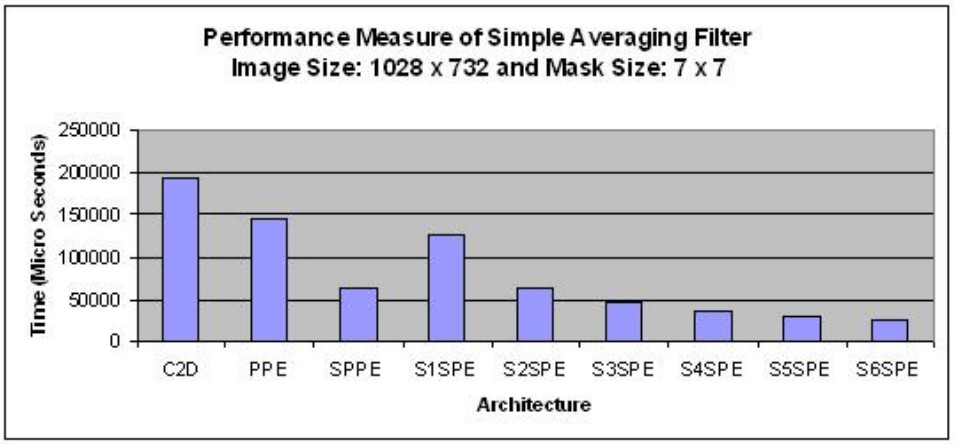

Figure 5.4: Performance Measure of Simple Averaging Filter for Image Size of 1028 x 732 and Mask Size of 7 x 7

the SIMD version of algorithm running on CBE PPE only is 2.26 times faster than the corresponding scalar version.

Figure 5.5 shows the graph of performance gain of Simple Averaging Filter algorithm on Cell Broadband Engine Processor compared to Intel Core 2 Duo Processor for Image Size of 1028 x 732 and Mask Size of 7 x 7. The maximum performance gain achieved is 7.07X for the SIMD version of algorithm running on CBE PPE and 6 SPE cores.

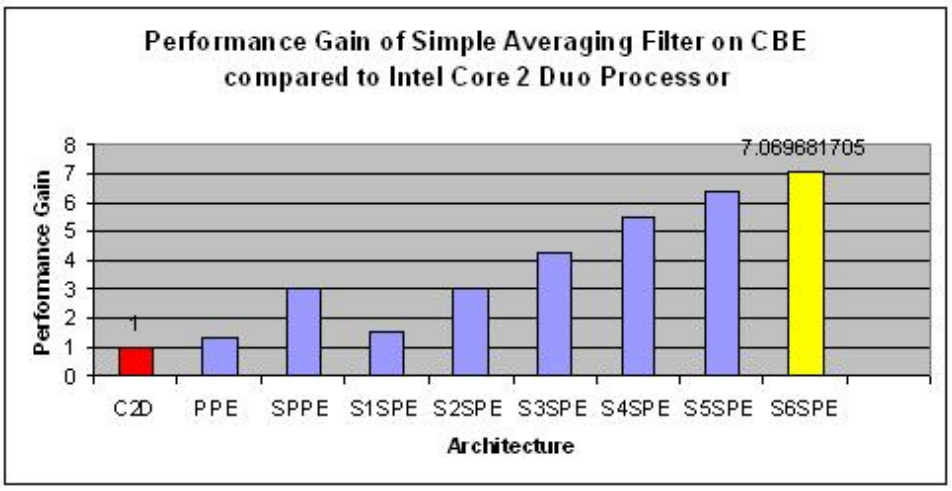

Figure 5.5: Performance Gain of Simple Averaging Filter on CBE compared to Intel Core 2 Duo Processor

Figure 5.6 shows the SPE Timing Statistics of the SIMD version of Simple Averaging Filter running on CBE PPE and 6 SPE cores. This SPE Timing Statistics were obtained by profiling the code on IBM Full-System Simulator. The stalls due to branch miss and the channel stall cycles of the Simple Averaging Filter are very high as shown in Figure 5.6. Thus still optimizing the SIMD version of Simple Averaging Filter algorithm by using techniques such as Double Buffering, Loop Unrolling, etc. can improve the performance of the algorithm.

