Design, Implementation and Performance Analysis of Image Processing Algorithms for Cell Multi-core Processor

By

HIMANSHU RAWAT

(06MCE013)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INSTITUTE OF TECHNOLOGY
NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY
AHMEDABAD 382 481
MAY 2008

Design, Implementation and Performance Analysis of Image Processing Algorithms for Cell Multi-core Processor

By

HIMANSHU RAWAT

(06MCE013)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INSTITUTE OF TECHNOLOGY
NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY
AHMEDABAD 382 481
MAY 2008

Major Project

On

Design, Implementation and Performance Analysis of Image Processing Algorithms for Cell Multi-core Processor

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science & Engineering

By

Himanshu Rawat
(06MCE013)

Under Guidance of

Dr. S.N. Pradhan

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INSTITUTE OF TECHNOLOGY
NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY
Ahmedabad 382481
May 2008

This is to certify that Dissertation entitled

Design, Implementation and Performance Analysis of Image Processing Algorithms for Cell Multi-core Processor

Submitted by

Himanshu Rawat

has been accepted towards partial fulfillment of the requirement

For the degree of

Master of Technology in Computer Science & Engineering

Dr. S. N. Pradhan	Prof. D. J. Patel
P.G.Coordinator	Head of the Department

Prof. A. B. Patel

Director, Institute of Technology

CERTIFICATE

This is to certify that the Major Project entitled **"Design, Implementation and Performance Analysis of Image Processing Algorithms for Cell Multi-core processor"** submitted by **Mr. Himanshu Rawat (06MCE013)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science & Engineering, 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 Master degree.

Dr. S.N.Pradhan
Project Guide,
P. G. Coordinator,
Department of Computer Science & Engineering,
Institute of Technology,
Nirma University,
Ahmedabad.

Date:-

ACKNOWLEDGEMENT

It gives me immense pleasure in expressing thanks and profound gratitude to **Dr. S.N Pradhan**, P.G. Coordinator, Computer Science & Engineering Department, Nirma University, Ahmedabad for his valuable guidance and continual encouragement throughout my Major project. I am heartily thankful to him for his precious time, suggestions and sorting out the difficulties of my topic that helped me a lot during this study.

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

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

Himanshu Rawat
Roll No. 06MCE013

ABSTRACT

Since the inception of the desktop computer, performance of software has improved at an exponential rate, primarily driven by the steady technological improvements of microprocessors. Due to fundamental physical limitations, power constraints and over heating, we are now witnessing a radical change in commodity microprocessor architecture, to multi-core designs. Multi-core architecture offers higher performance at same power consumption and manufacturing cost. As the name suggests, it combines two or more independent processors onto a single integrated circuit and is focused on improving parallelism on a thread level to improve performance (Thread Level Parallelism - TLP)

The Cell Broadband Engine Architecture (CBEA) is a Heterogeneous multi-core architecture, including a 64-bit PowerPC Processor Element and eight Synergistic Processor Elements which offers a raw computing power of up to 200 GFlops per 3.2 GHz chip. The Cell bears a huge potential for compute intensive applications.

Image processing applications such as filtering, edge detection, correlation etc. require large computation power. Processing large images on a system with single processor consumes a lot of time. In this thesis work some Image processing algorithms, which consumes a lot of processing power, are implemented on CELL processor so as to take full advantage of the high processing power provided by its novice architecture. However it requires addressing many challenges for implementing such applications for CBEA.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

BEI	Broadband Engine Interface
CBE	Cell Broadband Engine
CMP	Chip Multiprocessors
DCT	Discrete Cosine Transform
DIP	Digital Image Processing
DMA	Direct Memory Access
EIB	Elementary Interconnect Bus
FXU	Fixed-Point Unit
IU	Instruction Unit
LS	Local Store
LSU	Load and Store Unit
MFC	Memory Flow Controller
MIC	Memory Interface Controller
MMU	Memory Management Unit
PPE	PowerPC Processor Element
PPSS	PowerPC Processor Storage Subsystem
PPU	PowerPC Processor Unit
PS3	Playstation 3
RISC	Reduced Instruction Set Computer
SCN	SPU Control Unit
SDK	Software Development Kit
SFP	SPU Floating-Point Unit
SFS	SPU Odd Fixed-Point Unit
SFX	SPU Even Fixed-Point Unit
SIMD	Single Instruction Multiple Data
SLS	SPU Load and Store Unit
SPE	Synergistic Processor Element
SPU	Synergistic Processor Unit
SSC	SPU Channel and DMA Unit
VSU	Vector/Scalar Unit
YDL	Yellow Dog Linux

1 INTRODUCTION

1.1 General

Image processing applications such as filtering, edge detection, correlation etc. require large computation power. Processing large images on a system with single processors consumes a lot of time. Using Digital Signal Processors (DSPs) we can improve the performance of such applications. However performance gain is limited. We can use multiprocessor or multicore system to improve performance of such applications.

Cell Broadband Engine (Cell BE) which is a multicore system can be used for this purpose. Cell BE is the result of collaboration between Sony, Toshiba, and IBM. Although the Cell BE was initially intended for application in game consoles and media-rich consumer-electronics devices such as high-definition televisions, the architecture can be used for compute intensive applications such as satellite image processing, medical imaging, digital media, communications, and some scientific applications to improve performance. The Cell BE performs significantly faster than existing CPUs for many applications. IBM provides the Cell SDK for programming and simulating the cell architecture.

As exciting as it may sound, using the Playstation-3 (PS3) for scientific computing is a bumpy ride. Parallel programming models for multi-core processors are in their infancy, and standardized APIs are not even on the horizon. As a result, presently, only hand-written code fully exploits the hardware capabilities of the Cell processor. Ultimately, the suitability of the PS3 platform for scientific computing is most heavily impaired by the devastating disproportion between the processing power of the processor and the crippling slowness of the interconnect. Nevertheless, the Cell processor is a revolutionary chip, delivering ground-breaking performance and now available in an affordable package. Programming such a multicore system is a challenging task. To make full use of the architecture potential the algorithm design must be done carefully. All the cores of Cell BE are Single Instruction Multiple Data

(SIMD) based architecture cores. Hence Cell BE provides two levels of parallelism. This needs to be taken care of while designing the algorithm.

This report shows the details of the Cell architecture and design of Image Processing algorithms for Cell BE.

1.2 Motivation

Enhancement of CPU processing power and speed has been a priority issue in the past several years due to the widespread use of multimedia applications, medical imaging, scientific applications as well as the advent of 3D games and other advanced video applications that demand higher-speed processing of massive audio/video data to offer higher-quality entertainment.

Conventional approaches to satisfy this demand counted on the improvement of the performance of the core singularly used in each CPU. However, they soon proved to have limitations – the CPU design became much more complex, causing problems as increases in power consumption, over-heating and manufacturing cost. That's why CPU manufacturers set about developing multi-core configurations that allow multiple cores to run in parallel on a single chip to realize further performance upgrades.

The Cell employs the heterogeneous multi-core processor configuration. Instead of conventional multi-application cores, the Cell uses two types of cores optimized for different applications: a control-intensive processor core that excels in handling frequent thread switching and thus is suited for use with the operating system, and a simple compute-intensive processor core that addresses multimedia processing. With this configuration, each core can maintain its processing performance. The core structure can also be simplified drastically.

Initially the Cell project was intended for Playstation3 so the main concentration of the developers of Cell processor was on the better performance on gaming consoles. But later it was realized that the immense computing power of Cell Broadband engine can be utilized for other compute intensive scientific fields. Though the programmability of the this processor architecture is a bit complex but it is worth taking pains to utilize its computing power. A lot of work is being done in this direction and a lot of success has been achieved using this processor architecture for diverse application areas.

Image processing applications such as filtering, edge detection, correlation etc. require large computation power. Processing large images on a system with single processors consumes a lot of time. In this thesis work some Image processing algorithms, which consumes a lot of processing power, are implemented on Cell processor so as to take full advantage of the high processing power provided by its novice architecture. The reason for selecting image processing as the area of application is that image processing algorithms provide a lot of scope for parallelism, which is the hallmark of Cell processor architecture. Cell architecture provides two levels of parallelism, one at the inter-SPU level which is provided by the 8 SPU's and other at the intra-SPU level which is provided by the vector support. Image processing algorithms are very well suited to exploit these two levels of parallelism provided by Cell BE. Because the image can be easily broken into smaller images and also in image processing algorithms we generally have to perform the same operation over each and every pixel which gives us a scope for utilizing vector operations.

1.3 Scope of the Work

The main objective of this thesis is to modify the existing scalar image processing algorithms to utilize the Cell processor architecture and to claim the performance improvement over the traditional scalar versions of it. But since the architecture of Cell processor is quite new and different, the existing Image processing algorithms are designed for singlecore processors and would not be able to exploit the computing power provided by Cell BE. So these algorithms have to be redesigned and restructured in such a way so that all nine cores can be fully utilized. Also the SIMD feature provided by all these cores has to be taken care of while designing the algorithms. This makes the task of programming for Cell processor quite complex and the onus is on the programmer to restructure the algorithms carefully. The scope of this thesis work encompasses the detailed architectural study and programming methodology of the Cell architecture. Based on this knowledge two image processing algorithms are selected: Sum of Squared Difference (SSD) image matching algorithm and HAAR wavelet transform Algorithm. These algorithms are then worked upon and redesigned to execute on Cell BE so as to give a performance Speedup over the scalar versions of these algorithms executed on conventional single core processors.

1.4 Outline of Thesis

This thesis is organized as follows:

- Chapter 2 presents the details of the Cell architecture like detail architecture of the processor cores used in it, details of memory interface controller and bus connecting all the elements. It also presents some advantages of Cell architecture and gives specifications of some systems based on Cell architecture.

- Chapter 3 presents the details of SDK used for programming Cell and lists SDK installation steps. It also gives the details about the IBM's full system simulator, its usage and performance analysis support provided by the simulator.

- Chapter 4 describes the programming methods for Cell. It also describes some points to consider while partitioning the application and data between various cores and how the program control and data flows between processor cores. In the end it gives some advanced techniques for programming the cell.

- Chapter 5 explains a basic image matching algorithm: Sum of Squared Difference (SSD) and proposed algorithm which is optimized for Cell Broadband Engine. It also shows implementation details and performance results.

- Chapter 6 explains the Wavelet transform of images and Haar wavelet transform algorithm and the proposed algorithm optimized for Cell Broadband Engine. It also shows implementation details and performance results.
- Chapter 7 presents the Summary, Conclusion and Future work.

2 THE CELL ARCHITECTURE

2.1 History of the Cell Project

Cell represents a revolutionary extension of conventional microprocessor architecture and organization. This report discusses the history of the project, the program objectives and challenges, the design concept, the architecture and programming models, and the implementation.

Initial discussion on the collaborative effort to develop Cell began with support from CEOs from the Sony and IBM companies: Sony as a content provider and IBM as a leading-edge technology and server company. Collaboration was initiated among SCEI (Sony Computer Entertainment Incorporated), IBM, for microprocessor development, and Toshiba, as a development and high-volume manufacturing technology partner. This led to high-level architectural discussions among the three companies during the summer of 2000. During a critical meeting in Tokyo, it was determined that traditional architectural organizations would not deliver the computational power that SCEI sought for their future interactive needs. SCEI brought to the discussions a vision to achieve 1,000 times the performance of PlayStation2. At this stage of the interaction, the IBM Research Division became involved for the purpose of exploring new organizational approaches to the design IBM process technology was also involved, contributing state-of-the-art 90-nm process with silicon-on insulator (SOI), low-k dielectrics, and copper interconnects. During this interaction, a wide variety of multi core proposals were discussed, ranging from conventional chip multiprocessors (CMPs) to dataflow oriented multiprocessors. By the end of 2000 an architectural concept had been agreed on, that combined the 64-bit Power Architecture with memory flow control and ‘‘synergistic’’ processors in order to provide the required computational density and power efficiency. After several months of architectural discussion and contract negotiations, the STI (SCEI–Toshiba–IBM) Design Center was formally opened in Austin, Texas, on March 9, 2001. The STI Design Center represented a joint investment in design of about $400,000,000.Separate joint collaborations were also set in place for process technology development. A number of key elements were employed to drive the success of the Cell multiprocessor design. First, a holistic design approach was used, encompassing processor architecture, hardware implementation, system structures, and software programming models. Second, the design center staffed key leadership positions from various IBM sites. Third, the design incorporated many flexible elements ranging from reprogrammable synergistic processors to reconfigurable I/O interfaces in order to support many systems configurations with one high-volume chip. Although the STI design center for this ambitious, large-scale project was based in Austin (with IBM, the Sony Group, and Toshiba as partners), many other IBM sites were also involved and were critical to the project.

2.2 Architectural Overview

The Cell Broadband Engine is a single-chip multiprocessor with nine processors operating on a shared, coherent memory. There are two types of processors in it: the PowerPC Processor Element (PPE), and the Synergistic Processor Element (SPE). The Cell Broadband Engine has one PPE and eight SPEs [1], [2], [4]. The first type of processor element, the PPE, is a 64-bit PowerPC Architecture core. It is fully compliant with the 64-bit PowerPC Architecture and can run 32-bit and 64-bit operating systems and applications. The second type of processor element, the SPE, is optimized for running compute-intensive applications, and it is not optimized for running an operating system. The SPEs are independent processors, each running its own individual application programs. Each SPE has full access to coherent shared memory, including the memory-mapped I/O space. The SPEs are designed to be programmed in high-level languages and support a rich instruction set that includes extensive single-instruction, multiple-data (SIMD) functionality.

Figure 2-1 shows a block diagram of the Cell Broadband Engine.

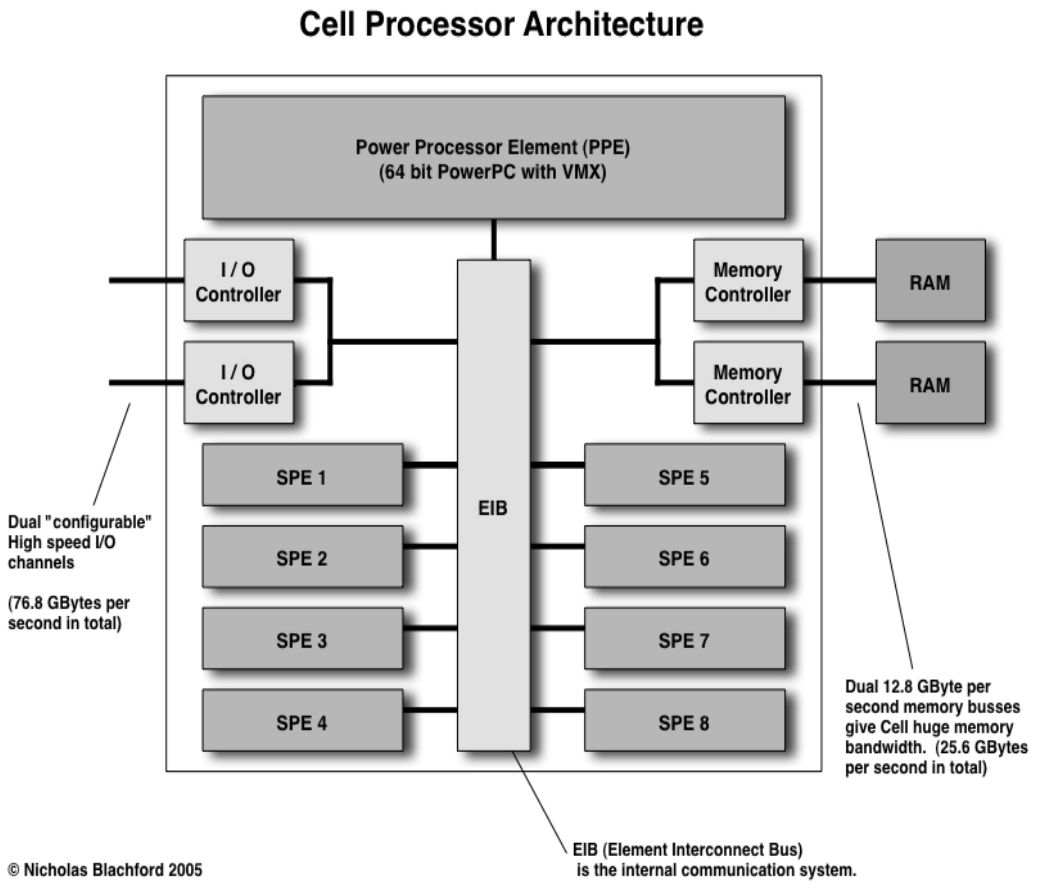

Cell Processor Architecture
Power Processor Element (PPE)
(64 bit PowerPC with VMX)
I / O Controller
I / O Controller
Memory Controller
Memory Controller
RAM
RAM
EIB
SPE 1
SPE 2
SPE 3
SPE 4
SPE 5
SPE 6
SPE 7
SPE 8
Dual "configurable" High speed I/O channels
(76.8 GBytes per second in total)
Dual 12.8 GByte per second memory busses give Cell huge memory bandwidth. (25.6 GBytes per second in total)
EIB (Element Interconnect Bus) is the internal communication system.
© Nicholas Blachford 2005

Figure 2-1 Cell Broadband Engine Overview

The PPE is more adept at control-intensive tasks and quicker at task switching. The SPEs are more adept at compute-intensive tasks and slower at task switching. A significant difference between the SPE and the PPE 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). However, 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.

