The Pennsylvania State University

The Graduate School

College of Engineering

A NEURAL NETWORK BASED CLASSIFIER ON THE CELL BROADBAND ENGINE

A Thesis in

Electrical Engineering

by

Srijith Rajamohan

© 2009 Srijith Rajamohan

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science

December 2009

The thesis of Srijith Rajamohan was reviewed and approved* by the following:

Suman Datta

Professor of Electrical Engineering

Thesis Co-Advisor

Vijaykrishnan Narayanan

Professor of Computer Science and Engineering

Thesis Co-advisor

Kenneth Jenkins

Professor of Electrical Engineering

Head of the Department of Electrical Engineering

*Signatures are on file in the Graduate School

Abstract

The Cell processor is a novel heterogeneous multicore architecture that holds the distinction of being part of the most powerful distributed computing system capable of sustaining over 4.5 Petaflops as well as in the RoadRunner supercomputer. This thesis hopes to meet the demands of the challenging task of face detection.

In this thesis a MLP neural network classifier is implemented on the multicore Cell architecture. Scalability of the code is addressed and every effort has been made to ensure that the code is not dependant on the structure of the network.

Table of Contents

List of Figures vii

Chapter 1 Introduction 1

Chapter 2 The Cell Broadband Engine Architecture 3

2.1 PowerPC Processor Element 4

2.2 Synergistic Processor Elements 5

2.3 Element Interconnect Bus 6

2.4 Neural Network Classifier 8

Chapter 3 Programming models for application partitioning 12

3.1.1 Function-Offload model 13

3.1.2 Device-Extension model 14

3.1.3 Computation acceleration model 14

3.1.4 Streaming model 14

3.1.5 Shared- Memory Model 15

3.1.6 Asymmetric-Thread Runtime Model 15

3.1.7 User mode Thread Model 16

3.2 Task Parallelism 16

3.2.1 Scalar intrinsics 17

3.2.2 Vector intrinsics 17

3.3 Multithreading and work scheduling on the SPE 18

3.4 Memory layout of the Cell architecture 19

3.4.1 SPU access to MFC 21

3.4.2 PPE access to MFC 22

3.4.3 DMA Access. . . . 23

3.4.4 Overlapped DMA and computation. . . . 26

3.5 Interprocessor Communication. . . . 26

3.5.1 Mailboxes. . . . 27

3.5.2 Signal notification registers. . . . 28

3.5.3 Atomic Caches. . . . 28

3.6 Shared storage synchronization. . . . 30

Chapter 4 Neural network and the Development environment 32

4.1 Introduction to the Mercury CAB 32

4.1.1 Neural network and the Development environment 33

4.1.2 Memory. . . . 34

4.2 Booting the CAB 34

4.2.1 Scaled Conjugate Gradient 39

4.3 Development environment for the Cell BE 40

4.3.1 Debugging applications for the Cell. . . . 41

4.3.2 Profile Checkpoints. . . . 45

Chapter 5 Cell Implementation of the classifier. . . . 47

5.1 Intra-frame and Inter-frame frameworks 47

5.2 Intra-frame Load partitioning models. . . . 48

5.3 Inter-frame SPE model. . . . 52

5.3.1 Independent SPE model. . . . 56

5.3.2 Memory Mapped Files. . . . 60

5.4 Hardware Implementation 63

Appendix 67

Bibliography. . . . 72

List of Figures

1. Architectural overview of the Cell Broadband Engine 3

2. PPE Block diagram. 4

3. SPE Block diagram. 5

4. Element Interconnect Bus. 7

5. Accelerator framework. 12

6. RPC Model 12

7. Streaming model 14

8. DMA access over the EIB. 19

9. CAB Overview 31

10. Overview of the network structure. 39

11. Systemsim main window 42

12. Intra-frame dataflow diagram. 48

13. Inter SPE synchronization protocol. 49

14. Intra-frame partitioning scheme 50

15. Performance Statistics from SystemSim. 51

16. Intra-frame timing diagram 52

17. Performance Statistics for an isolated SPE model 54

18. Independent SPE model 56

19. Bufferless DMA scheme 58

20. Double Buffered DMA 58

21. File partition. 58

22. Memory mapped file partition. 60

23. Performance cycle split-up. 63

24. Comparison of runtimes. 64

25. Cell vs X86. 65

Acknowledgments

I am indebted to Dr Vijay Narayanan and Dr Suman Datta for their advice and assistance; without their help, this thesis would not have been possible. I am grateful to all the members of the FPGA Camera Group at the Microprocessor Design Lab, Penn State. Finally, I must thank my family and friends, who have supported, encouraged and been patient with me over the course of this project.