5.2 Programming Notes on Simple Averaging Filter

The following observations are made in programming Cell Broadband Engine of Sony PlayStation 3 for Simple Averaging Filter algorithm:

- The Simple Averaging Filter algorithm applied on small images with small Mask Size like 3 x 3 or 5 x 5 has less computational load. Hence running Simple Averaging Filter in such scenarios on all the available SPE cores actually

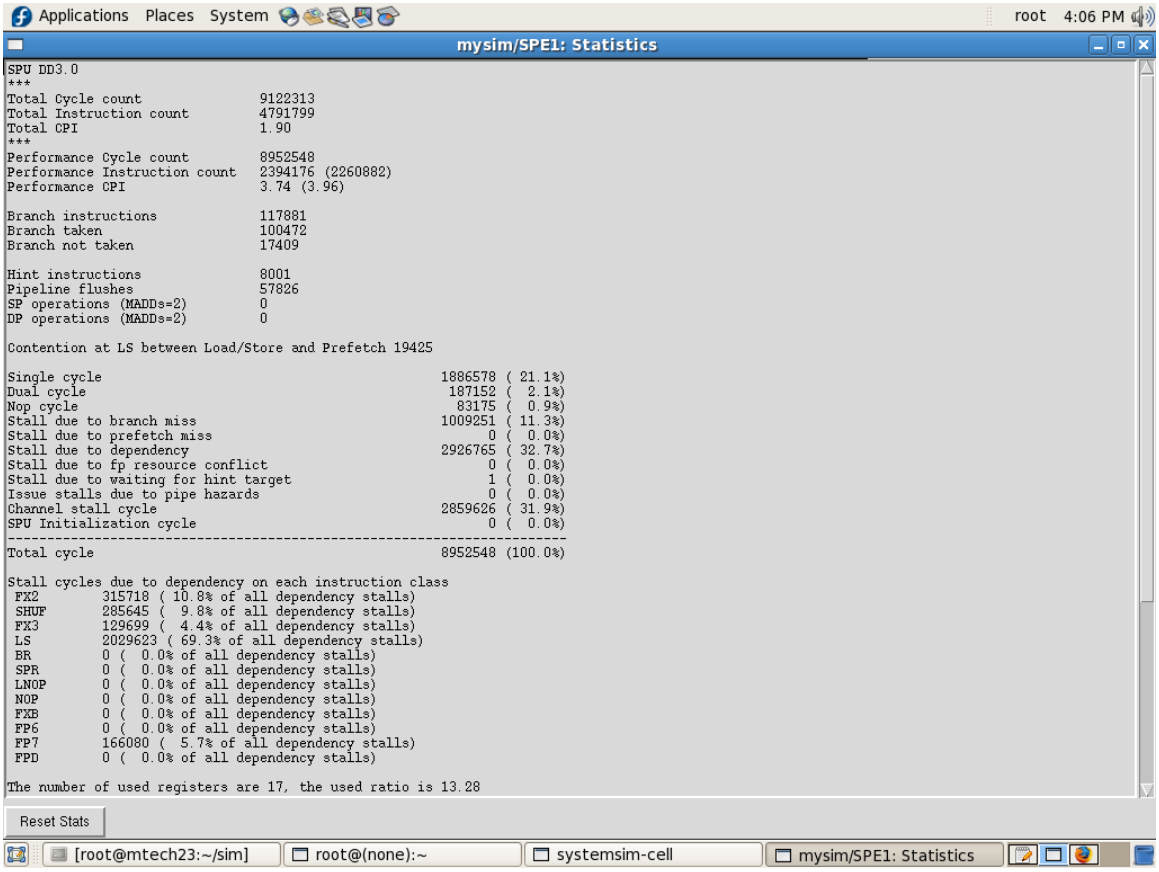

Figure 5.6: SPE Timing Statistics for SIMD Simple Averaging Filter

degrades the performance because of bottleneck of data transfer between the main memory and the Local Store of the SPEs.

- To achieve the maximum performance gain for the algorithm, optimal number of SPE threads must be spawned depending on the Image Size and Mask Size. The optimal number of SPE threads for different sizes of image and mask must be determined after prior benchmarking of the algorithm with images and mask of different sizes.

- Higher performance gain is achieved by SIMDization of the algorithm as the SPEs of the CBE are SIMD processors. The manual optimization of the algorithm produces more optimal code than auto-vectorizing compilers.

- The current SIMD implementation of the algorithm can be optimized by re-

ducing the column wise vector additions. Only one incremental vector addition and subtraction is required instead of w vector additions for mask size of w x w. This will reduce the computational load of the algorithm itself.