2.3 PowerPC Processor Element (PPE)

The PPE is the main processor. It contains a 64-bit PowerPC Architecture reduced instruction set computer (RISC) core. It runs an operating system, manages system resources, and is intended primarily for control processing, including the allocation and management of SPE threads. It supports both the PowerPC instruction set and the Vector/SIMD Multimedia Extension instruction set. The PPE contains two main units: the PowerPC Processor Unit (PPU) and the PowerPC Processor Storage Subsystem (PPSS), shown in figure 2-2.

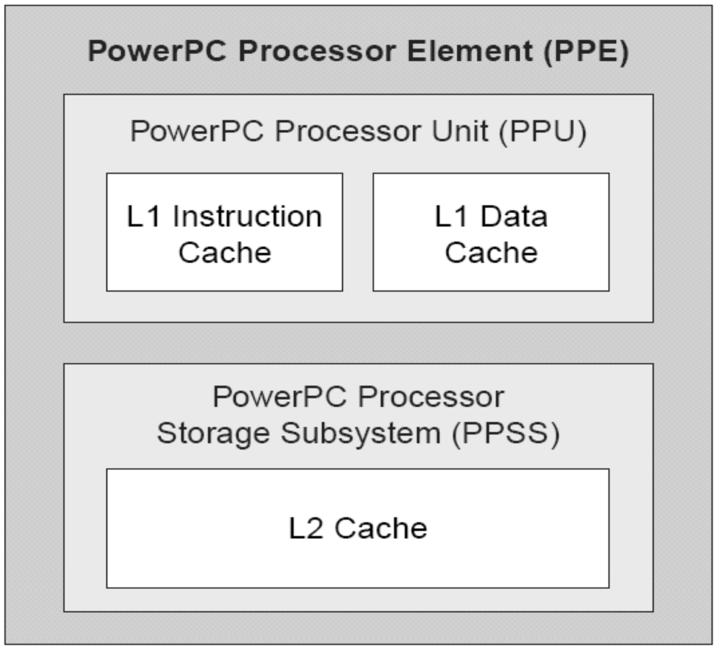

PowerPC Processor Element (PPE)
PowerPC Processor Unit (PPU)
L1 Instruction Cache
L1 Data Cache
PowerPC Processor Storage Subsystem (PPSS)
L2 Cache

Figure 2-2 PPE Block Diagram

2.3.1 PowerPC Processor Unit (PPU)

The PPU is the processing unit in the PPE. It contains six execution units. It also contains a primary cache comprised of a 32KB instruction cache and a 32KB data cache. The PPU executes the PowerPC Architecture instruction set and the Vector/SIMD Multimedia Extension instructions. It has duplicate sets of the PowerPC and vector user-state register files (one set for each thread) plus one set of the following functional units:

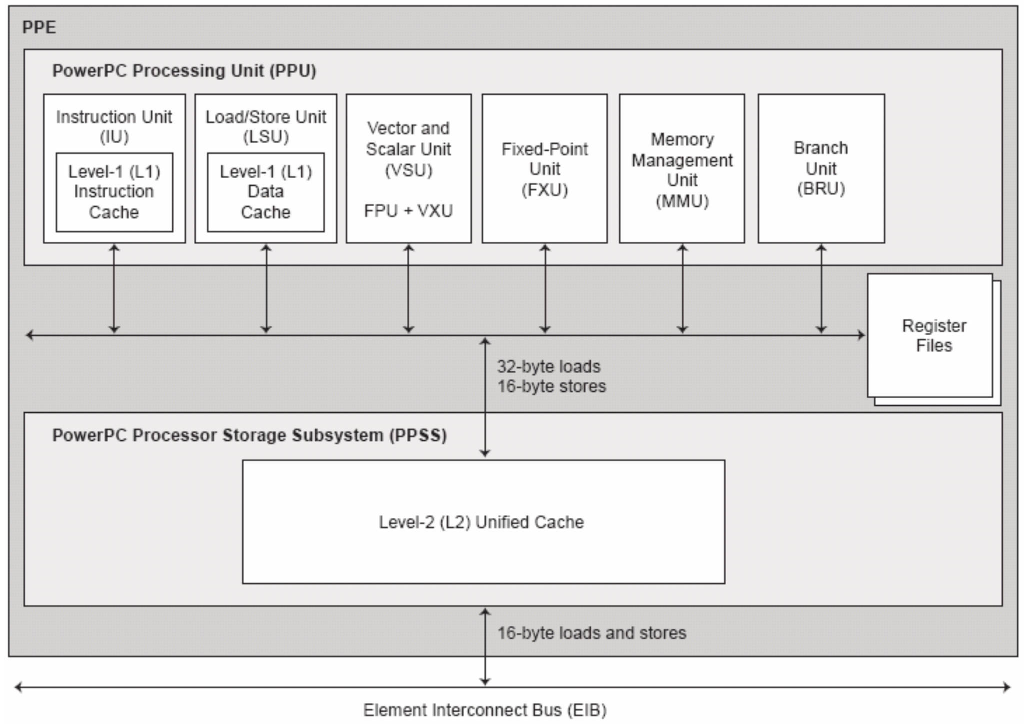

PPE
PowerPC Processing Unit (PPU)
Instruction Unit (IU)
Level-1 (L1) Instruction Cache
Load/Store Unit (LSU)
Level-1 (L1) Data Cache
Vector and Scalar Unit (VSU)
FPU + VXU
Fixed-Point Unit (FXU)
Memory Management Unit (MMU)
Branch Unit (BRU)
Register Files
32-byte loads
16-byte stores
PowerPC Processor Storage Subsystem (PPSS)
Level-2 (L2) Unified Cache
16-byte loads and stores
Element Interconnect Bus (EIB)

Figure 2-3. PPE Functional Units

- **Instruction Unit (IU):** The IU performs the instruction-fetch, decode, dispatch, issue, branch, and completion portions of execution. It contains the L1 instruction cache, which is 32 KB, 2-way set-associative, parity protected. The cache-line size is 128 bytes.

- **Load and Store Unit (LSU):** The LSU performs all data accesses, including execution of load and store instructions. It contains the L1 data cache, which is 32 KB, 4-way set-associative, write-through, and parity protected. The cache-line size is 128 bytes.

- **Vector/Scalar Unit (VSU):** The VSU includes a Floating-Point Unit

(FPU) and a 128-bit Vector/SIMD Multimedia Extension Unit (VXU), which together execute floating-point and Vector/SIMD Multimedia Extension instructions.

- **Fixed-Point Unit (FXU):** The FXU executes fixed-point (integer) operations, including add, multiply, divide, compare, shift, rotate, and logical instructions.

- **Memory Management Unit (MMU):** The MMU manages address translation for all memory accesses. It has a 64-entry Segment Lookaside Buffer (SLB) and 1024-entry, unified, parity protected Tanslation Lookaside Buffer (TLB).

2.3.2 PowerPC Processor Storage Subsystem (PPSS)

The PPSS handles all memory accesses by the PPU. The PPSS has a unified, 512-KB, 8-way set-associative, write-back L2 cache. Like the L1 caches, the cache-line size for the L2 is 128 bytes. The cache has a single-port read/write interface to main storage that supports eight software-managed data-prefetch streams. It includes the contents of the L1 data cache but is not guaranteed to contain the contents of the L1 instruction cache, and it provides fully coherent symmetric multiprocessor (SMP) support.

The PPSS performs data-prefetch for the PPU and bus arbitration. Traffic between the PPU and PPSS is supported by a 32-byte load port, and a 16-byte store port. The interface between the PPSS and EIB supports 16-byte load and 16-byte store buses.

2.3.3 PPE Registers

The PPE problem-state (user) registers are shown in Figure 2-4. 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.

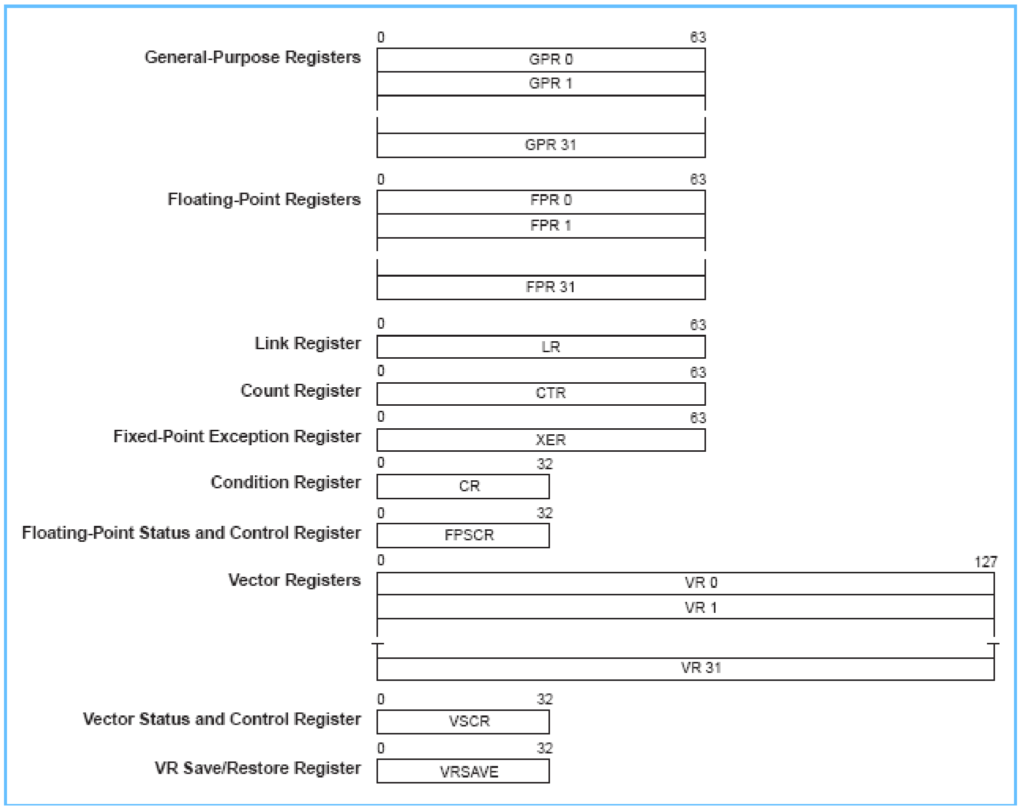


Figure 2-4. PPE User Register Set

2.3.4 Vector/SIMD Multimedia Extension Instructions

PPE supports PowerPC instructions as well as Vector/SIMD Multimedia Extension instructions. Vector/SIMD Multimedia Extension instructions can be freely mixed with PowerPC instructions in a single program. The 128-bit Vector/SIMD Multimedia Extension unit (VXU) operates concurrently with the PPU's 32-bit fixed-point unit (FXU) and 64-bit floating-point unit (FPU). Like PowerPC instructions, the Vector/SIMD Multimedia Extension instructions are four bytes long and word-aligned. The Vector/SIMD Multimedia Extension instructions support simultaneous execution on multiple elements that make up the 128-bit vector operands. Vector/SIMD Multimedia Extension instructions do not generate exceptions (other than data storage interrupt exceptions on loads and stores), do not support unaligned memory accesses or complex functions, and share few resources or communication paths with the other PPE execution units.

A vector is an instruction operand containing a set of data elements packed into a one-dimensional array. The elements can be fixed-point or floating-point values. Most Vector/SIMD Multimedia Extension and SPU instructions operate on vector operands. Vectors are also called Single-Instruction, Multiple-Data (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.

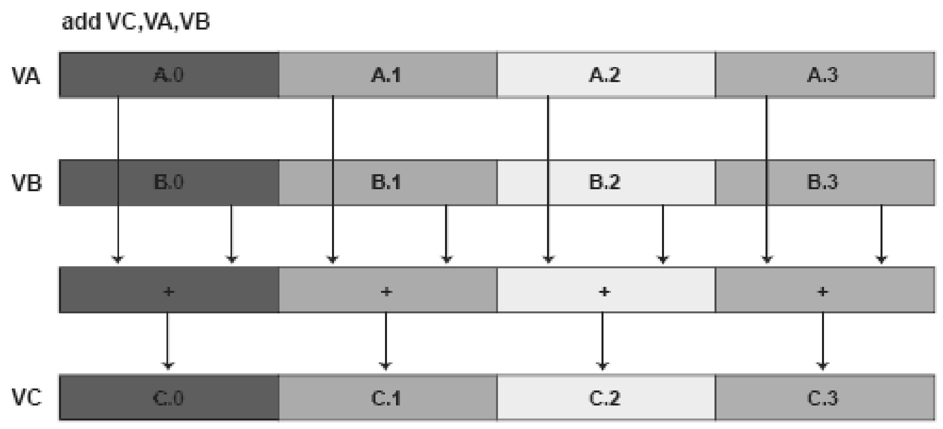


Figure 2-5. SIMD Add Operations

Support for SIMD operations is pervasive in the CBE processor. In the PPE, they are supported by the Vector/SIMD Multimedia Extension instructions. 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 register, and, for example, added to four other words in a different register in a single operation. Figure 2-5 shows such an operation. Similar operations can be performed on vector operands containing 16 bytes, 8 halfwords, or 2 doublewords.

2.4 Synergistic Processor Elements (SPEs)

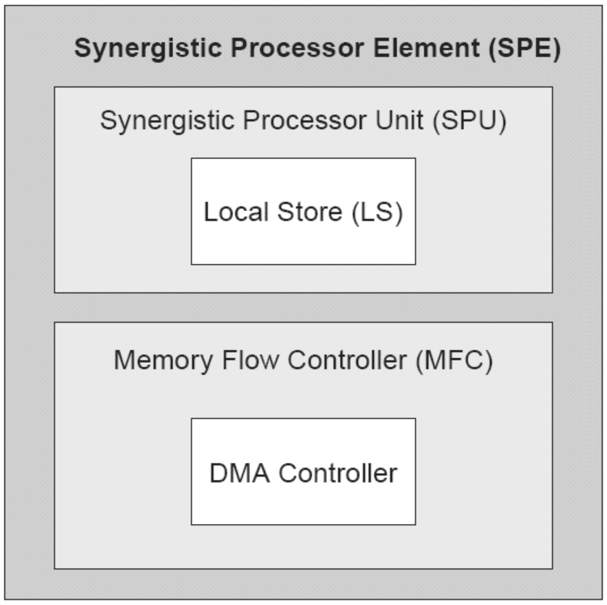

Synergistic Processor Element (SPE)
Synergistic Processor Unit (SPU)
Local Store (LS)
Memory Flow Controller (MFC)
DMA Controller

Figure 2-6. SPE Block Diagram

The eight SPEs are SIMD processors optimized for data-rich operations allocated to them by the PPE. Each of these identical elements contains a RISC core, 256-KB, software-controlled local store for instructions and data, and a large (128-bit, 128-entry) unified register file. The SPEs support a special SIMD instruction set, and they rely on asynchronous DMA transfers to move data and instructions between main storage and their local stores. The SPEs are not intended to run an operating system. The SPE contains two main units: Synergistic Processor unit (SPU) and Memory Flow Control (MFC).

2.4.1 Synergistic Processor Unit (SPU)

Each SPE incorporates its own SPU to perform its allocated computational task. The SPUs use a unique instruction set designed specifically for their operations. It contains six execution units and a 256 KB local store. The SPU fetches instructions from its unified (instructions and data) 256-KB local store (LS), and it loads and stores data between its LS and its single register file for all data types, which has 128 registers, each 128 bits wide. The SPU has a DMA interface and a channel interface for communicating with its MFC, the PPE and other devices (including other SPEs). Each SPU is an independent processor element with its own program counter, optimized to run SPU programs. The SPU fills its LS by requesting DMA transfers from its MFC, which implements the DMA transfers using its DMA controller. Then, the SPU fetches and executes instructions from its LS, and it loads and stores data to and from its LS. The main SPU functional units are shown in Figure 2-7. These include the Synergistic Execution Unit (SXU), the LS, and the SPU Register File Unit (SRF).

The SPU can issue and complete up to two instructions per cycle, one on each of the two (odd and even) execution pipelines. Whether an instruction goes to the odd or even pipeline depends on the instruction type. The instruction type is also related to the execution unit that performs the function.

- **SPU Odd Fixed-Point Unit (SFS):** Executes quadword shift and rotates mask operations on bits, bytes, halfwords, and words, and shuffle operations on bytes.

- **SPU Even Fixed-Point Unit (SFX):** Executes arithmetic instructions, logical instructions, word SIMD shifts and rotates, floating-point compares, and floating-point reciprocal and reciprocal square-root estimates.