Chapter 1

Introduction

The Cell Broadband Engine evolved in response to the need of Sony and Toshiba to come up with a high performing but cost effective architecture for video processing that can also cater to a wider range of applications. The Cell BE originally intended for media rich applications, has come to be useful to a wide range of high bandwidth applications in both the commercial and scientific fields. The CBEA has been designed. The CBE is single chip multiprocessor with nine processor that operate on a shared memory model. The goal of this thesis is to implement a scalable neural network classifier for human faces on the Cell platform. The hardware that we used is the Mercury Cell Accelerator Board which is a 16x PCIe accelerator card based on the Cell processor.

One of the drivers for such architecture is the demand for a higher level of performance per watt. According to an IDC study as referenced in Solutions for the Datacenter's Thermal Challenges[20] customers were looking for more computing power for a given space and electrical power budget. The design of the Cell architecture meets this requirement with a power efficiency that is over two times better than conventional general purpose processors. In addition to this the Cell BE has very high computational density with eight processing elements per system, each having a large register file and very low local storage latency.

The Cell BE processor has PPE and eight SPEs. The PPE contains a 64-bit PowerPC® Architecture™ core. The SPE is optimized for running compute-intensive single-instruction, multiple-data (SIMD) applications and is not optimized for running an operating system. The SPEs are independent processor elements, each running their own individual application programs or threads. The SPEs depend on the PPE to run control the flow of an application. The PPE depends on the SPEs to provide do the bulk of the processing.

The task at hand was to investigate the suitability of the Cell processor to a task such as a classifier that is scalable. The input data set is 38000 samples of 20x20 images, about 20000 of which are used for training the neural and the rest for classification. The network parameters are passed to the Cell which then uses the PPE to schedule and partition the load on the SPEs.

The SPEs are designed to be programmed in high-level languages, such as C/C++ and they support a rich instruction set that includes extensive SIMD functionality. PPE also supports the standard PowerPC Architecture instructions and the vector/SIMD multimedia extensions. This specialization is a significant factor accounting for the order-of-magnitude improvement in peak computational performance that the Cell/B.E. processor achieves over conventional PC processors.

PPE is rather like a conventional processor with regards to memory accesses. The PPE accesses main storage with load and store instructions that move data between main storage and a private register file, the contents of which may be cached. The SPEs access main storage with DMA commands that move data and instructions between main storage and a private local memory, called local storage (LS). An instruction-fetches and load and store instructions of an SPE access its private LS rather than shared main storage. SPEs have a three level organization of storage – register file, local store and main store.

Chapter 2

The Cell Broadband Engine Architecture

One of the motivations for the Cell architecture is the fact that memory latency has gone up several hundredfold and application performance is, in most cases, limited by memory latency rather than by peak compute capability or peak bandwidth. When a sequential program on a conventional architecture performs a load instruction that misses in the caches, program execution can come to a halt for several hundred cycles. Compare this with the few cycles it takes to set up a DMA transfer for an SPE, and it does not so bad after all. The explicit DMA model allows each SPE to have many concurrent memory accesses in flight, thereby avoiding the expensive speculation in conventional processors.

Conventional processors have vector units on board (SSE or VMX / AltiVec) however they are not dedicated vector processors. The SPEs are dedicated vector processors with their own memory; add 8 of them and you can see why their computational capacity is enormous.