- The major bottleneck of the algorithm is the DMA transfer between the main memory and local store. This can be optimized by using double buffering techniques.
- Larger size of DMA transfers gives better performance than DMA transfers of very small size.
- The CBE of PlayStation 3 takes significantly more time in disk I/O operations than Intel Core 2 Duo Processors.

5.3 Mean Normalized Digital Cross Correlation

The functional results of the Mean Normalized Digital Cross Correlation algorithm for Search Area of size 424 x 432 and Window Area of size 64 x 48 are shown in Figure 5.7. The brightest spot is seen in the output image where the object in the Window Area strongly matches the object in the Search Area and the cross correlation coefficient at that pixel has the maximum value. The functional results of Mean Normalized Digital Cross Correlation algorithm are verified and matched with the results obtained by corr2() function of the Matlab.

Table III shows the measured execution time of different versions of Mean Normalized Cross Correlation algorithm for Search Area of 424 x 432 and Window Area of 64 x 48. The execution time is measured in micro-seconds using gettimeofday() function of sys/time.h library in C Language. The time required to read the input image from disk and to write back the results to the disk are not included in the measured timings.

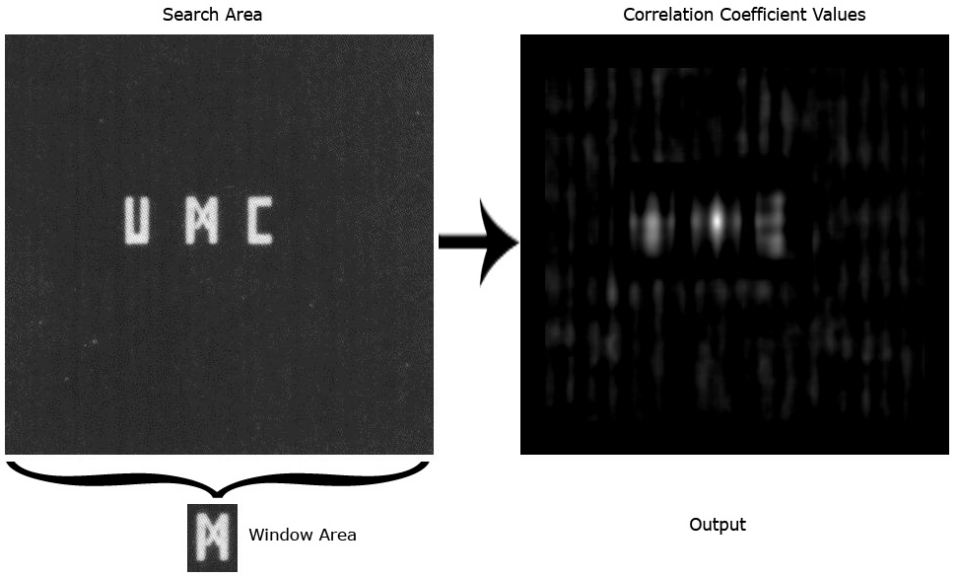

Figure 5.7: Functional Result of Mean Normalized Digital Cross Correlation for Search Area: 424 x 432 and Window Area: 64 x 48

The effect on performance of cross correlation algorithm by using different data types for storing image pixels can be clearly seen from the graph of Figure 5.8 and the timing values in Table III. The computation involved in the cross correlation algorithm requires floating point operations and the image pixels can be stored in minimum memory space by using unsigned char or unsigned short int data types. When the image pixels on CBE PPE are stored using unsigned short int data type, the performance of the algorithm is degraded severely because PPE has to convert the unsigned short int operands to floating point operands and align them before performing computation. This conversion takes considerable time in CBE than in Intel Core 2 Duo Processors. Also the double floating point operations on CBE takes considerable time compared to single floating point operations. Hence in simple SIMD and double buffered SIMD version of the algorithm, single floating point vector operands are used. This compromise with the accuracy of the results is done to achieve better performance.

Figure 5.9 shows the graph of performance measure of Mean Normalized Digital

Algorithm Version	*short int*	*Single Buffering (float)*	*Double Buffering (float)*
Scalar Version on Intel Core2Duo	11190000	-	-
Scalar Version on CBE PPE	33390000	3990000	-
Scalar Version on PPE + 1 SPE	-	14152563	-
Scalar Version on PPE + 2 SPE	-	7130102	-
Scalar Version on PPE + 3 SPE	-	5594991	-
Scalar Version on PPE + 4 SPE	-	4239463	-
Scalar Version on PPE + 5 SPE	-	3378727	-
Scalar Version on PPE + 6 SPE	-	2833243	-
SIMD Version on CBE PPE	-	1790000	-
SIMD Version on PPE + 1 SPE	-	5763532	3151503
SIMD Version on PPE + 2 SPE	-	3026199	1694075
SIMD Version on PPE + 3 SPE	-	2414738	1363968
SIMD Version on PPE + 4 SPE	-	1869236	1076704
SIMD Version on PPE + 5 SPE	-	1518370	895030
SIMD Version on PPE + 6 SPE	-	1296795	783178