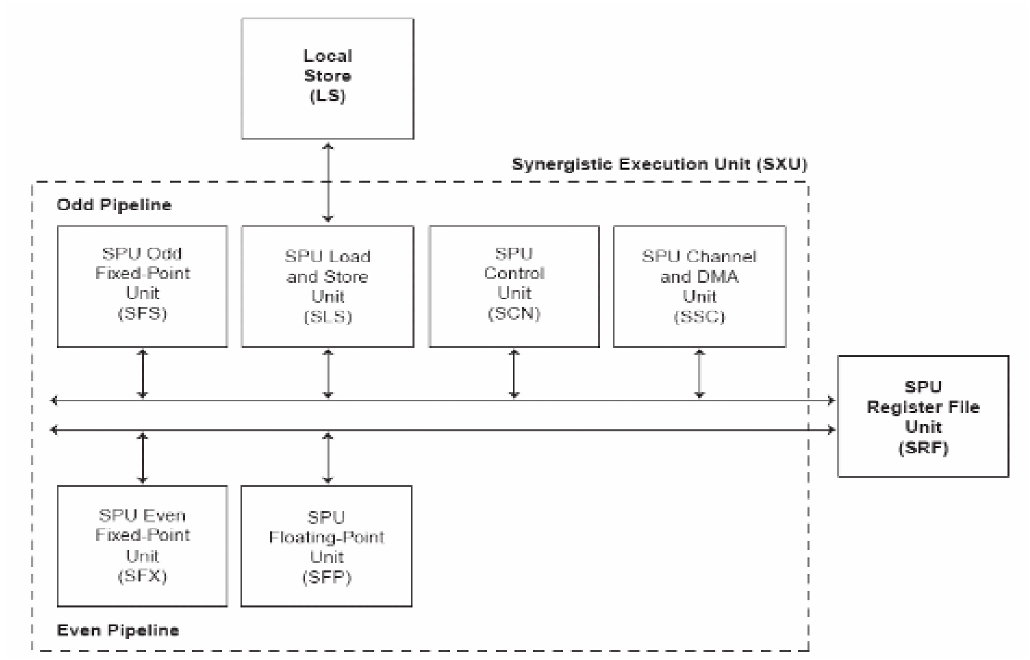

Local Store (LS)
Synergistic Execution Unit (SXU)
Odd Pipeline
SPU Odd Fixed-Point Unit (SFS)
SPU Load and Store Unit (SLS)
SPU Control Unit (SCN)
SPU Channel and DMA Unit (SSC)
SPU Register File Unit (SRF)
SPU Even Fixed-Point Unit (SFX)
SPU Floating-Point Unit (SFP)
Even Pipeline

Figure 2-7. SPU Functional Units

- **SPU Floating-Point Unit (SFP):** Executes single- and double-precision floating-point instructions, integer multiplies and conversions, and byte operations. The SPU supports only 16-bit multiplies, so 32-bit multiplies are implemented in software using 16-bit multiplies.

- **SPU Load and Store Unit (SLS):** Executes load and store instructions and load branch-target- buffer (BTB) instructions. It also handles DMA requests to the LS.

- **SPU Control Unit (SCN):** Fetches and issues instructions to the two pipelines, executes branch instructions, arbitrates access to the LS and register file, and performs other control functions.

- **SPU Channel and DMA Unit (SSC):** Enables communication, data transfer, and control into and out of the SPU.

2.4.2 Local Store (LS)

The Local Store (LS) is a 256-KB, single-ported, noncaching memory. It stores all instructions and data used by the SPU. It supports one access per cycle from either SPE software or DMA transfers. SPU instruction prefetches are 128 bytes per cycle. SPU data-access bandwidth is 16 bytes per cycle, quadword aligned. DMA-access bandwidth is 128 bytes per cycle. It is the only memory that can be referenced directly from the SPU. It's controlled by software running on SPU.

The SPU accesses its LS with load and store instructions, and it performs no address translation for such accesses. Privileged software on the PPE can assign effective-address aliases to LS. This enables the PPE and other SPEs to access the LS in the main-storage domain. The PPE performs such accesses with load and store instructions, without the need for DMA transfers. However other SPEs must use DMA transfers to access the LS in the main-storage domain. Figure 2-8 illustrates the methods by which an SPU, the PPE, other SPEs, and I/O devices access the SPU's associated LS, when the LS has been aliased to the main storage domain.

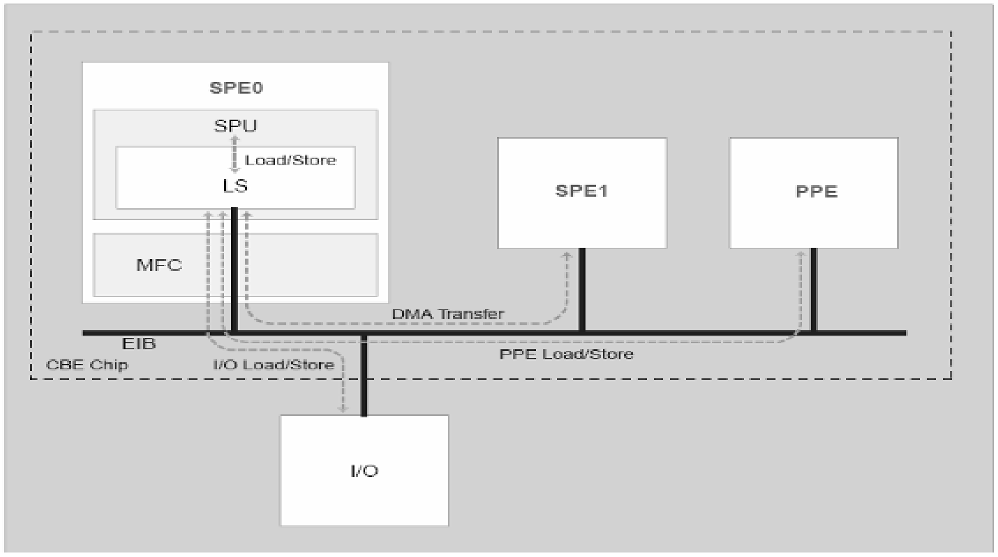


Figure 2-8. LS Access Methods

2.4.3 Memory Flow Controller (MFC)

Each SPU has its own Memory Flow Controller (MFC). The MFC serves as the SPU's interface, by means of the Element Interconnect Bus (EIB), to main-storage and other processor elements and system devices. The MFC's primary role is to interface its LS with the main-storage domain. It does this by means of a DMA controller that moves instructions and data between its LS and main storage. The MFC also supports synchronization between main storage and the LS, and communication functions (such as mailbox and signal-notification messaging) with the PPE and other SPEs and devices.

2.4.4 Register File

The SPU's 128-entry, 128-bit register file stores all data types—integer, single- and double-precision floating-point, scalars, vectors, logical, bytes, and others [1]. It also stores return addresses, results of comparisons, and so forth. All computational instructions operate on registers—there are no computational instructions that modify storage. Figure 2-9 shows SPE register file.

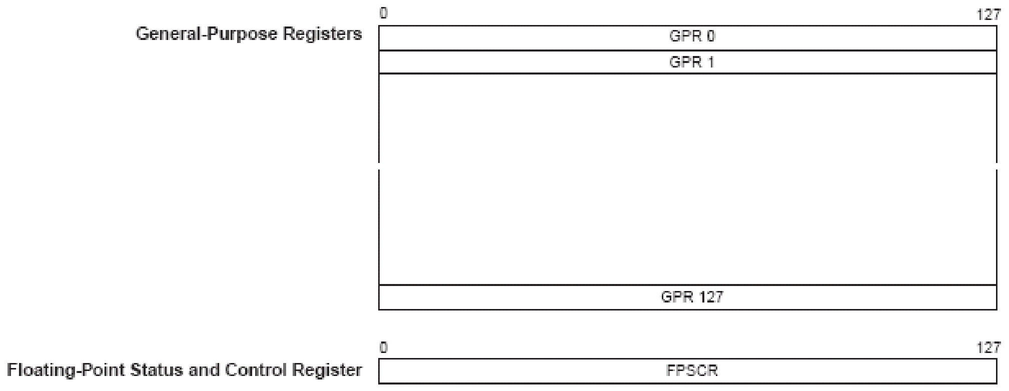

0
127
General-Purpose Registers
GPR 0
GPR 1
GPR 127
0
127
Floating-Point Status and Control Register
FPSCR

Figure 2-9. SPE Register Set

2.5 Element Interconnect Bus (EIB)

The Element Interconnect Bus (EIB) is the communication path for commands and data between all processor elements on the CBE processor and the on-chip controllers for memory and I/O. The EIB supports full memory-coherent and symmetric multiprocessor (SMP) operations [3].

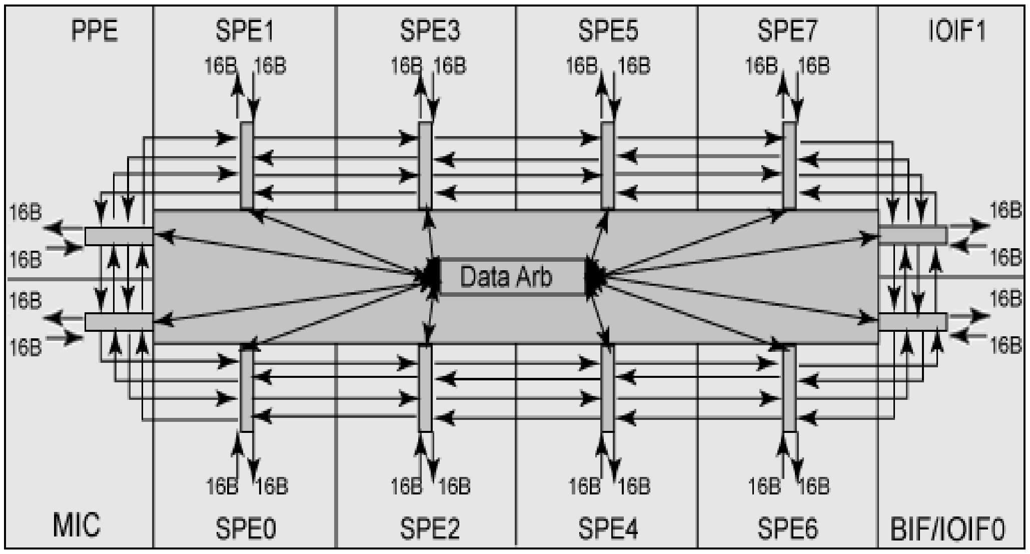


Figure 2-10. Element Interconnect Bus

EIB works on half the processor clock rate. The EIB is a 4-ring structure (two clockwise and two counterclockwise). Each ring supports 3 transfers simultaneously. The EIB's internal bandwidth is 96 bytes per cycle (4 rings * 3 simultaneous transfers/ring * 16 bytes/ring /2). A Resource Allocation Management (RAM) facility resides in the EIB and privileged software can use it to regulate the rate at which resource requestors (the PPE, SPEs, and I/O devices) can use memory and I/O resources.

2.6 Memory Interface Controller (MIC)

The on-chip Memory Interface Controller (MIC) provides the interface between the EIB and physical memory. It supports one or two Rambus Extreme Data Rate (XDR) memory interfaces. Memory accesses on each interface are 1 to 8, 16, 32, 64, or 128 bytes. Up to 64 reads and 64 writes can be queued.

2.7 Cell Broadband Engine Interface (BEI)

The 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.

The on-chip Cell Broadband Engine Interface (BEI) Unit supports I/O interfacing. It includes a Broadband Interface Controller (BIC), I/O Controller (IOC), and Internal Interrupt Controller (IIC). It manages data transfers between the EIB and I/O devices and provides I/O address translation and command processing. The BEI supports two Rambus FlexIO interfaces.

2.8 CELL Architecture Advantages

- **Faster Processing of Compute-Intensive Algorithms:** With the combination of single-precision 32-bit floating-point and 16-bit integer (or fixed-point) processing, the Cell BE processor handles compute-intensive algorithms and is particularly impressive for those requiring many floating-point calculations.

- **Higher Bandwidth Interconnect:** Each SPE contains a 256 KB high-speed local store and a DMA (direct memory access) engine for moving data and code to and from XDR memory and even to other SPEs – all via the EIB interconnect. The DMA engine can simultaneously read and write at the rate of 24 GB/s to and from the EIB and can handle numerous outstanding DMA requests.

- **Backward Compatibility with PowerPC Applications:** Because the Cell architecture is compatible with PowerPC processors, existing PowerPC applications can run on the Cell BE processor without modification. This flexibility provides a convenient entry point for programmers with symmetric multiprocessor (SMP) experience and eases the porting of existing software, including the operating system, to the Cell architecture.

- **Low Voltage/Low Power with High Performance/High Frequency:** The small number of gates per cycle enables the Cell BE processor to operate at low voltage and low power while maintaining high performance and high frequency. By using the SIMD architecture for both the vector media extensions (VMX) on the PPE and the instruction set of the SPEs, both performance and power efficiency are improved.

- **Efficient Communication and Ease of Programming:** The Cell BE processor's high-bandwidth memory and on-chip, coherent, high-bandwidth EIB deliver higher performance on memory bandwidth-intensive applications by enabling high-bandwidth internal interactions among the SPEs and the PPE. This coherency allows the SPEs and the PPE to share a single address space for efficient communication and ease of programming.

2.9 Systems based on CELL Architecture

There are two systems available based on CELL architecture. One is Sony PLAYSTATION 3 ant other is IBM BladeCenter QS20. Their details are given below.

2.9.1 IBM BladeCenter QS20

It is the first Cell Broadband Engine based system [6]. It is a high performance blade especially suitable for some compute intensive, single-precision, floating-point workloads. It helps to accelerate these targeted workloads to many times the speed of a traditional microprocessor, including image processing, signal processing, and graphics rendering applications.
Its specifications are as follows:

- **Processor:** Two 3.2 GHz Cell BE Processors
- **L2 Cache:** 512KB per Cell BE Processor, plus 256KB of local store memory for each SPE
- **Memory:** 1GB (512MB per processor)
- **Disk Storage:** 40GB IDE HDD
- **Networking:** Dual Gigabit Ethernet
- **Optional connectivity:** 1 or 2 InfiniBand 4x adapters connected via PCI-Express
- **Operating System:** Fedora core 5 Linux

2.9.2 Sony PLAYSTATION 3

PLAYSTATION 3, also called PS3, is another system based on the CELL architecture. It’s basically a very high resolution gaming console. It provides direct support for installing and booting foreign operating systems. Of course, many of the game-related features such as video acceleration are locked out for the third-party operating systems, but this series focuses on more general-purpose and scientific applications anyway. Note the PS3 has one of the SPE disabled, and one SPE reserved for system use, leaving seven processing units at your disposal [7].

Terra Soft Solutions has developed Yellow Dog Linux 5 in cooperation with Sony specifically for the PS3. Yellow Dog Linux (also known as YDL) has been an exclusively PowerPC-based distribution since its inception, so it was not surprising that Sony contracted it to develop the next version of YDL specifically for the PS3. YDL 5.0 includes libspe [Appendix A] support so that one can utilize power of all the SPEs available within PS3. We can get free version of YDL (YDL 5.0) from Terra Soft Solutions website which can be installed on PS3. See below for instructions on installing the YDL 5 onto the PS3.

Preparing PS3 for Installation

- Start your PS3 and perform initial settings if you are using it first time.
- Go to Settings, then System Settings, and choose Format Utility.
- Select Format Hard Disk, and confirm your selection twice.
- Select that you want a Custom partitioning scheme.
- Select that you want to Allot 10GB to the Other OS. This will automatically reserve the remaining disk space for the PS3's game operating system. When finished, it will restart the system.
- Once the PS3 restarts, it's ready to have Linux installed on it.

YDL Installation Steps

- Download the YDL bootloader from Terra Soft and save it as otheros.bld. This bootloader will be installed by gameOS preinstalled on PS3.

- On your flash drive create a directory called PS3. Immediately under the PS3 directory, create another directory called otheros and copy otheros.bld in this directory.
- Insert the flash drive into the PS3.
- Go to Settings, then System Settings, and then choose Install Other OS.
- Confirm the location of the installer, and follow the screens for the installation process. Note that this only installs the bootloader, not the operating system.
- When the installer finishes, go to Settings, then System Settings, and select Default System. Then choose Other OS and press the X button.
- Insert the YDL 5 DVD.
- Plug in your USB keyboard and mouse.
- Now restart the system. You can either do this by holding down the PS button on the controller and then choosing Turn off the system, or by simply holding the power button down for five seconds. Then turn the system back on.
- When it boots back up, you will get a kboot prompt.
- At this prompt type install if your output is going through the HDMI port, or installtext if you are going analog. After this YDL installation will start in a low resolution graphics mode.
- After completion of installation when you reboot PS3, you need to type in ydl480i at the kboot: boot prompt if you are using analog output. Otherwise it will likely change the output to a resolution that the analog output isn't capable of.

3 CELL SDK

IBM provides the Software Development Kit for programming and simulating the CELL Architecture. The Software Development Kit 2.1 (SDK) for the Cell Broadband Engine (Cell B.E.) is a complete package of tools to enable you to program applications on the Cell B.E. Processor. The SDK includes both PPU and SPU compilers for all the supported platforms [8]. A Cell B.E. application can run natively on a BladeCenter QS20, PlayStation 3 or in the IBM Full-System Simulator (simulator), which is supported on all of the host platforms. The SDK is composed of following items:

- SDK 2.1 Installation Guide
- Programmer's Guide
- Cell Broadband Engine Programming Tutorial
- XL C/C++ compiler
- IBM Full-System Simulator
- SIMD math AND MASS libraries
- Sample libraries and files

3.1 Prerequisites

This section shows some prerequisites for installation of CELL SDK on your host system. The Cell/B.E. SDK runs in Fedora Core 6, which must be installed before you install the SDK. Table 3.1 shows the recommended system configuration for CELL SDK installation.

Table 3.1 Recommended system configuration for CELL SDK installation

System	**Recommended minimum configuration**
x86 or x86-64	2GHz Pentium® 4 processor
PowerPC	64-bit PPC with a clock speed of 1.42 GHz. 32-bit PPC platforms are not supported.