Table III: Performance Measure of Mean Normalized Cross Correlation for Search Area: 424 x 432 and Window Area: 64 x 48

Cross Correlation algorithm for Search Area of 424 x 432 and Window Area of 64 x 48. The SIMD version of algorithm running on CBE PPE only is 2.22 times faster than the corresponding scalar version. The SIMD version of algorithm running on CBE PPE and 1 SPE is 2.45 times faster than the corresponding scalar version. The cross correlation algorithm is highly compute intensive and hence increasing the SPE cores to its maximum (6 in our case) results in increase in performance gain of the algorithm.

Figure 5.10 shows the comparison of performance measure of the simple SIMD

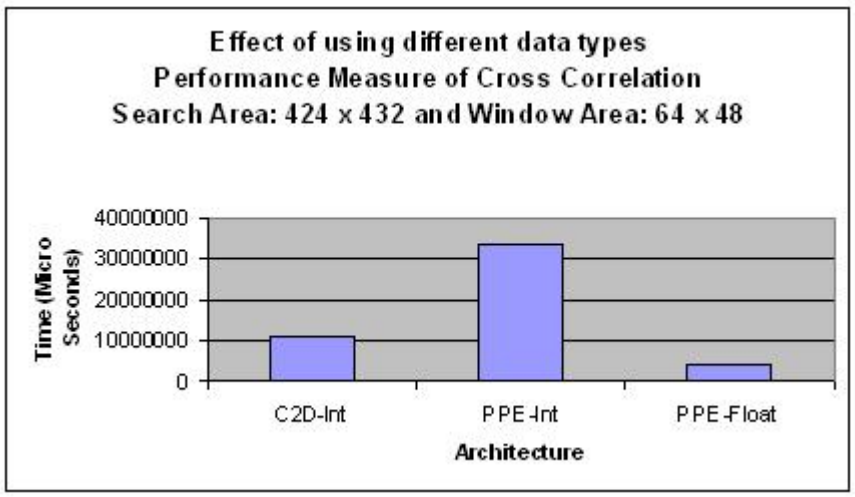

Figure 5.8: Effect on Performance of Cross Correlation by using different data types

version and the optimized Double Buffered SIMD version of the Cross Correlation Algorithm for Search Area of 424 x 432 and Window Area of 64 x 48. The double buffered SIMD version of the algorithm is nearly 2 times faster than the simple SIMD version. The performance gain of double buffered SIMD version over simple SIMD version of algorithm is between 1.83X to 1.66X.

Figure 5.11 shows the graph of performance gain of Mean Normalized Cross Correlation algorithm on CBE Processor compared to Intel Core 2 Duo Processor. The maximum performance gain of 14.29X is obtained for the double buffered SIMD version of algorithm running on CBE PPE and 6 SPEs.

5.4 Programming Notes on Mean Normalized Digital Cross Correlation

The following observations are made in programming Cell Broadband Engine of Sony PlayStation 3 for Mean Normalized Digital Cross Correlation algorithm:

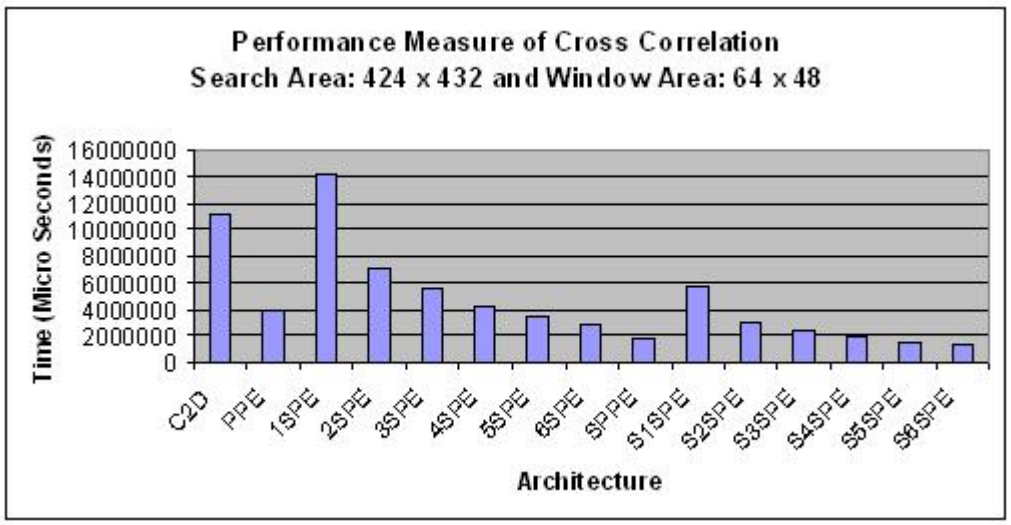

Figure 5.9: Performance Measure of Mean Normalized Cross Correlation for Search Area: 424 x 432 and Window Area: 64 x 48

- There is much scope in optimizing the algorithm itself to decrease the computational load in the current implementation by considering only the pixels in the new column and new row for computation after shifting the position of window area.

- Due to the limitation of size of Local Store of SPE to 256KB, the window area of even moderate size and the portion of search area overlapped by the window area cannot be stored in the Local Store of the SPE simultaneously. This results in transferring the same image data of Search Area twice - first time to compute mean and second time to compute the difference between the mean the pixel value.

- The high computation requirement of mean normalized cross correlation algorithm than its data transfer from main memory to Local Store leads to performance gain of 14.29X on CBE. The performance gain of only 7.07X is obtained for Simple Averaging filter which has the similar data transfer requirement as cross correlation algorithm, but the computational requirement is very low than cross correlation algorithm. Hence, the algorithms with higher computational

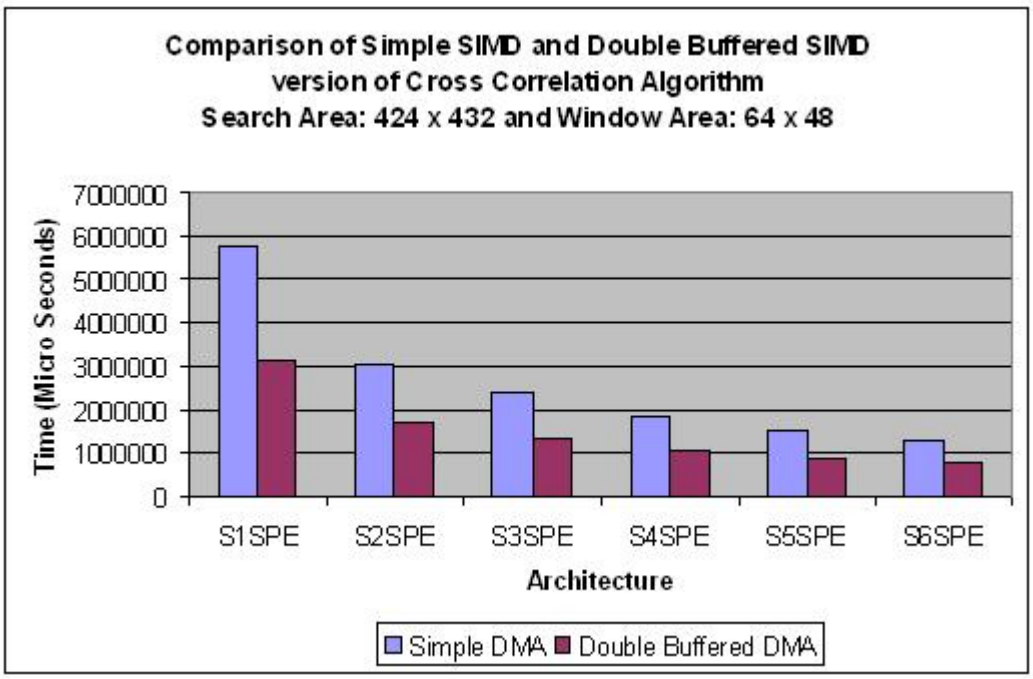

Figure 5.10: Performance Comparison of Simple SIMD and Double Buffered SIMD versions of Cross Correlation Algorithm

load on its data achieve higher performance gain on CBE.