All systems should have:

- 5 GB Hard disk space to install the source package and the accompanying development tools

- 1 GB RAM on the host system

3.2 Installing the SDK

This section shows how to install CELL SDK on your host system. The cellsdk shell script handles the SDK installation. The SDK's tools are primarily installed in /opt/ibm, although some of the toolchain commands are installed in the regular path. for example, the spu-gcc command is installed in /usr/bin on a PPC machine and in /opt/cell on an x86 machine. The simulator is installed in /opt/ibm/systemsim-cell, and the SDK in /opt/ibm/cell-sdk/prototype/. The script checks to see what features might be needed and what features are available. If some of the other installation files (from the BSC Web site) are needed, they are automatically downloaded to the directory /tmp/cellsdk-1.1.
Do the following to install the SDK :

1. As root download the SDK ISO disk image, CellSDK21.iso from the Cell/B.E. SDK alphaWorks Web site:
   http://www.alphaworks.ibm.com/tech/cellsw

2. Create a mount directory and make sure nothing else is mounted on this directory:
   ```
   mkdir –p /mnt/cellsdk
   ```

3. Mount the disk image on the mount directory:
   ```
   mount –o loop CellSDK21.iso /mnt/cellsdk
   ```
4. Change directory to /mnt/cellsdk/software:
   ```
   cd /mnt/cellsdk/software
   ```

5. Install the SDK by using the following command and answer any prompts:

```
./cellsdk install
```

6. Optionally build the samples and libraries and copy into the sysroot image for the simulator:
   ```
    cd /opt/ibm/cell-sdk/prototype/src
   ./cellsdk build
   ```

7. Change directory to any directory which is not the mount directory or below it:
   ```
    cd /
   ```
8. Unmount the disk image: umount /mnt/cellsdk

3.3 IBM’s Full-System Simulator- SystemSim

3.3.1 Simulator Overview

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 [9]. 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 1-1 shows the simulation stack. The simulator is part of the software development kit (SDK), which is available through IBM alphaWorks Emerging Technologies at http://www.alphaworks.ibm.com/tech/cellsystemsim.

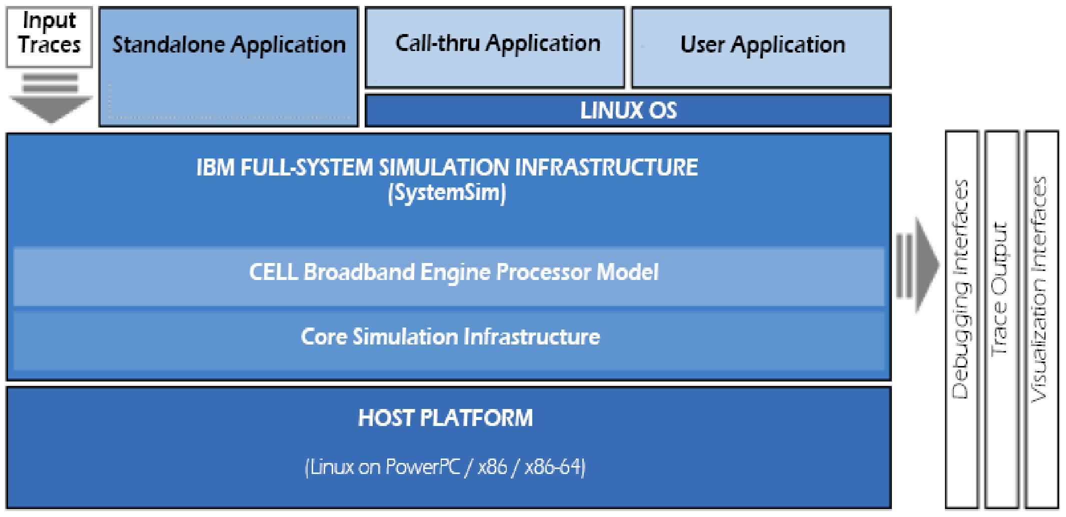

Input Traces
Standalone Application
Call-thru Application
User Application
LINUX OS
IBM FULL-SYSTEM SIMULATION INFRASTRUCTURE (SystemSim)
CELL Broadband Engine Processor Model
Core Simulation Infrastructure
HOST PLATFORM
(Linux on PowerPC / x86 / x86-64)
Debugging Interfaces
Trace Output
Visualization Interfaces

Figure 3.1 Simulator Stack for the Cell Broadband Engine

If accurate timing information and performance statistics are not required, the simulator can be used in its *functional only* mode, simulating the architectural behavior of the system to test the functions and features of a program. For performance analysis, the simulator can be used in *performance simulation* mode. The simulator is a general tool that can be configured for a broad range of microprocessors and hardware simulations. The SDK, however, provides a ready-made configuration of the simulator for Cell Broadband Engine system development.

3.3.2 Invoking the Simulator

The simulator is invoked using the systemsim command. This command is in the bin directory of the systemsim-cell release, which should be added to the user's PATH before invoking systemsim.

When the simulator starts, it loads an initial run script which typically configures and initializes the simulated machine. The name of the initial run script can be passed to systemsim with the -f option. When not specified on the command line, the simulator will look in the current directory for the file .systemsim.tcl, and if present, will use this file as the initial run script. Otherwise, it will use the file systemsim.tcl in the lib/cell directory of the systemsim-cell release. When specified using the -f option, the name of the initial run script can contain an absolute or relative path. The simulator searches for Initial run scripts with a relative path by first looking in the current directory, and then in the lib/cell directory of the systemsim-cell release, and finally in the lib directory of the systemsim-cell release. If the simulator fails to find the initial run script specified with the -f option, it issues an error message and exits.

It is generally the task of the initial run script to locate the operating system and filesystem images to be used by simulated machine. For the Cell simulator, the default initial run script searches for a Linux kernel image named vmlinux and a filesystem image named sysroot_disk. The script will look first in the current directory and then in the systemsim-cell/images/cell directory, and uses the first instance it finds for these images. If the script fails to find either of these images in one of these locations, it will print an error message and terminate the simulator.

The following examples illustrate various ways to invoke the simulator. These examples assume that the simulator was installed into /opt/ibm/systemsim-cell.

1. To run the simulator without the GUI, issue:
PATH=/opt/ibm/systemsim-cell/bin:$PATH systemsim
If the user has created a run script named .systemsim.tcl in the current directory, the simulator will use this as the initial run script. Otherwise, the simulator uses systemsim.tcl in the lib/cell directory of the systemsim-cell release as the initial run script.

2. To start the simulator with the GUI window enabled, specify the "-g" option on the command line when invoking systemsim. For example, to run the simulator with the GUI using either the user’s .systemsim.tcl or the simulator’s lib/cell/systemsim.tcl as the initial run script, issue:
PATH=/opt/ibm/systemsim-cell/bin:$PATH systemsim -g

3. To run the simulator without the gui, without a console window (-n), in quiet mode (-q), using the initial run script myrun.tcl, issue:
PATH=/opt/ibm/systemsim-cell/bin:$PATH systemsim -n -q -f myrun.tcl

When the simulator starts, the window in which it was started becomes the simulator command window where you can enter simulator commands. The simulator also creates the console window (unless this was disabled with -n) which is initially labeled UART0 in the window's title bar, and a GUI window if this was requested with the -g option.

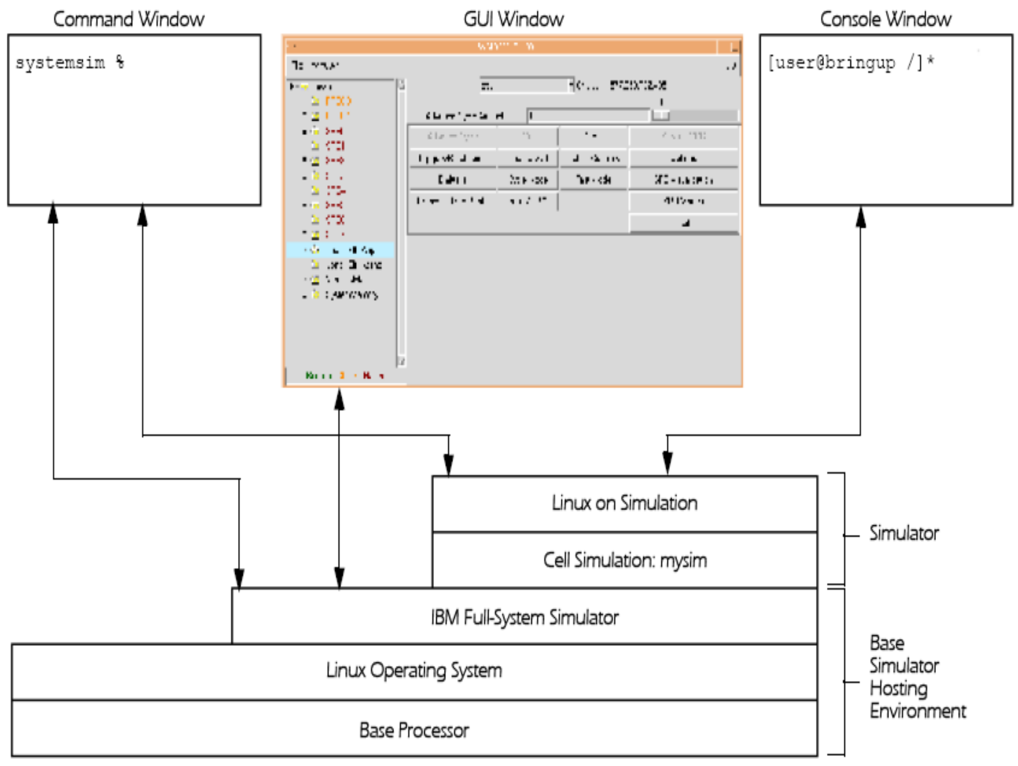


Figure 3.2 Simulator Structure and Screens

3.3.3 Operating-System Modes

A key attribute of the IBM Full-System Simulator is its ability to boot and run a complete PowerPC system. By booting an operating system, such as Linux, the IBM Full-System Simulator can execute many typical application programs that utilize standard operating system functionality. Alternatively, applications can be run in standalone mode, in which all operating system functions are supplied by the simulator and normal operating system effects do not occur, such as paging and scheduling. The IBM Full-System Simulator can also execute SPU programs in standalone mode on a given SPU. These two approaches to running applications on the simulator are referred to as Linux mode and standalone mode [9].

- **Linux Mode:** In Linux mode, after the simulator is configured and loaded, the simulator boots the Linux operating system on the simulated system. At runtime, the operating system is simulated along with the running programs. The simulated operating system takes care of all the system calls, just as it would in a non-simulation (real) environment.

- **Standalone Mode:** In standalone mode, the application is loaded without an operating system. Standalone applications are user-mode applications that are normally run on an operating system. On a real system, these applications rely on the operating system to perform certain tasks, including loading the program, address translation, and system-call support. In standalone mode, the simulator provides some of this support, allowing applications to run without having to first boot an operating system on the simulator.

However, there are limitations that apply when building an application to be loaded and run by the simulator without an operating system. For example, applications should be linked statically with any libraries they require since the standard operating system shared libraries are not available in standalone mode. Another example is support for virtual memory address translation. Typically, the operating system provides address-translation support. Since an operating system is not present in standalone mode, the simulator loads executables without address translation, so that the effective address is the same as the real address. Therefore, all addresses referenced in the executable must be valid real addresses.

3.3.4 Graphical User Interface

The simulator's GUI offers a visual display of the state of the simulated system, including the PPE and the eight SPEs. You can view the values of the registers, memory, and channels, as well as viewing performance statistics. The GUI also offers an alternate method of interacting with the simulator.

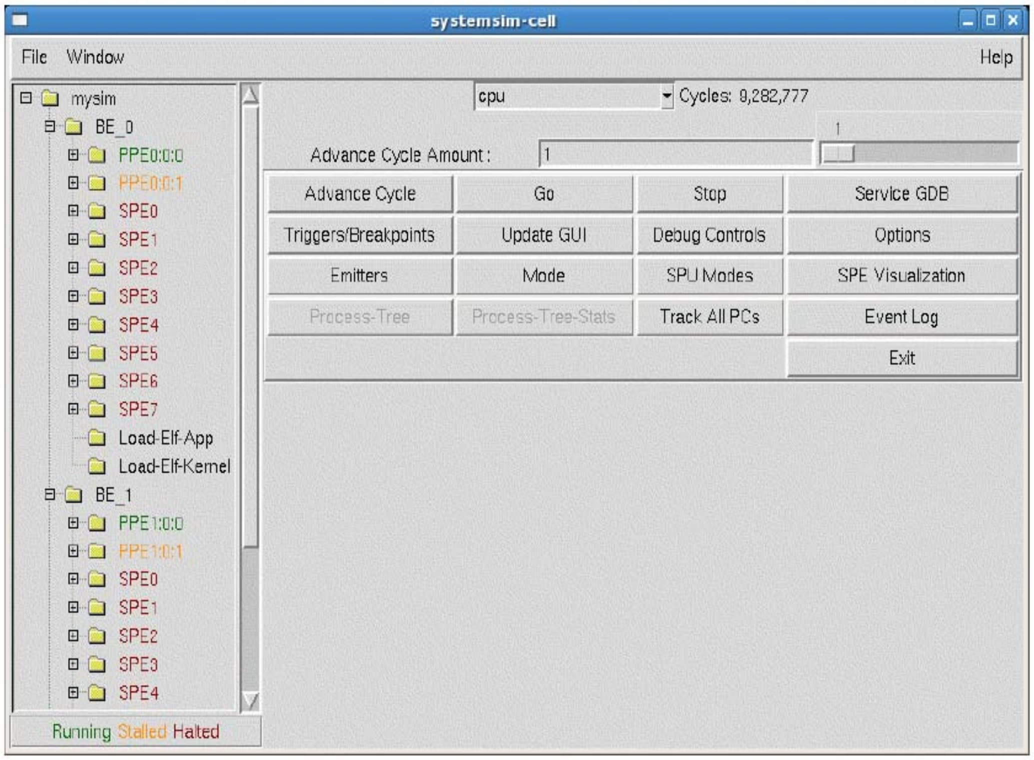


Figure 3.3 Graphical User Interface for the Simulator

The main GUI window has two basic areas: the vertical panel on the left, and the rows of buttons on the right. The vertical panel represents the simulated system and its components. The rows of buttons are used to control the simulator. When the simulator is started it creates a simulated machine containing a Cell Broadband Engine processor and displays the main GUI window, labeled with the name of the simulator program. When the GUI window first appears, click the Go button to boot the Linux operating system.

If the simulator is launched in SMP, or dual Cell-based system, the vertical panel in the main window displays each BE with its components, as shown in Figure 3.4

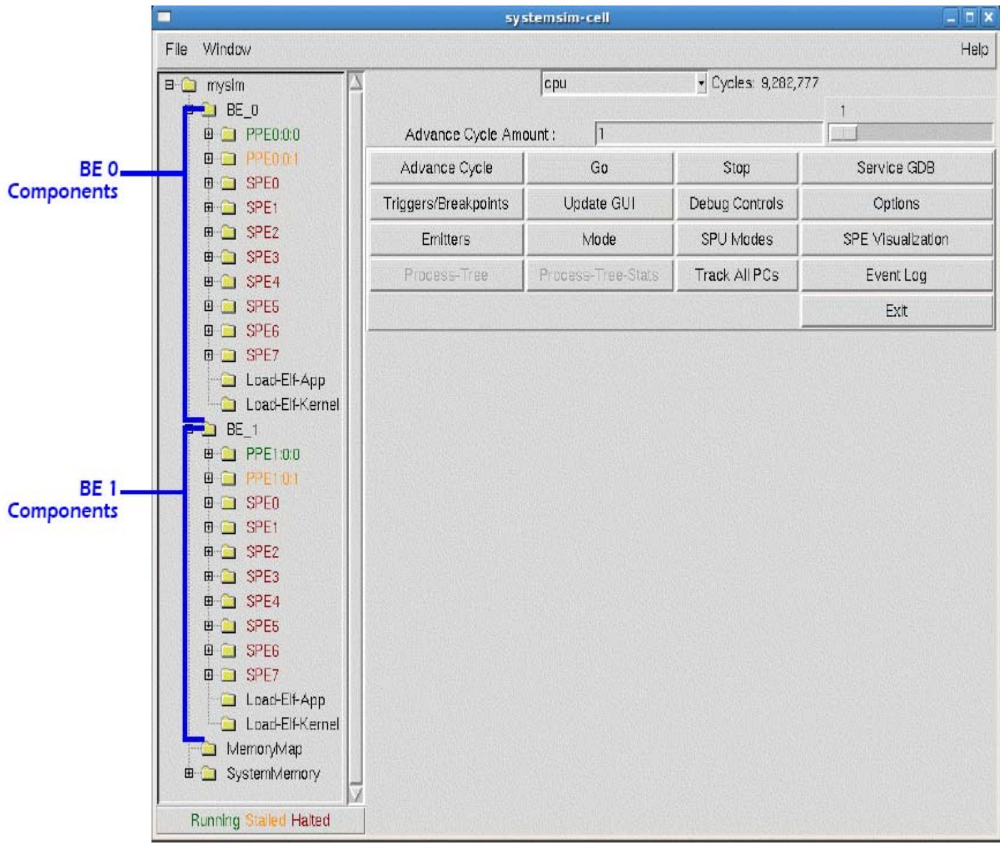


Figure 3.4 Simulator GUI started in SMP mode

3.3.5 Accessing the Host Environment: 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.

- To stop the simulator and return control back to currently active run script or the GUI / command line, issue
  callthru sink <filename>

3.3.6 Simulator Support for Performance Analysis

The simulator provides several modes of functional-only and performance simulation. 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 cycle mode allows you to:

- Gather and compare performance statistics on individual components (such as the SPU) or full systems.
- Characterize the system workload.
- Forecast performance at future loads, and fine-tune performance benchmarks for future validation.

The performance models described above can be enabled from the GUI using the performance models dialog or with simulator commands. The performance models can be enabled at any time after the simulated machine has been defined, but typically are not enabled until after the operating system has been booted. Note, however, that once the performance models for the memory subsystem are enabled, the simulation will continue in this mode until the session is terminated. To enable the performance models in a simulation, complete the following steps:

- Click the Perf Models button on the main GUI window.

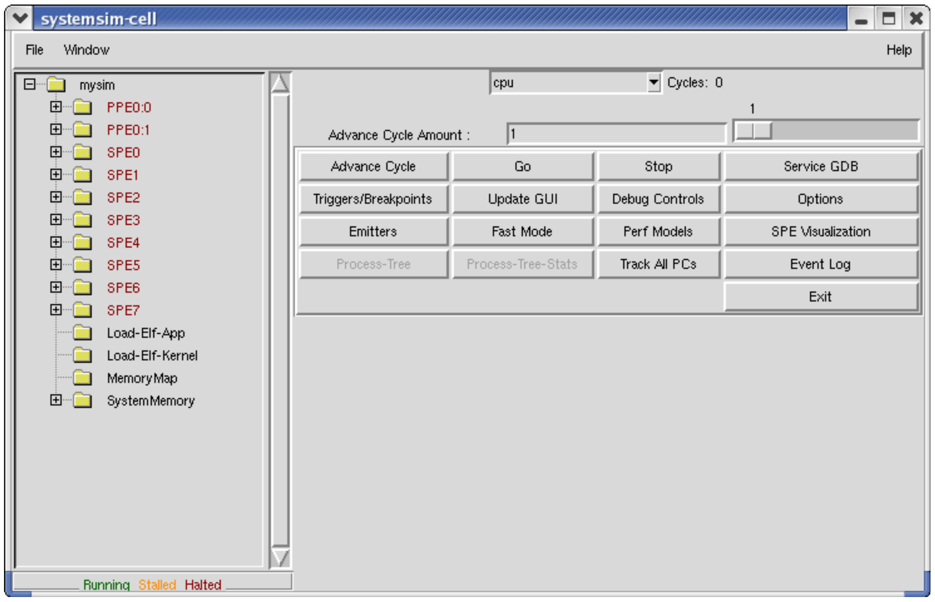


Figure 3.5 SystemSim Cell Graphical User Interface

- SystemSim displays the perf window that provides checkboxes to enable the performance model for all supported system components or for only the memory subsystem.

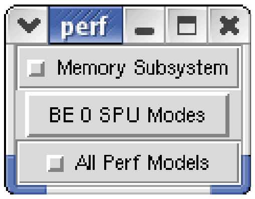
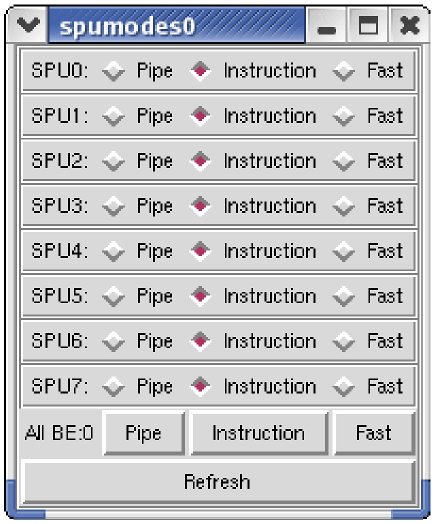


Figure 3.6 SystemSimWindow

- Click the checkbox for the desired performance models. The All Perf Models button sets the performance mode to pipeline mode for all SPUs in addition to enabling the memory subsystem performance models.

- In addition to the checkboxes, the perf window provides buttons to view timing modes for each SPU in a BE. To configure the timing modes for each SPU in a BE, click the BE n SPU Modes button, where n is either BE 0 or BE 1 if the simulator is launched in SMP mode; otherwise only a button for BE 0 is displayed.

- SystemSim displays the spumodes*n* (where *n* indicates the BE number):



SPU0: Pipe [✓ Instruction] Fast
SPU1: Pipe [✓ Instruction] Fast
SPU2: Pipe [✓ Instruction] Fast
SPU3: Pipe [✓ Instruction] Fast
SPU4: Pipe [✓ Instruction] Fast
SPU5: Pipe [✓ Instruction] Fast
SPU6: Pipe [✓ Instruction] Fast
SPU7: Pipe [✓ Instruction] Fast

Figure 3.7 SystemSim Spu Timing Modes

a. For each individual SPU, click the level of timing mode to simulate: Pipe, Instruction, or Fast.

b. To enable the same timing mode for all SPUs in the BE, click the corresponding button.

c. Click Refresh to synchronize the window with any changes to the modeling mode that may have been updated by the command line interface or the tree view.

- Click Go in the main GUI window to start the simulation with performance models enabled.

3.3.7 Performance Profile Checkpoints

The simulator can collect performance statistics for each SPU running in pipeline mode that are useful in determining the sources of performance degradation, such as channel stalls and instruction-scheduling problems. You can also use performance profile checkpoints to delimit a specific region of code over which performance statistics are to be gathered.

Performance profile checkpoints can be used to capture higher-level statistics such as the total number of instructions, the number of instructions other than no-op instructions, and the total number of cycles executed by the profiled code segment. The checkpoints are special no-op instructions that indicate to the simulator that some special action should be performed. No-op instructions are used because they allow the same program to be executed on real hardware. he application program interface (API) for the performance profile checkpoints is defined in the profile.h header file. This file provides the C-language procedures, named prof_cp{n}() where n is a numeric value ranging from 0 to 31, that generate the special no-op instructions. In addition to displaying performance information, certain performance profile checkpoints can control the statistics-gathering functions of the SPU.

For example, profile checkpoints can be used to capture the total cycle count on a specific SPE. The resulting statistic can then be used to further guide the tuning of an algorithm or structure of the SPE. The following example illustrates the profile-checkpoint code that can be added to an SPE program in order to clear, start, and stop a performance counter:

```
#include <profile.h>
. . .
prof_clear(); // clear performance counter
prof_start(); // start recording performance statistics
. . .
<code_to_be_profiled>
. . .
prof_stop(); // stop recording performance statistics
```

When a profile checkpoint is encountered in the code, the simulator prints data identifying the calling SPE and the associated timing event. The data is displayed on the simulator control window in the following format:

SPU*n*: CP*m*, *xxxxx*(*yyyyy*), *zzzzzzz*

where n is the number of the SPE on which the profile checkpoint has been issued, m is the checkpoint number, xxxxx is the instruction counter, yyyyy is the instruction count excluding no-ops, and zzzzzz is the cycle counter.

4 PROGRAMMING THE CELL

4.1 Programming Overview

The instruction set for the PPE is an extended version of the PowerPC instruction set. The extensions consist of the Vector/SIMD Multimedia Extension instruction set plus a few additions and changes to PowerPC instructions. The instruction set for the SPE is similar to that of the PPE's Vector/SIMD Multimedia Extension instruction set. Although the PPE and the SPEs execute SIMD instructions, the two instruction sets are different, and programs for the PPE and SPEs must be compiled by different compilers. C language extension for both PPE and SPE instruction set and their compilers are provided with SDK.

In order to maximize the performance of the Cell, the following two points need to be paid attention to [5]:

- Operate multiple SPEs in parallel to maximize operations that can be executed in certain time unit.
- Perform SIMD parallelization on each SPE to maximize operations can be executed per instruction.

4.2 Application Partitioning

Programs running on the Cell Broadband Engine's nine processor elements typically partition the work among the available processor elements. In determining when and how to distribute the workload and data, take into account the following considerations [5]:

- 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

The main model for partitioning an application is PPE-centric, as shown in Figure 4.1.

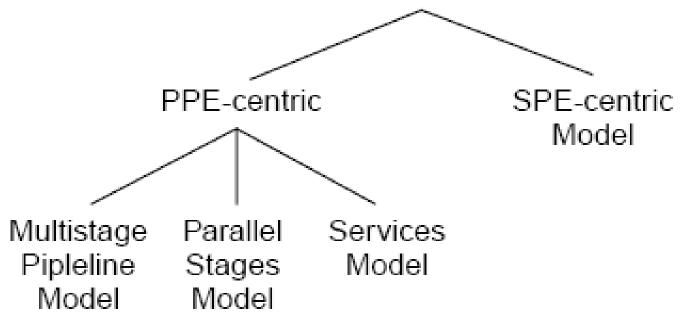


Figure 4.1 Application Partitioning Model

In the PPE-centric model, the main application runs on the PPE, and individual tasks are offloaded to the SPEs. The PPE then waits for, and coordinates, the results returning from the SPEs. This model fits an application with serial data and parallel computation. In the SPE-centric model, most of the application code is distributed among the SPEs. The PPE acts as a centralized resource manager for the SPEs. Each SPE fetches its next work item from main storage (or its own local store) when it completes its current work.

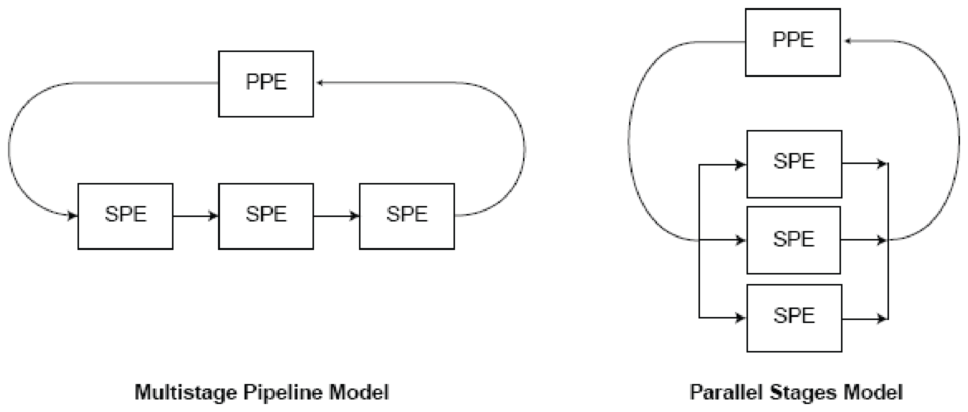


Figure 4.2 PPE-Centric Multistage Pipeline Model and Parallel Stages Model

There are three ways in which the SPEs can be used in the PPE-centric model (1) Multistage Pipeline Model (2) the Parallel Stages Model and (3) the Services Model [5]. The first two of these are shown in Figure 4.2.

If a task requires sequential stages, the SPEs can act as a multistage pipeline. The left side of Figure 4.2 shows a multistage pipeline. Here, the stream of data is sent into the first SPE, which performs the first stage of the processing. The first SPE then passes the data to the next SPE for the next stage of processing. After the last SPE has done the final stage of processing on its data, that data is returned to the PPE. As with any pipeline architecture, parallel processing occurs, with various portions of data in different stages of being processed. Multistage Model increases the data-movement requirement because data must be moved for each stage of the pipeline.

If the task to be performed is not a multistage task, but a task in which there is a large amount of data that can be partitioned and acted on at the same time, then it typically make sense to use SPEs to process different portions of that data in parallel. This Parallel Stages Model is shown on the right side of Figure 4.2.

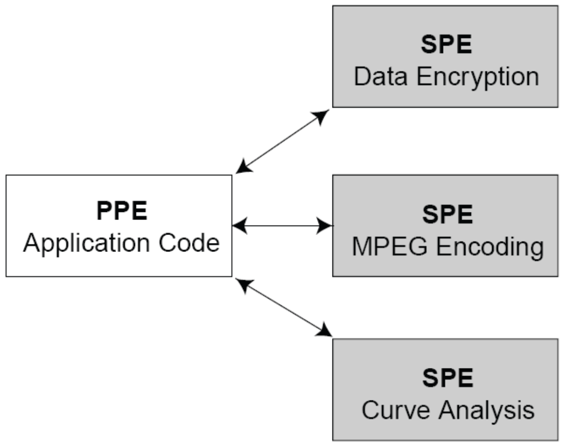


Figure 4.3 PPE-Centric Services Model

The third way in which SPEs can be used in a PPE-centric model is the Services Model. In the Services Model, the PPE assigns different services to different SPEs, and the PPE's main process calls upon the appropriate SPE when a particular service is needed. Figure 4.3 shows the PPE-centric Services Model.

4.3 Data Partitioning

With Cell programming, how the work is partitioned among available SPEs is an important consideration. Although there are many ways of doing this, the following explanation is based on a model that subdivides data to enable concurrent processing. The data-partitioning application model parallels the same program across multiple SPEs. Data is partitioned by the PPE program and uniformly distributed to SPEs. Figure. 4.4 provide an image of this approach.

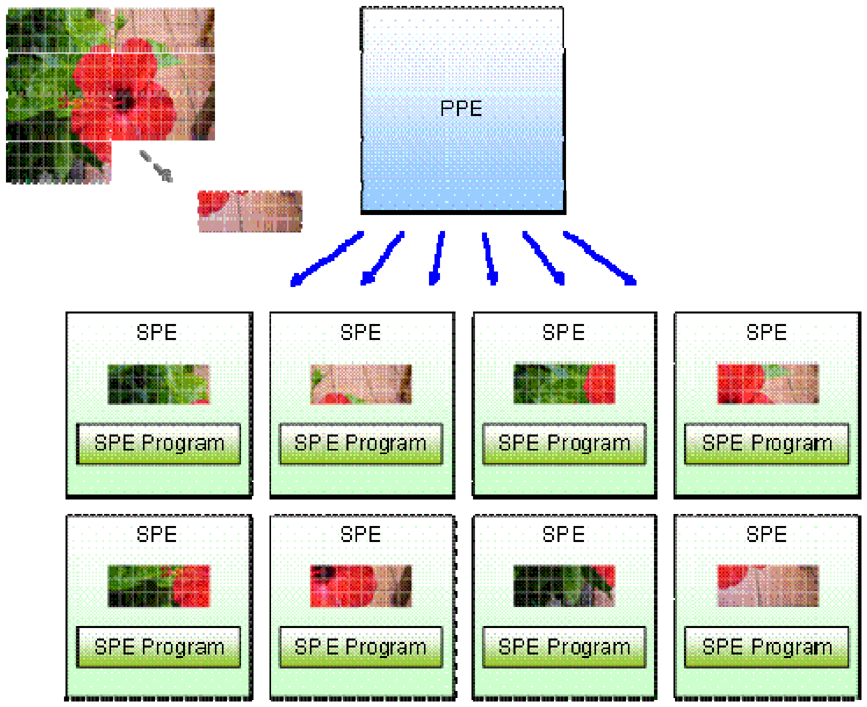


Figure 4.4 Data Partitioning for Parallel Processing

4.4 Program Control and Data Flow

In line with the Cell architecture, let's take a look at how PPE and SPE programs are executed, together with how necessary data is transmitted and received [5].Steps defined in figure 4.5:

1. PPE program loads the SPE program to the LS.
2. PPE program instructs the SPEs to execute the SPE program.
3. SPE program transfers required data from the main memory to the LS.

4. SPE program processes the received data.
5. SPE program transfers the processed result from the LS to the main memory.
6. SPE program notifies the PPE program of the termination of processing.

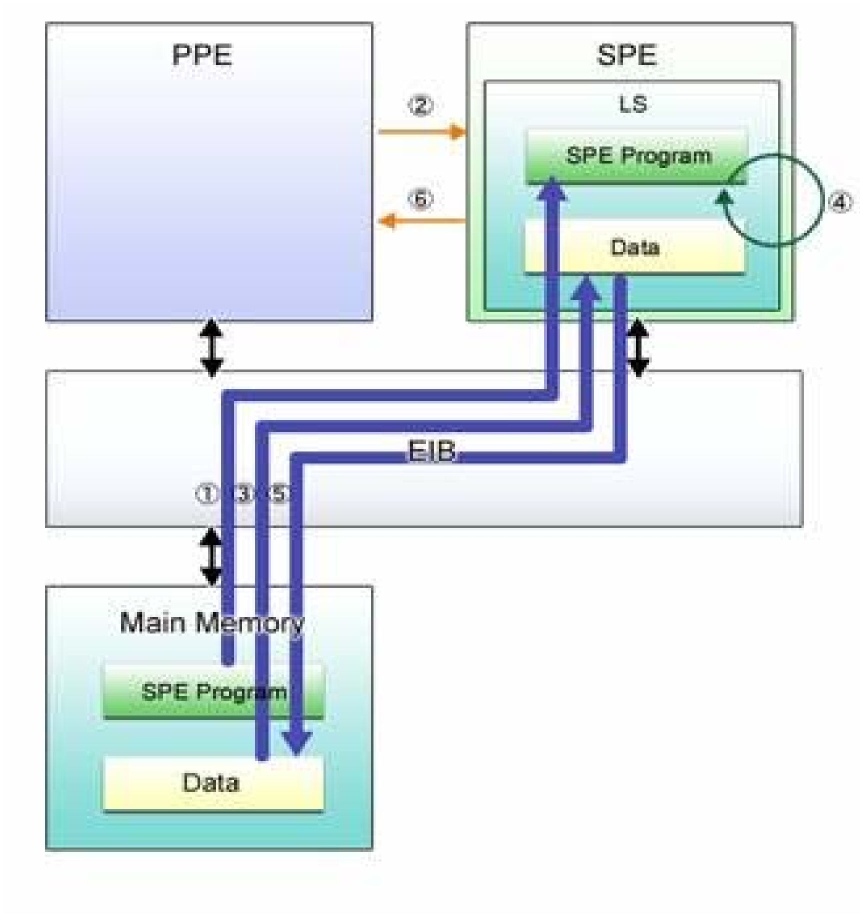


Figure 4.5 SPE program execution sequence

4.5 Vector SIMD'ization and Loop Unrolling [12]

The power of the CELL is in its SPEs. The SPEs are inherently vector processors, probably the most powerful short vector SIMD engines in existence today, capable of processing multiple data elements in the same clock cycle, e.g., four single precision floating point numbers. At the same time they do not efficiently implement scalar (non-vector) operations. Non vector operations are implemented using a so-called preferred slot, which is a location within a vector, which can be operated upon in a non-vector fashion. Using the preferred slot requires the extra operations of shifting the element to the preferred slot and then shifting it back to its original location, which introduces costly overheads.