- The maximum performance gain is achieved by manual SIMDization of the algorithm.
- The double buffering optimization technique nearly doubles the performance gain of the algorithm than the algorithm without double buffering.
- The type conversion of scalar variables into data types of different sizes (for example, short int to float) degrades the performance of CBE severely.
- As the cross correlation algorithm requires floating point operations, the data type of vectors for storing image data is chosen as float. This uses 4 times more memory than the actual memory requirement for storing the image data. But now the computation between the vectors of similar types can be done is straightforward manner. If memory was saved by using data type of smaller size for storing image data, the vector should be manually converted to float by using

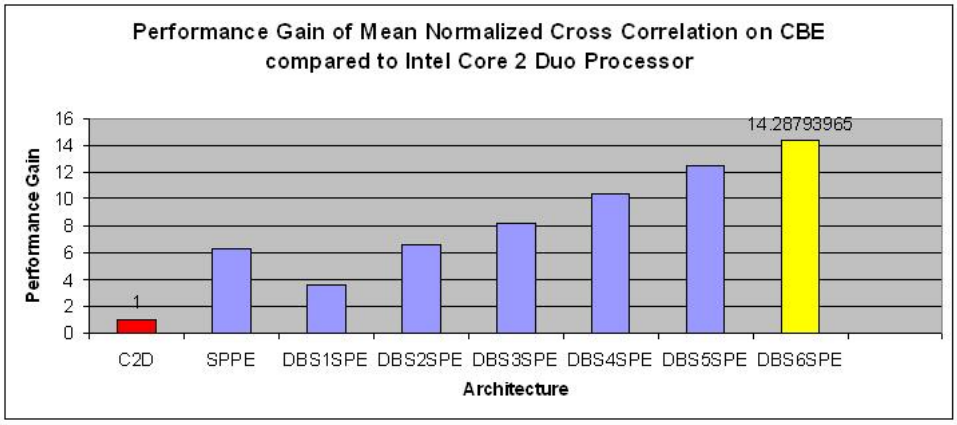

Figure 5.11: Performance Gain of Mean Normalized Cross Correlation on CBE compared to Intel Core 2 Duo Processor

vec_perm() or spu_shuffle() instruction to perform floating point operation thus degrading the performance. Hence while programming CBE, the programmer has to do tradeoff between the memory and performance requirements.

- The double floating point operation on CBE is accurate but very slow. The single floating point operation on CBE is fast but less accurate.

5.5 Suggested Strategies for Designing Image Processing Algorithms on Cell Broadband Engine

The programming approach and strategies for designing Image Processing algorithms effectively on Cell Broadband Engine are summarized below:

- The PPE must be used as the control processor and all the computational work must be offloaded to SPEs to achieve performance enhancement on CBE.

- The manual SIMDization of the code is necessary to achieve the maximum performance gain on the Cell Broadband Engine. The appropriate SIMD strategy must be selected according to the algorithm. Evaluate the pros and cons of implementing the array-of-structure and structure-of-arrays organization for the

algorithm.

- The limitation of the maximum size of DMA transfer - 16KB and the maximum size of Local Store - 256KB should be kept in while programming CBE.

- Try to avoid using more SPE threads than physical number of SPEs because context switching consumes a fair amount of time.

- The optimal number of physical SPE cores to be used by the algorithm must be decided based on the computational load of the algorithm and size of the data passed to the algorithm by prior benchmarking. Adding more SPE cores than this optimal number will leave the SPE cores with very less computational work and the performance will start degrading.

- The data should be aligned or padded for efficient quadword accesses, using the aligned attribute.

- The casting between vectors is not recommended as none of these casts performs any data conversion and the bit pattern of the result is the same as the bit pattern of the argument that is cast.

- When any computation is required to be done between the scalar and a vector, the scalar must be promoted to vector before performing the computation to make such operations more efficient.

- The single floating point operations are less accurate but fast, while the double floating point operations are more accurate but too slow. Hence to have better performance, it is recommended to use the double floating point operations only when the accuracy provided by using the single floating point operations is not sufficient.

- The DMA transfer of data between the main memory and Local Store of the SPEs must be carefully designed in order to avoid the bottleneck in performance

of the algorithm. The DMA transfer must be boosted by using double buffering optimization technique. If possible, the size of DMA transfer must be multiple of 128 bytes for effective utilization of the bus and memory bandwidth.

- A 16 bytes alignment is mandatory for data transfer of more then 16 bytes while 128 bytes alignment is optional but provides better performance.
- Prefer to use SPE initiated DMA transfers versus PPE initiated DMA transfers because each SPE of the CBE and enqueue 16 DMA requests and there are 6 usable SPE cores available on CBE of Sony PlayStation 3.

5.6 Shortcomings and Drawbacks in Implementing Image Processing Algorithms on Cell Broadband Engine

The following shortcomings and drawbacks were observed while developing different versions of Simple Averaging Filter and Mean Normalized Cross Correlation algorithms on Cell Broadband Engine and after analyzing the performance results obtained by deploying these algorithms on Sony PlayStation 3:

- The manual SIMDization of the algorithm takes much time and efforts of the application designer, which leads to increase in design time of algorithm development on CBE.
- While programming for CBE, the programming efforts also increase due to considerations like efficient application partitioning and data transfer, optimized DMA operations, alignment of data, inter-processor communication, local store size limitations, etc.
- The single floating point operations are not accurate.
- The double floating point operations are very slow.

- The disk I/O operations by CBE processor of Sony PlayStation3 are too slow compared to Intel Core 2 Duo Processor.
- Sometimes redundant extraneous data have to be DMA transferred to SPE due to the limitation of the size of Local Store which in turn becomes the major performance bottleneck of the algorithm.

Chapter 6

Conclusion and Future Scope

6.1 Conclusion

The manual SIMDization of the algorithm is necessary to produce optimal code for Cell Broadband Engine. The data transfer between the SPE and PPE elements is the major performance bottleneck in Cell Broadband Engine and must be optimized by using techniques like double buffering. There is large number of application designing and programming considerations to be taken care of while developing image processing algorithms on Cell Broadband Engine.

The Simple Averaging filter algorithm on Cell Broadband Engine runs 7 times faster than Intel Core 2 Duo Processor. The mean normalized cross correlation algorithm runs 14 times faster on Cell Broadband Engine than Intel Core 2 Duo Processor. Thus substantial performance enhancement is achieved by implementing image processing algorithm on Cell Broadband Engine but with significant increase in programming efforts and development time.

6.2 Future Scope

The future scope of this work is the development of Image Processing Libraries that will be optimized for maximum performance enhancement on Cell Broadband Engine. This will help in rapid development of applications involving image processing using these libraries on Cell Broadband Engine. Thus the performance of applications involving image processing will be improved with minimum programming efforts on hybrid multicore architecture of Cell Broadband Engine.

Appendix A

Vector/SIMD Multimedia Extension data types

The Vector/SIMD Multimedia Extension model adds a set of fundamental data types, called vector types.

Vector types are shown in Table I. The represented values are in decimal (base-10) notation. The vector registers are 128 bits and can contain:

- Sixteen 8-bit values, signed or unsigned
- Eight 16-bit values, signed or unsigned
- Four 32-bit values, signed or unsigned
- Four single-precision IEEE-754 floating-point values

Vector Data Type	*Meaning*	*Values*
vector unsigned char	Sixteen 8-bit unsigned values	0 ... 255
vector signed char	Sixteen 8-bit signed values	-128 ... 127
vector bool char	Sixteen 8-bit unsigned boolean	0 (false), 255 (true)
vector unsigned short	Eight 16-bit unsigned values	0 ... 65535
vector unsigned short int	Eight 16-bit unsigned values	0 ... 65535
vector signed short	Eight 16-bit signed values	-32768 ... 32767
vector signed short int	Eight 16-bit signed values	-32768 ... 32767
vector bool short	Eight 16-bit unsigned boolean	0 (false), 65535 (true)
vector bool short int	Eight 16-bit unsigned boolean	0 (false), 65535 (true)
vector unsigned int	Four 32-bit unsigned values	0 ... 2^{32} - 1
vector signed int	Four 32-bit signed values	-2^{31} ... 2^{31} - 1
vector bool int	Four 32-bit unsigned values	0 (false), 2^{31} - 1 (true)
vector float	Four 32-bit single precision	IEEE-754 values
vector pixel	Eight 16-bit unsigned values	1/5/5/5 pixel

Table I: Vector/SIMD Multimedia Extension data types

Appendix B

Generic SPU Intrinsics

Generic intrinsics map to one or more assembly-language instructions, as a function of the type of its input parameters. Generic intrinsics are often implemented as compiler built-ins.

Many generic intrinsics accept scalars as one of their operands. These correspond to intrinsics that map to instructions with immediate values. Table I lists the generic SPU intrinsics.