As a result, great speeds can be achieved using vectorization, and very poor performance can be expected when programming the SPEs using standard, scalar, C code. However, the fact that a standard C code will compile and run on the SPE gives the programmers a great starting point and allows for applying optimizations in a gradual manner.

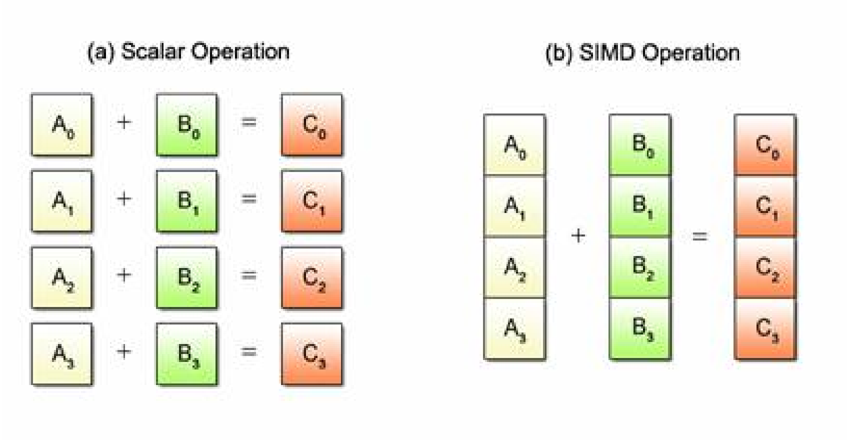


Figure 4.6 Scalar Vs SIMD Operation

Going to the level of assembly is not required. C language extensions, called intrinsics, are provided for convenience. Along with the intrinsics come vector type definitions and vector literals. Most of the intrinsics have a one to one translation to assembler instructions. Coding using the intrinsic is very close to coding in assembly, while leaving room for the compiler to do instruction reordering and register coloring.

The goal of vectorization is to eliminate scalar operations, which means eliminating insertions and extractions of elements to and from a vector. This goal can be achieved by using shuffle operations to rearrange elements within a vector (Figure 4.7).

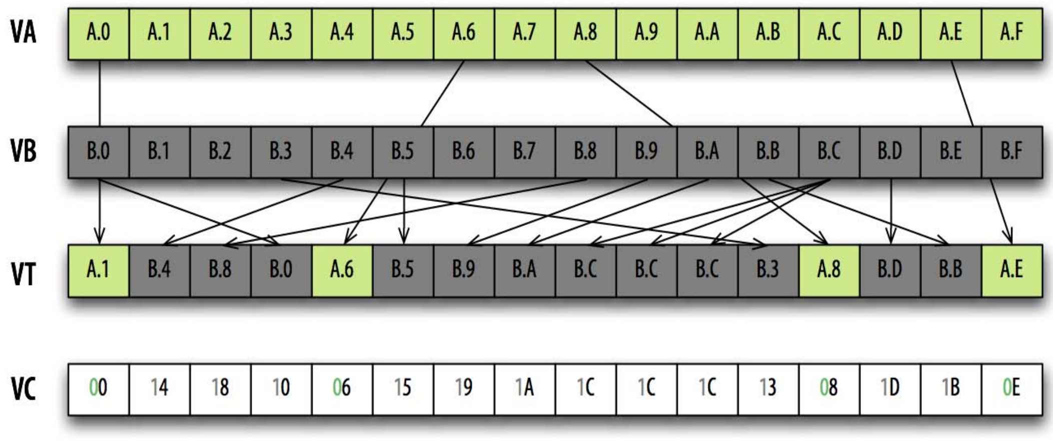


Figure 4.7 Shuffle Operation

The dual issue pipelines of the SPE enable performing shuffles in parallel with arithmetic operation, which minimizes their impact. In many situations, shuffles can be almost completely hidden behind arithmetic.

The technique of loop unrolling goes hand in hand with vectorization. Compilers can do a great job scheduling operations and optimizing for dual issues, if they are provided with many independent operations. Independent is the keyword here, since inexperienced programmers tend to attempt unrolling by putting a number of instructions in the loop, which carry dependencies from one instruction to another. As always, loop unrolling also plays its traditional role of minimizing the overhead of loop branches.

Unroll heavily. Take advantage of the enormous set of 128 vector registers at the SPE's disposal. Create big chunks of straight line code (big basic blocks). Keep your source small and readable by using defines and nested defines. There is a limit to the usability of the technique. At some point you can exhaust the register file and experience a performance drop. Also, the code size may become an issue due to the size of the local store. Nevertheless, do not be shy when unrolling. Do not unroll just a few iterations. Unroll tens of iterations. In case of nested loops, think if it is feasible to unroll the inner loop completely.

4.6 Intra-Chip Communication: DMA and Mailboxes [12]

The CELL processor provides three basic methods of communication - DMA transfers, mailboxes and signals. Here we will focus on the first two mechanisms, DMAs and mailboxes, since these are most useful in programming any kind of numerical applications. DMA transfers are the most important means of communication on the CELL processor, facilitating both bulk data transfers and synchronization. In case of SPE to SPE communication, the other two communication mechanisms are actually implemented by means of multiple DMA transfers. Nevertheless, the most important function of DMAs is bulk data movement, equivalent to message passing with MPI. Here we briefly summarize the capabilities of DMA transfers:

- DMA transfers enable exchange of data between the main memory and the local stores of the SPEs, as well as transfers from one local store to another.

- The messages can be of size 1, 2, 4, 8, and 16 bytes, and multiplicities of 16 bytes up to 16KB. Source and destination addresses of messages 16 bytes and larger have to be 16 bytes aligned, and addresses of messages shorter than 16 bytes require the same alignment as the message size. Additionally, messages of subvector sizes (less than 16 bytes) have to have the same alignment of source and destination addresses within the vector.

- Messages larger than 16KB can only be achieved by combining multiple DMA transfers. DMA lists are a convenient facility to achieve this goal, as well as to implement strided memory access. A DMA list can combine up to 2048 DMA transfers.

- DMA transfers are most efficient if they transfer at least one cache line and if they are aligned to the size of a cache line, which is 128 bytes.

- DMA transfers are non-blocking in their very nature. While DMAs are in progress, the SPE should be doing some useful work and only check for DMA completion, when it comes to processing of the transferred data.

- DMA engines are parts of the SPEs. Each SPE can queue up to 16 requests in its own DMA queue. Each DMA engine also has a proxy DMA queue, which can be accessed by the PPE and other SPEs. The proxy queue can hold up to eight requests.

Both the SPEs and the PPE are capable of initiating DMAs, but the SPE-initiated DMAs are more efficient and should be given preference over the PPE-initiated DMAs.

Although each single SPE has a theoretical bandwidth of 25.6 GB/s, which is equal to the peak bandwidth of the main memory, a single SPE will have a hard time saturating this bandwidth. In order to get good utilization of the bus, you should initiate many requests from many SPEs, and also restrain
from ordering the messages, if possible, to give the arbiter the most room for traffic optimization.

An important aspect of the CELL communication system is the efficiency of local store to local store communication. The main memory offers considerable bandwidth of 25.6 GB/s. At the same time, however, the bus connecting the PPE, the SPEs and the main memory possesses much greater internal bandwidth of 204.8 GB/s. The important aspect here is that the bus is almost impossible to saturate by communication between the interconnected elements. It means that the SPEs, when accessing the main memory heavily, will exhaust the memory bandwidth. At the same time, when communicating between one another, they will never encounter a communication bottleneck. By the same token, if an application has the potential for SPE to SPE communication, such communication should definitely be given preference over main memory communication. An example of such patterns would be stream processing, where data is passed from one SPE to another in a pipeline fashion. Although local store to local store transfers may seem a little less straightforward than main memory transfers, in practice they are not difficult to implement at all. Given that the CELL processor implements a global addressing scheme, in which each local store can be accessed by its effective address, local store to local store communication can be implemented as follows:

- The PPE retrieves the effective address of each local store by calling the spe_get_ls() function.
- The PPE passes the list of addresses of all local stores to all SPEs, through a DMA transfer.
- On the SPE side, a communication buffer is declared as a global variable and as a result has the same physical addresses within the local store on all SPEs.
- An SPE sums the physical buffer address with the effective address of the local store of another SPE to get the address of the remote buffer. It uses this address as the source address to pull data from the other SPE, or as a destination address to push data to the other SPE.

Local store to local store communication may prove invaluable not only for bulk data transfers, but also for synchronization between SPEs. One thing to remember here is the subvector alignment of source and destination for subvector length transfers.

Mailboxes are a convenient mechanism for sending short, 32-bit messages from the PPE to the SPEs and between the SPEs. The mailboxes are First-In-First-Out (FIFO) queues, meaning the messages are processed in the order of their issue. Each SPE has a four-entry mailbox for receiving incoming messages from the PPE and other SPEs, and two one-entry mailboxes for sending outgoing messages to the PPE and other SPEs - one of which serves the purpose of raising an interrupt on the receiving device. Mailbox operations have blocking nature on the SPE. An attempt to write to a full outbound mailbox will stall until the mailbox is cleared by a PPE read. Similarly, an attempt to read from an empty inbound mailbox will stall until the PPE writes to the mailbox. The same does not apply to the PPE. Neither an attempt to write to a full mailbox nor an attempt to read an empty mailbox will stall the PPE. Mailboxes are useful to communicate short messages, such as completion flags or progress status. They can also serve the purpose of communicating short data, such as storage addresses and function parameters.

The blocking nature of the mailboxes on the SPE side makes them perfect for the PPE to initiate actions on the SPEs. However for two reasons they should not be used by the SPEs to acknowledge completion of operations to the PPE. DMA completion has a local meaning on the SPE. In other words, completion of a DMA on the SPE means that the local buffers are available for reuse, but not that the data made it to the memory. If a DMA transfer is immediately followed by an acknowledgment mailbox message, the message can make it to the PPE before the data. Also, the PPE continuously reading the SPE’s outbound mailbox will flood the bus causing loss of bandwidth. A better way of acknowledging completion of an operation or a data transfer from an SPE to the PPE is to use an acknowledgment DMA protected by a fence with respect to the data transfer DMA. The PPE can periodically test the memory location (variable) written to by the SPE, or even spin (busy wait) on the variable, whichever is appropriate. You should remember to declare the variable as volatile to prevent the compiler from optimizing it out. A standard communication scenario could have the following structure:

- The PPE sends a mailbox message to the SPE, waiting on a mailbox.

- The SPE receives the mailbox message and interprets the command.

- The SPE pulls data for processing from the main memory, performs appropriate actions, and returns the result to the main memory.
- The SPE sends an acknowledgment message to the PPE by placing a value in the acknowledgment variable in the main memory.

- The PPE tests the acknowledgment variable and receives the completion notification.

5 IMPLEMENTATION OF SSD ALGORITHM

5.1 Introduction to Stereo matching

The term image matching means automatically correspondence establishment, between primitives extracted from two or more (digital) images depicting at least partly the same physical objects in space. Thus the 3D information of the objects can be computed. In photogrammetry and remote sensing, image matching techniques have been employed for automatic relative and absolute orientation, point transfer in aerial triangulation image registration and automatic DSM generation.

5.2 SSD Algorithm

The most common approach in both stereo disparity calculations and motion compensation is to slide a block taken from one image over a second image. This approach is known as the *Block Matching Algorithm*. At each possible offset, a square-sense error is computed. Finding the position where the sub-images are most similar (and the minimum error occurs) is equivalent to computing the disparity.

Disparities typically have a small dynamic range (often < 8 pixels) compared to the actual distances to objects. Therefore, measuring disparities to integral pixel values results in very low depth resolution.

Finding corresponding points for every pixel in an image is an extremely computationally expensive task. Consider a straightforward implementation: for every pixel in the left image, a surrounding block of pixels is slid across a row from the right image (which is the same height as the block from the left, but the width of the whole image.) At each position, the square-sense error (or other error metric) is computed, involving a large number of additions and multiplications.

Various optimizations intended to reduce the amount of computation have been proposed. Rather than searching an entire row, a subset of it is usually selected based on an estimate of the maximum disparity likely to be seen in the data.The search range can also be dynamically adjusted by exploiting the fact that nearby points are likely to have similar disparities.

Another class of optimizations relies on the observation that the error function as a function of horizontal offset (from which disparity is determined) is typically quite smooth, with a single and dramatic minimum.

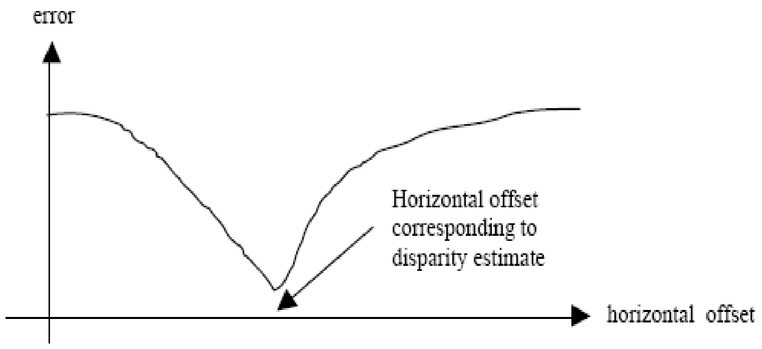


Figure 5.1. Typical error versus horizontal offset curve.

The smoothness of the error curve often makes it possible to find the minimum without an exhaustive search. For example, one can sample the error curve at a relatively small number of points, and select the best point(s) for further refinement. Logarithmic searches, common in motion compensation applications, employ this exact strategy.

Methods can be used for Correlation are:

1. SSD (Sum of Squared Differences)
2. SAD(Sum of Absolute Differences)
3. NCC (Normalized Cross Correlation)
4. ZNCC (Zero mean Normalized Cross Correlation)

Disparity Computation Using SSD

D =maximum disparity

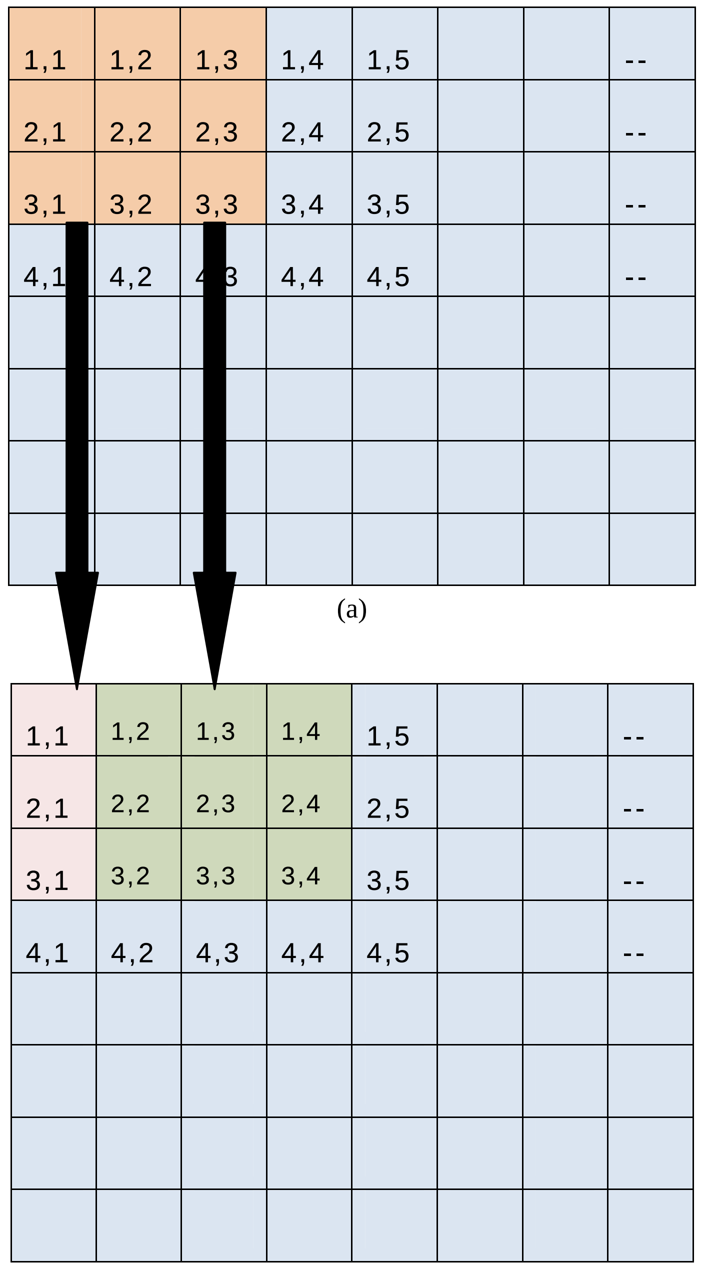
SSD(x, y) = SUM OF SQUARED DIFFERENCES of window centered at (x,y) in left image with maximum +D resolution window in Right Image

d = gives disparity with minimum SSD in horizontal line with –D to +D horizontal displacement.

Disparity(x,y,d)=Min {SSD(x+0,y),...,SSD(x+D,y)}

5.3. SSD Implementation on Cell Broadband Engine:

Let us consider two images Image A and Image B.