Intrinsic	*Description*
d = spu_splats(a)	Replicate scalar a into all elements of vector d
d = spu_add(a, b)	Vector add
d = spu_genc(a, b)	Vector generate carry
d = spu_madd(a, b, c)	Vector multiply and add
d = spu_msub(a, b, c)	Vector multiply and subtract
d = spu_mul(a, b)	Vector multiply
d = spu_rsqrte(a)	Vector floating-point reciprocal square root estimate
d = spu_and(a, b)	Vector bit-wise AND
d = spu_nand(a, b)	Vector bit-wise complement of AND
d = spu_nor(a, b)	Vector bit-wise complement of OR
d = spu_xor(a, b)	Vector bit-wise exclusive OR
d = spu_rl(a, count)	Element-wise bit rotate left
d = spu_sl(a, count)	Element-wise bit shift left
d = spu_extract(a, element)	Extract vector element from vector
d = spu_insert(a, b, element)	Insert scalar into specified vector element
d = spu_promote(a, element)	Promote scalar to vector

Table I: Generic SPU Intrinsics

References

[1] F. Rinnerthaler, W. Kubinger, J. Langer, M. Humenberger, and S. Borbely, "Boosting the performance of embedded vision systems using a dsp/fpga co-processor system," *Systems, Man and Cybernetics, 2007*, vol. 7-10, pp. 1141 – 1146, October 2007.

[2] B. Rajan and S. Ravi, "Fpga based hardware implementation of image filter with dynamic reconfiguration architecture," *IJCSNS International Journal of Computer Science and Network Security*, vol. 6, December 2006.

[3] A. Kayi, Y. Yao, T. ElGhazawi, and G. Newby, "Experimental evaluation of emerging multi-core architectures," *IPDPS 2007*, vol. 26-30, pp. 1 – 6, March 2007. IEEE International.

[4] J. Janczyn, "Alias cloth technology demonstration for the cell processor," tech. rep., Alias Systems Corp., 2005.

[5] B. Minor, G. Fossum, and V. To, "Terrain rendering engine (tre): Cell broadband engine optimized real-time ray-caster," (New York), IBM Corporation Systems and Technology Group, May 2005.

[6] F. Petrini, G. Fossum, J. Fernandez, A. L. Varbanescu, M. Kistler, and M. Perrone, "Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine," IEEE, 2007.

[7] L. Liu, Q. Liu, A. Natsev, K. A. Ross, J. R. Smith, and A. L. Varbanescu, "Digital media indexing on the cell processor," *Multimedia and Expo, 2007*, pp. 1866–1869, July 2007. IEEE International Conference on.

[8] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos, "Dynamic multigrain parallelization on the cell broadband engine," *Conf. PPoPP 2007*, March 2007.

[9] A. C. Chow, G. C. Fossum, and D. A. Brokenshire, "A programming example: Large fft on the cell broadband engine," (New York), IBM Corporation Systems and Technology Group, May 2005.

[10] D. A. Bader and V. Agarwal, "Fftc: Fastest fourier transform for the ibm cell broadband engine," in *HiPC 2007*, pp. pp. 172–184, LNCS 4873, 2007.

[11] L. Chen, Z. Hu, J. Lin, and G. R. Gao, "Optimizing the fast fourier transform on a multi-core architecture," IEEE, 2007.

[12] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation, *Example Library API Reference*, version 3.0 ed., 2007.

[13] L. J. Siegel, H. J. Siegel, and A. E. Feather, "Parallel processing approaches to image correlation," vol. Vol. C-31 of *No. 3*, IEEE Transactions on Computers, March 1982.

[14] International Business Machines Corporation, New York, *Software Development Kit for Multicore Acceleration Programmer's Guide*, version 3.0 ed., 2007.

[15] A. Arevalo, R. M. Matinata, M. Pandian, E. Peri, K. Ruby, F. Thomas, and C. Almond, *Programming the Cell Broadband Engine Examples and Best Practices*. International Business Machines Corporation, first edition ed., 2007.

[16] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation, New York, *Cell Broadband Engine Programming Handbook*, version 1.1 ed., April 2007.

[17] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation, New York, *Cell Broadband Engine Architecture*, version 1.0 ed., October 2006.

[18] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation, New York, *Software Development Kit for Multicore Acceleration Programming Tutorial*, version 3.0 ed., 2007.

[19] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. Prentice Hall, second edition ed., 2002.

[20] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation, *C/C++ Language Extensions for Cell Broadband Engine Architecture*, version 2.5 ed., February 2008.

[21] B. G. G. Krishna, "Conjugate point identification and image matching methods," tutorial on satellite photogrammetry and surveying, INCA GB and SAC (ISRO), Ahmedabad, December 2000.

[22] International Business Machines Corporation, *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*, version 2.07c ed., 2006.