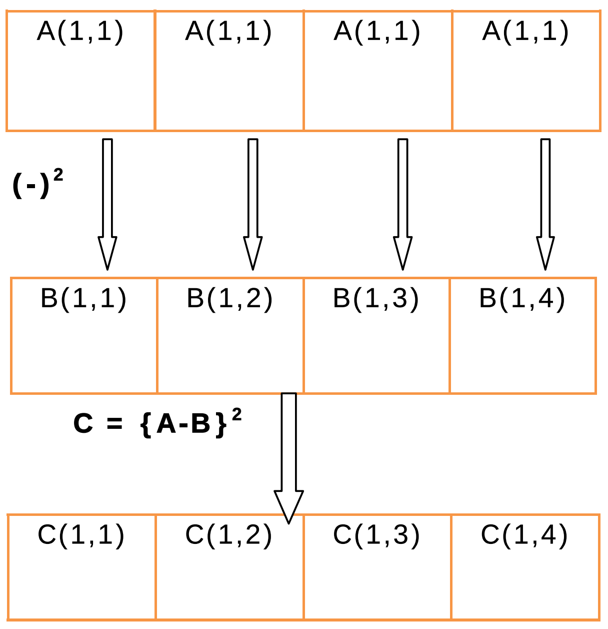


(b)

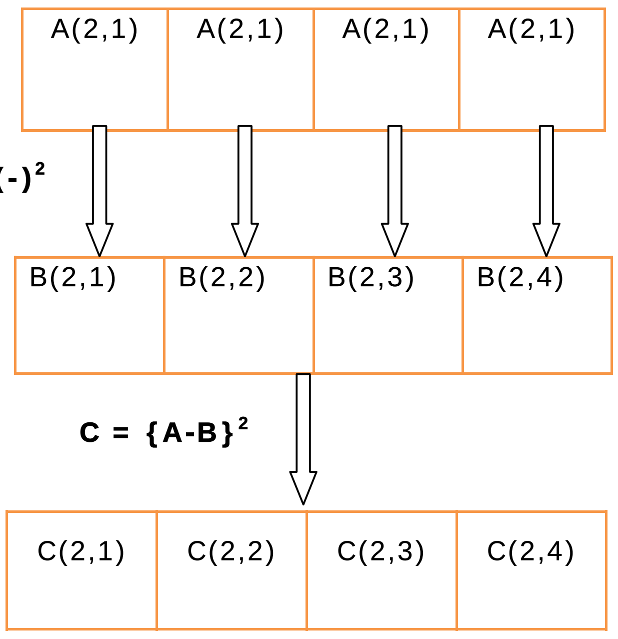
Figure 5.2 (a) Image A (b) Image B

Now first take a 3x3 mask/window from image A which will act as the base image. Now this mask is acted upon the shifted 3x3 counterparts from image B.

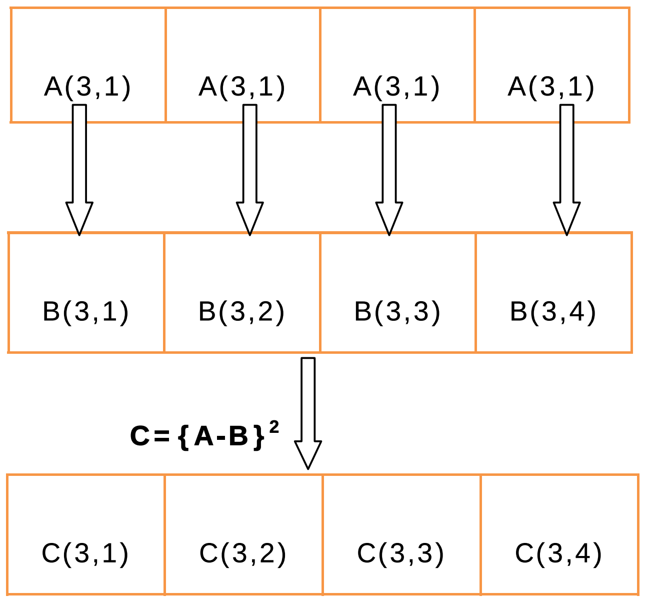
Step 1:


A(1,1)
A(1,1)
A(1,1)
A(1,1)
(-)²
B(1,1)
B(1,2)
B(1,3)
B(1,4)
C = {A-B}²
C(1,1)
C(1,2)
C(1,3)
C(1,4)

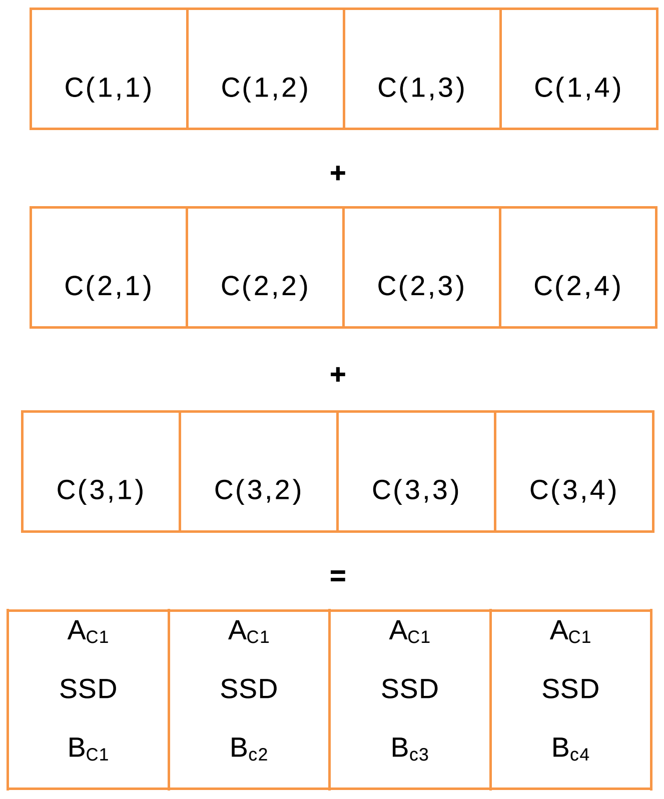
Step 2:


A(2,1)
A(2,1)
A(2,1)
A(2,1)
(-)²
B(2,1)
B(2,2)
B(2,3)
B(2,4)
C = {A-B}²
C(2,1)
C(2,2)
C(2,3)
C(2,4)

Step 3:



Step 4:



The above output vector will give us the SSD of A's 1st column with B's 4 successive columns (i.e. with B's 1st column, B's 2nd column, B's 3rd column and B's 4th column).In the same way, in the next step we will find out the vector containing SSD of A's 2nd column with B's 4 successive columns (i.e. with B's 1st column, B's 2nd column, B's 3rd column and B's 4th column).

A_{C2} SSD B_{C2}	A_{C2} SSD B_{c3}	A_{C2} SSD B_{c4}	A_{C2} SSD B_{c5}

Then in the next step we will find out the vector containing SSD of A's 3rd column with B's successive columns (i.e. with B's 1st column, B's 2nd column, B's 3rd column and B's 4th column).

A_{C3} SSD B_{C3}	A_{C3} SSD B_{c4}	A_{C3} SSD B_{c5}	A_{C3} SSD B_{c6}

Now we add these three resultant vectors to get the SSD of 3x3 mask (from image A) with the 4 successive shifted counterparts from Image B.

A_{C1} SSD B_{C1}	A_{C1} SSD B_{c2}	A_{C1} SSD B_{c3}	A_{C1} SSD B_{c4}

+

A_{C2} SSD B_{C2}	A_{C2} SSD B_{c3}	A_{C2} SSD B_{c4}	A_{C2} SSD B_{c5}

+

A_{C3} SSD B_{C3}	A_{C3} SSD B_{c4}	A_{C3} SSD B_{c5}	A_{C3} SSD B_{c6}

=

$Mask_{3x3}$ SSD $Mask_0$	$Mask_{3x3}$ SSD $Mask_1$	$Mask_{3x3}$ SSD $Mask_2$	$Mask_{3x3}$ SSD $Mask_3$

Here $mask_i$ = 3x3 mask from image 2 shifted by i.

Next step is to calculate the minimum of the values contained in the above vector. The index i of the minimum value is kept in the corresponding position in a new matrix.

Then the above procedure is repeated to find out the other values and the values are kept at appropriate positions in the matrix. This new matrix thus formed contains the minimum disparity values for each pixel which can be used for image matching.

5.4. Implementation details and Performance Results:

The above proposed algorithm was implemented on Cell simulator and an appreciable performance gain was achieved. Sum of Square Difference(SSD) Algorithm was developed in two different versions: (1) original (scalar) version which can run on any single core system such as Intel Pentium 4 and (2) parallel version (as explained above) which can be executed on Cell Broadband Engine. Two 256x256 BMP images were taken as input and an output array containing the minimum disparity values for each corresponding pixel was generated.

Original version of the algorithm was executed on Simple-scalar version 3.0d simulator while parallel version of the program was executed on simulator called SystemSim, which is provided with Cell Software Development Kit by IBM. In both cases, performance was measured in terms of clock cycles. The results are shown in following table:

	Compiler	**Simulator**	**Clock Cycles**
Scalar Version (P-4)	gcc	Simple-scalar	23,587,588
Parallel Version (Cell BE)	ppu-gcc spu-gcc	IBM SystemSim	3,408,936

Table 5.1 Performance comparison for SSD Algorithm

So from Table 5.1, we can observe that the Speed-up achieved is (23,587,588) / (3,408,936) = 6.92 = 7 (approx).

Now since we have used 8 SPE's (image has been divided into 8 parts) and in each SPE we are using 4 way SIMD operations. Hence theoretically, the Overall Speedup should be around 4x8=32 Times.

Amdahl's law [Appendix C] states that the overall speedup of applying the improvement will be

$$Speedup = \frac{1}{(1-P)+\frac{P}{S}}$$

Where P gives the portion of the computation time in the original machine that has been enhanced and S gives the Speedup gained for the portion P. The above equation was used to calculate the Overall Speedup.

For calculating P gprof command was used which gave us the timing information of the scalar version of the SSD algorithm. From the profiling output it was observed that the portion P of original computation time which we have parallelized is around 98.2% and the other unimproved portion is around 0.8% only. So we can say that we are improving almost 100% of the original code. Hence P=1.

We have already calculated S which is coming out to be around 7 times. Hence putting P=1 in above equation, Overall Speedup is given by,

Overall Speedup = S = 7 Times

As we can observe that the Overall Speedup is quite less as compared to the theoretically calculated Speedup which was coming out to be 32. There can be a lot of reasons because of which we are not able to fetch the Speedup of 32 times.

The huge discrepancy between the theoretical performance and the original performance can be attributed to the overhead associated with thread creation and the additional data transfers (DMA) that must take place on the EIB in order to move the data to a SPE. Another reason can be the overhead associated with the vector arrangements since one has to explicitly arrange the data in the form of vectors. Moreover one has to convert from one data type to another since SPU vector intrinsics does not have support for all the data types. So switching from vector of one type to another as per your requirements induces this additional overhead.

6 IMPLEMENTATION OF HAAR TRANSFORM

6.1 Introduction to Wavelet Transform

With the increasing growth of technology and the entrance into the digital age, we have to handle a vast amount of information every time which often presents difficulties. So, the digital information must be stored and retrieved in an efficient and effective manner, in order for it to be put to practical use. Wavelets provide a mathematical way of encoding information in such a way that it is layered according to level of detail. This layering facilitates approximations at various intermediate stages. These approximations can be stored using a lot less space than the original data. Wavelet transform is the reason behind the success of the JPEG-2000 coding. Wavelet transform exploits both the spatial and frequency correlation of data by dilations (or contractions) and translations of mother wavelet on the input data. It supports the multiresolution analysis of data i.e. it can be applied to different scales according to the details required, which allows progressive transmission and zooming of the image without the need of extra storage. Another encouraging feature of wavelet transform is its symmetric nature that is both the forward and the inverse transform has the same complexity, building fast compression and decompression routines. Its characteristics well suited for image compression include the ability to take into account of Human Visual System's (HVS) characteristics, very good energy compaction capabilities, robustness under transmission, high compression ratio etc.

The implementation of wavelet compression is very similar to that of subband coding scheme: the signal is decomposed using filter banks. The output of the filter banks is down-sampled, quantized, and encoded. The decoder decodes the coded representation, up-samples and recomposes the signal.

Wavelet transform divides the information of an image into approximation and detail sub signals. The approximation sub signal shows the general trend of pixel values and other three detail sub signals show the vertical, horizontal and diagonal details or changes in the images. If these details are very small (threshold) then they can be set to zero without significantly changing the imagethey can be set to zero without significantly changing the image. The greater the number of zeros the greater the compression ratio. If the energy retained (amount of information retained by an image after compression and decompression) is 100% then the compression is lossless as the image can be reconstructed exactly. This occurs when the threshold value is set to zero, meaning that the details have not been changed. If any value is changed then energy will be lost and thus lossy compression occurs. As more zeros are obtained, more energy is lost. Therefore, a balance between the two needs to be found out.

This Chapter covers the simplest of all wavelet transforms: Haar wavelet transform of an image and its implementation on Cell B.E.

6.2 Haar Wavelet Transform

The 2D-Haar wavelet transform involves two operations namely Averaging and Differencing on each row of the image and then on each column of the image. The transformation of the 2D image is a 2D generalization of the 1D wavelet transform. It applies the1D wavelet transform to each row of pixel values. This operation provides us an average value along with detail coefficients for each row. Next, these transformed rows are treated as if they were themselves an image and apply the 1D transform to each column. The resulting values are all detail coefficients except a single overall average co efficient. In order to complete the transformation, this process is repeated recursively only on the quadrant containing averages.

1D-Haar transform consists of following two operations-

Example: data = (5 7 6 5 3 4 6 9)

Average coefficients: (5+7)/2, (6+5)/2, (3+4)/2, (6+9)/2

Detail coefficients: (5-7)/2, (6-5)/2, (3-4)/2, (6-9)/2

- After 1st pass=(6 5.5 3.5 7.5 | -1 0.5 -0.5 -1.5)
- After 2nd pass=(23/4 22/4 | 0.25 -2 -1 0.5 -0.5 -1.5)
- After 3rd pass=(45/8 | 1/8 0.25 -2 -1 0.5 -0.5 -1.5)

Now let us see how the 2D Haar wavelet transformation is performed. The image is comprised of pixels represented by numbers. Consider the 8×8 image taken from a specific portion of a typical image. The matrix (a 2D array) representing this image is shown in Figure 6.1(a).

$$
\begin{array}{cccccccc}
12 & 12 & 12 & 12 & 14 & 12 & 12 & 12 \\
12 & 12 & 12 & 12 & 14 & 12 & 12 & 12 \\
12 & 12 & 12 & 12 & 14 & 12 & 12 & 12 \\
12 & 12 & 12 & 12 & 14 & 12 & 12 & 12 \\
12 & 12 & 12 & 12 & 14 & 12 & 12 & 12 \\
16 & 16 & 16 & 16 & 14 & 16 & 16 & 16 \\
12 & 12 & 12 & 12 & 14 & 12 & 12 & 12 \\
12 & 12 & 12 & 12 & 14 & 12 & 12 & 12
\end{array}
\qquad
\begin{array}{cccc|cccc}
12 & 12 & 13 & 12 & 0 & 0 & 2 & 0 \\
12 & 12 & 13 & 12 & 0 & 0 & 2 & 0 \\
12 & 12 & 13 & 12 & 0 & 0 & 2 & 0 \\
12 & 12 & 13 & 12 & 0 & 0 & 2 & 0 \\
12 & 12 & 13 & 12 & 0 & 0 & 2 & 0 \\
16 & 16 & 15 & 16 & 0 & 0 & \underline{2} & 0 \\
12 & 12 & 13 & 12 & 0 & 0 & 2 & 0 \\
12 & 12 & 13 & 12 & 0 & 0 & 2 & 0
\end{array}
\qquad
\begin{array}{cccc|cccc}
12 & 12 & 13 & 12 & 0 & 0 & 2 & 0 \\
12 & 12 & 13 & 12 & 0 & 0 & 2 & 0 \\
14 & 14 & 14 & 14 & 0 & 0 & 0 & 0 \\
12 & 12 & 13 & 12 & 0 & 0 & 2 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\underline{4} & \underline{4} & \underline{2} & \underline{4} & 0 & 0 & 4 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

(a) (b) (c)

Figure 6.1 (a) 8x8 image block (b) after horizontal transform (c) after vertical transform

Now we perform the operation of averaging and differencing to each row to arrive at a new matrix (Figure 6.1(b)) representing the same image in a more concise manner. Then in the next step we perform the averaging and differencing operation on each column of the matrix of figure 6.1(b) to arrive at a new matrix of figure 6.1(c) which is the Haar representation of the original 8x8 image block.

In this way, after horizontal transform the image gets divided into two halves: one contains the low level details and the second half contains the high level details. After vertical transform, the image gets divided into four quarters: low-low, high-low, low-high, high-high. [14]

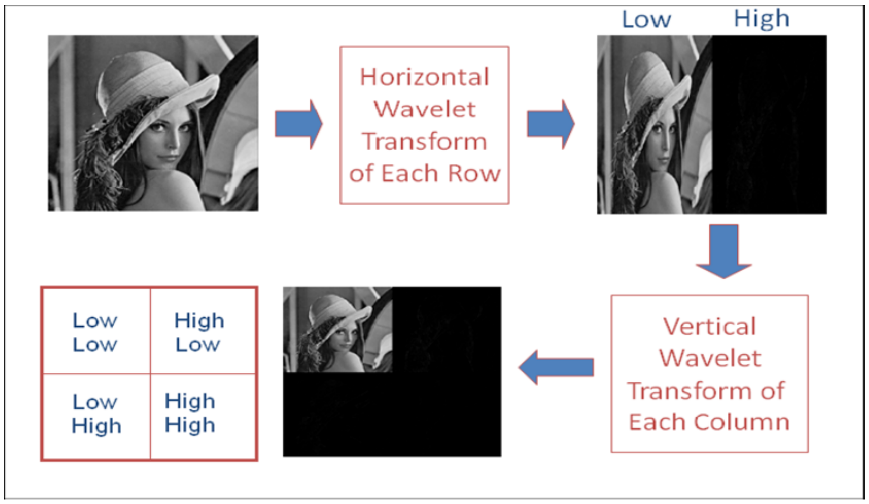

Horizontal Wavelet Transform of Each Row
Low
High
Vertical Wavelet Transform of Each Column
Low Low
High Low
Low High
High High

Figure 6.2 Haar wavelet transform

6.3 Haar Transform Implementation on Cell

The main objective of implementing Haar transform algorithm on Cell BE is to show the performance improvement over the Pentium-4 system. We basically have two stages in Haar transform: *vertical Haar transform* which involves the averaging and differencing operations on each column of the image, and *horizontal Haar transform* which involves the averaging and differencing operations on each row of the image. But here for the purpose of cycle count, only one stage i.e. vertical Haar transform have been implemented on both Cell BE and Pentium-4 and the results are compared. Since both vertical and horizontal Haar transforms involves same operations and same procedures, both would take almost same time. So for comparison purpose, only vertical Haar transform have been implemented.

First of all the whole image is divided into eight equal subimages which are then transferred to the LS's of eight SPE's through DMA transfers. Then each SPE performs the vertical Haar transform on its own subimage and the whole output image is reconstructed after all the SPE's have finished. In this way we achieved the parallelism provided by the eight cores of the Cell processor. As we already know that the Cell BE also provides another level of parallelism i.e. SIMD support inside each SPE, there is a need of restructuring the conventional scalar Haar transform algorithm to reap benefits of SIMD support. In this implementation of Haar transform, floating point vectors have been used i.e. 4-way SIMD have been used.

The algorithm was implemented on Cell simulator and an appreciable performance gain was achieved. Haar Transform Algorithm was developed in two different versions: (1) original (scalar) version which can run on any single core system such as Intel Pentium 4 and (2) parallel version which can be executed on Cell Broadband Engine. A 256x256 BMP image was taken as input which was divided into 8 32x256 BMP images. Now one 32x256 BMP image is send to each SPE where the Haar transform is applied to it making use of floating point vectors. Figure 6.1 (a) and (b) shows the input and the output images respectively.

From the output image one can easily see the strips of averaging component (i.e. low level details) and differencing component (i.e. high level details). The existence of strips in the output image is attributed to the fact that each SPE has got a 32x256 image only. Hence each SPE has divided its own 32x256 image into two halves, one representing averaging component and other representing the differencing component. When all these 32x256 output images from all SPE's are combined to generate the final 256x256 image, we got the output image of Figure 6.1(b)

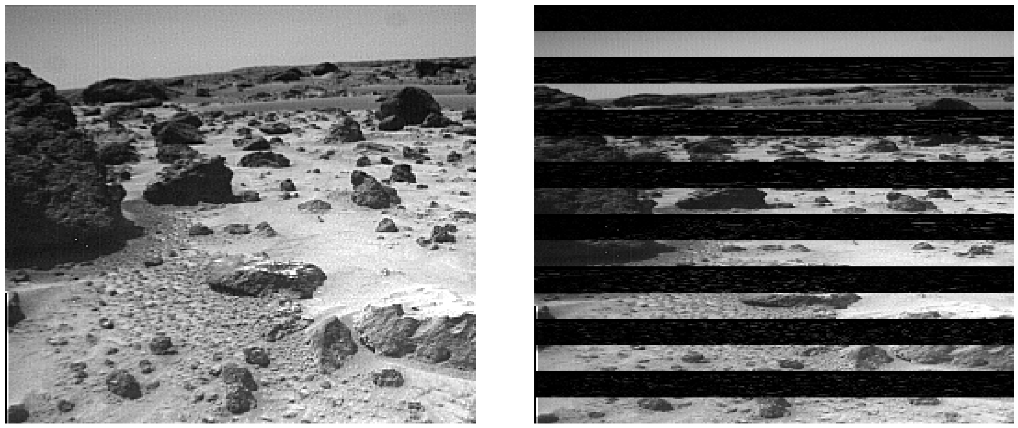


Figure 6.3 (a).Input Image (BMP format) (b).Output Image (BMP format)

Original version of the algorithm was executed on Simple-scalar version 3.0d simulator while parallel version of the program was executed on simulator called SystemSim, which is provided with Cell Software Development Kit by IBM. In both cases, performance was measured in terms of clock cycles. The results are shown in following table:

	Compiler	**Simulator**	**Clock Cycles**
Scalar Version (P-4)	gcc	Simple-scalar	20,225,198
Parallel Version (Cell BE)	ppu-gcc spu-gcc	IBM SystemSim	2,214,867

Table 6.1 Performance comparison for Haar Transform

So from Table 5.1, we can observe that the Speed-up achieved is (20,225,198) / (2,214,867) = 9.13 times.

Now here again we have used 8 SPE,s (image has been divided into 8 parts) and in each SPE we are using 4 way SIMD operations. Hence theoretically, the Overall Speedup should be around 4x8=32 Times.
We have calculated the Overall Speedup for the SSD algorithm in the 5th chapter. In the same way here again the Overall Speedup is given by,

$$\textit{Overall Speedup} = S = 9.13 \text{ times}$$

The discrepancy between the theoretical Speedup and the original Speedup that we are getting can be attributed to the reasons given in the 5th chapter.

7 CONCLUSION AND FUTURE WORK

7.1 Summary

This thesis has covered the hardware aspects, software aspects, programming environment and implementation of two image processing algorithms, SSD algorithm and Haar wavelet transform, on Cell BE. And the work successfully shows the performance gains achieved for these two algorithms.

As seen in chapter 2 Cell Broadband Engine contains one processor specifically for running operating system, while eight other cores which can be used as accelerator for many applications and can be programmed separately. All the cores are connected with each other and with other devices through a very high speed bus providing bandwidth of 96bytes/cycle. All the cores contains a DMA controller also which can be effectively used for memory access while performing computations in parallel. In addition all the cores support SIMD also, providing two levels of parallelism. Because of these features Cell can perform many times faster than conventional processors.

Cell was initially designed for high resolution gaming console but the systems based on Cell are now available, which can be efficiently used for other applications. IBM provides SDK for programming the Cell which is freely available from IBM’s website and can be easily installed in Fedora Core 6 operating system.

In chapter 4 we saw programming methods i.e. how application and data can be partitioned and work can be distributed among different cores. Maximum performance can be achieved when equal amount of work can be distributed among different cores. Image processing applications are best suited for such environment because it requires performing same operation on entire image. Hence work can be evenly distributed. However this distribution task is very tedious and requires lots of programming efforts.

From results in Chapter 5 and 6, we can see that the CELL is capable of giving a huge performance gain (around 10 times) over Pentium 4. SSD algorithm and

Haar wavelet transform algorithm gave an Overall Speedup of around 7x and 9x respectively, which is quite appreciable.

7.2 Conclusion

This work has successfully shown the glimpse of performance gains which can be achieved by using the Cell BE processor. Both, the SSD algorithm and the Haar Wavelet Transform algorithm shown an appreciable Speedup: 7x and 9x respectively.

Hence we can conclude saying that Cell's new processor architecture

promises a better performance over today's desktop processors. Graphic Processing Unit (GPU) is already able to run approximately ten times faster than the desktop processors. The Cell will finally allow the performance to be brought up to a similar level but with more applications.

The major strength of Cell is that it is for more general purpose so it can be usable for a wider variety of tasks. The Cell is designed to fit into everything from Personal Digital Assistants, up to servers so you can make a Cell computer out of completely different systems.

The Cell processor presents many unique challenges in the programming space, while at the same time providing great opportunities to see massive performance gains. This research shows these possibilities, and covers the design, implementation and performance analysis of some DIP algorithms on Cell processor. In the end, the programming model is a difficult one, the Cell's performance benefits can be difficult to achieve, but given enough time and careful consideration invested, the merits of the Cell can shine through.

7.3 Future Work

There are a number of areas that I would have liked to have touched upon, but did not have the time to do so. After seeing the impact of DMA overhead on our benchmarks, it would also be interesting to see how implementing double buffering would help to reduce memory transfer penalties. Finally, after viewing the overhead associated with thread creation, there is much work to be done in investigating how the thread scheduler on the Cell performs.

As we saw cell provides two level of parallelism: one is multiple cores and other is SIMD within each core. Because of this, theoretically we should be able to achieve maximum speedup of 32X. But from results we can see that speedup achieved is around 7X for SSD implementation and around 9X for Haar transform implementation. Though some of the reasons for this discrepancy between the theoretical Speedup and the Speedup achieved have been listed in Chapter 5, still the reasons for this has to be explored more.

[11] Rafael C. Gonzalez, Richard E. Woods, “Digital Image Processing”, Pearson Prentice Hall.

[12] Alfredo Buttari, Piotr Luszczek, “A Rough Guide to Scientific Computing On the PlayStation 3”, Technical Report UT-CS-07-595.

[13] http://www.kernel.org/pub/linux/kernel/people/geoff/cell/CELL-LinuxCL_20080201ADDON/doc/CellProgrammingTutorial/AdvancedCell Programming.html.

[14] Kamrul Hasan Talukder and Koichi Harada, “Haar Wavelet Based Approach for Image Compression and Quality Assessment of Compressed Image”, IAENG International Journal of Applied Mathematics.

REFERENCES

[1] Cell Broadband Engine Programming Handbook, version 1.0, cellsdk/pdfs.

[2] Cell Broadband Engine Programming Tutorial, version 2.1, cellsdk/pdfs.

[3] David Krolak," The Element Interconnect Bus", Papers from the Fall Processor Forum 2005, http://www.ibm.com/developerworks/power/library/pa-fpfeib/

[4] http://www.kernel.org/pub/linux/kernel/people/geoff/cell/CELL-Linux-L20070831ADDON/doc/CellProgrammingTutorial/BasicsOfCellArchitecture.html

[5] http://www.kernel.org/pub/linux/kernel/people/geoff/cell/CELL-Linux-CL20070831_ADDON/doc/CellProgrammingTutorial/BasicsOfSPEProgramming.html

[6] IBM BladeCenter QS20, cellsdk/pdfs.

[7] http://www.ibm.com/developerworks/power/library/pa-linuxps3-1/

[8] Software Development Kit 2.1 Installation Guide, Version 2.1, cellsdk/pdfs.

[9] IBM Full-System Simulator User's Guide, Version 2.1, cellsdk/pdfs.

[10] IBM Full-System Simulator Performance Analysis Guide, Version 2.1, cellsdk/pdfs.

APPENDIX A LIBSPE

A-1 Overview

The SPE Runtime Management Library (libspe) is the standardized low-level application programming interface (API) that enables application access to the Cell/B.E. SPEs. This library provides an API that is neutral with respect to the underlying operating system and its methods to manage SPEs. Implementations of libspe can provide additional functionality that enables access to operating system or implementation-dependent aspects of SPE runtime management. In general, applications do not have control over the physical SPE system resources. The operating system manages these resources. Applications manage and use software constructs called SPE contexts. These SPE contexts are a logical representation of an SPE and are the base object on which libspe operates. The operating system schedules SPE contexts from all running applications onto the physical SPE resources in the system for execution according to the scheduling priorities and policies associated with the runable SPE contexts. Libspe also provides the means for communication and data transfer between PPE threads and SPEs.The basic scheme for a simple application using an SPE is as follows:

1. Create an SPE context.
2. Load an SPE executable object into the SPE context local store.
3. Run the SPE context. This transfers control to the operating system, which requests the actual scheduling of the context onto a physical SPE in the system.
4. Destroy the SPE context.

A-2 PPE functions

To provide this functionality, libspe consists of the following sets of PPE functions to:

- Create and destroy SPE and gang contexts
- Load SPE objects into SPE local store memory for execution

- Start the execution of SPE programs and to obtain information about reasons why an SPE has stopped running
- Receive asynchronous events generated by an SPE
- Access the MFC (Memory Flow Control) problem state facilities

Some useful functions for creating SPE context and start execution of SPE program are given below:

- **spe_context_create:** Create a new SPE context.
- **spe_context_destroy:** Destroy the specified SPE context.
- **spe_image_open:** Open an SPE ELF executable.
- **spe_image_close:** Close an SPE ELF object.
- **spe_program_load:** Load an SPE main program.
- **spe_context_run:** Request execution of an SPE context.

APPENDIX B SIMPLE SCALAR SIMULATOR

The Simple-Scalar tool set is a system software infrastructure used to build modelling applications for program performance analysis, detailed micro architectural modelling, and hardware-software co-verification. Using the Simple Scalar tools, users can build modelling applications that simulate real programs running on a range of modern processors and systems. The tool set includes sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction.

Simple-Scalar simulators can emulate the Alpha, PISA, ARM, and x86 instruction sets. The tool set includes a machine definition infrastructure that permits most architectural details to be separated from simulator implementations. All of the simulators distributed with the current release of Simple-Scalar can run programs from any of the above listed instruction sets.
The Simple-Scalar distribution comes with three types of simulators: Sim-fast, sim-safe and sim-outorder. Sim-fast does no time accounting, only functional simulation—it executes each instruction serially, simulating no instructions in parallel. Sim-fast is optimized for raw speed, and assumes no cache, instruction checking. A separate version of sim-fast, called sim-safe, also performs functional simulation, but checks for correct alignment and access permissions for each memory reference. The most complicated and detailed simulator in the distribution, by far, is sim-outorder. This simulator supports out-of-order issue and execution and gives the exact timing measurements.

As given above, Simple-Scalar simulator can simulate x86 instruction sets. We can pass configuration of Pentium-4 to Simple-Scalar so that it will give exact timing analysis for Pentium-4. Table B-1 gives the exact Pentium-IV processor's configuration.

Table B-1: Pentium 4 Processor Configuration

Instruction fetch queue size (in insts) -fetch: ifqsize	64
Extra branch mis-prediction latency -fetch: mplat	3
bimodal predictor BTB size -bpred:bimod	2048
2-level predictor config (<l1size> <l2size><hist_size>) -bpred:2lev	1 1024 8
Instruction decode B/W (insts/cycle) -decode:width	4
Instruction issue B/W (insts/cycle) -issue:width	4
run pipeline with in-order issue -issue: inorder	false
issue instructions down wrong execution paths -issue: wrongpath	true
register update unit (RUU) size -ruu: size	16
load/store queue (LSQ) size -lsq: size	8
l1 data cache config, i.e., {<config>|none} -cache: dl1	dl1:128:64:4:l
l1 data cache hit latency (in cycles) -cache: dl1lat	1
l2 data cache config, i.e., {<config>|none} -cache:dl2	ul2:16384:64:8:l
l2 data cache hit latency (in cycles) -cache: dl2lat	6
l1 inst cache config, i.e., {<config>|dl1|dl2|none} -cache:il1	il1:512:32:1:l
l1 instruction cache hit latency (in cycles) -cache:il1lat	2
l2 instruction cache hit latency (in cycles) -cache:il2lat	7
flush caches on system calls -cache:flush	false
convert 64-bit inst addresses to 32-bit inst equivalents -cache: compress	false
memory access latency (<first_chunk> <inter_chunk>) -mem:lat	18 2

Memory access bus width (in bytes) -mem: width	8
Instruction TLB config, i.e., {<config>|none} -tlb: itlb	dtlb: 16:4096:4: l
data TLB config, i.e., {<config>|none} -tlb: dtlb	dtlb: 16:4096:4: l
inst/data TLB miss latency (in cycles) -tlb: lat	30
total number of integer ALU's available -res:ialu	2
total number of integer multiplier/dividers available -res:imult	1
total number of floating point ALU's available -res:fpalu	1
number of floating point multiplier/dividers available -res:fpmult	1

APPENDIX C AMDAHL'S LAW

Amdahl's law is a model for the relationship between the expected speedup of parallelized implementations of an algorithm relative to the serial algorithm. It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors.

More technically, the law is concerned with the speedup achievable from an improvement to a computation that affects a proportion P of that computation where the improvement has a speedup of S. (For example, if an improvement can speed up 30% of the computation, P will be 0.3; if the improvement makes the portion affected twice as fast, S will be 2.) Amdahl's law states that the overall speedup of applying the improvement will be

$$Speedup = \frac{1}{(1-P)+\frac{P}{S}}$$

To see how this formula was derived, assume that the running time of the old computation was 1, for some unit of time. The running time of the new computation will be the length of time the unimproved fraction takes, (which is 1 − P), plus the length of time the improved fraction takes. The length of time for the improved part of the computation is the length of the improved part's former running time divided by the speedup, making the length of time of the improved part P/S. The final speedup is computed by dividing the old running time by the new running time, which is what the above formula does.

Similarly in case of parallelization, Amdahl's law states that if P is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and (1 − P) is the proportion that cannot be parallelized (remains serial), then the maximum speedup that can be achieved by using N processors is

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}}$$

In the limit, as N tends to infinity, the maximum speedup tends to 1 / (1-P).

As an example, if P is 90%, then (1 − P) is 10%, and the problem can be sped up by a maximum of a factor of 10, no matter how large the value of N used. For this reason, parallel computing is only useful for either small numbers of processors, or problems with very high values of P: so-called embarrassingly parallel problems. A great part of the craft of parallel programming consists of attempting to reduce (1-P) to the smallest possible value.