Overview of the Architecture

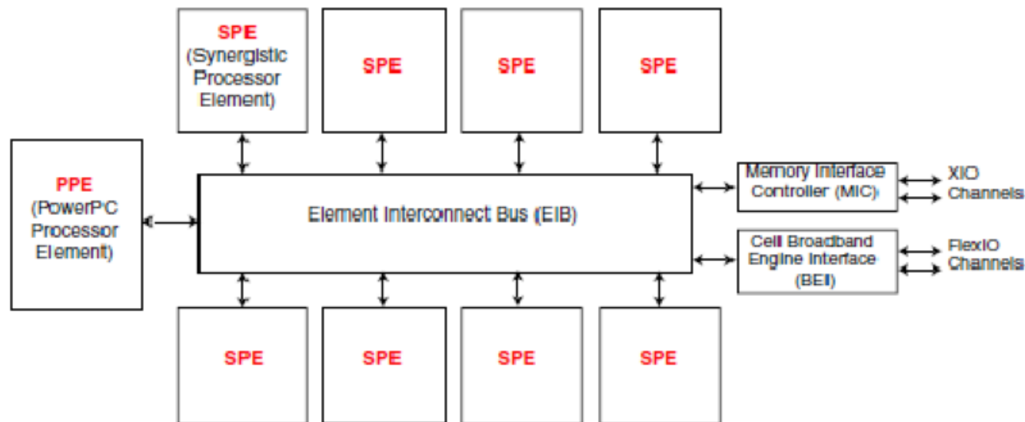


Fig 2.1: Architectural overview of the Cell Broadband Engine [2]

The Cell BE consists of nine processors on a single chip and it is connected to each other and to other external devices by a high bandwidth memory coherent bus. The main components are the Power Processor Element, Synergistic Processor Element and the Element Interconnect Bus

2.1 PowerPC Processor Element

The PPE is the main processor and it is a 64-bit PowerPC Architecture reduced instruction set computer (RISC) core. It can run an operating system and manage system resources for the SPE. It 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.

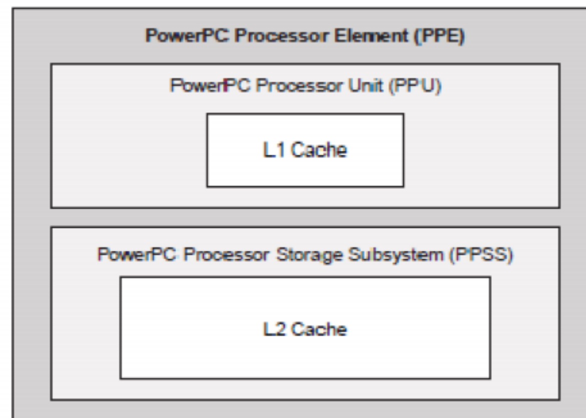


Fig 2.2: PPE Block diagram [2]

The PPE consists of two main units, the Power Processor Unit (PPU) and the Power Processor Storage Subsystem (PPSS). PPE has 64-bit PowerPC registers, 32 128-bit vector multimedia registers, a 32-KB level 1 (L1) instruction cache, a 32-KB level 1 (L1) data cache, an instruction-control unit, a load and store unit, a fixed-point integer unit, a floating-point unit, a vector unit, a branch unit, and a virtual-memory management unit. The PPSS handles all memory requests for the PPE from other processors or I/O devices. It has a unified 512-KB level 2 (L2) instruction and data cache, queues, and a bus interface unit that handles arbitration on the EIB. Memory is

seen as a linear array of bytes indexed from 0 to 264 - 1. Each byte is identified by its index, called an address and one storage access occurs at a time, with all accesses occuring in program order.

2.2 Synergistic Processor Elements

The eight SPEs are SIMD processors optimized for data-rich operations. It 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 (the effective-address space that includes main memory) and their local stores. SPE DMA transfers access main storage using PowerPC effective addresses and address translation is governed by PowerPC Architecture segment and page tables. The SPEs are not intended to run an operating system.

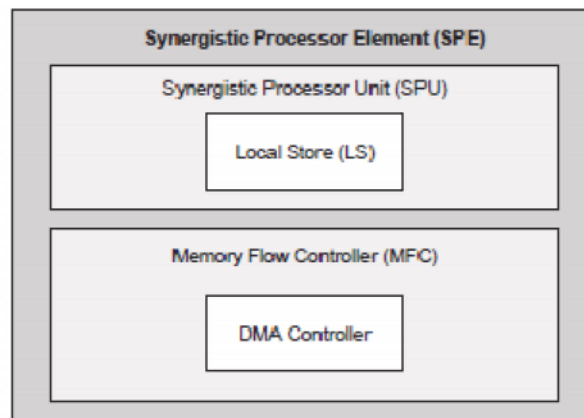


Fig 2.3: SPE Block diagram [2]

It includes a single register file with 128 registers (each one 128 bits wide), a unified (instructions and data) 256-KB local store (LS), an instruction-control unit, a load and store unit, two fixed-point units, a floating-point unit, and a DMA interface. Each SPU is an independent

processor with its own program counter and runs SPE threads spawned by the PPE. The SPU fetches instructions and loads and stores data from and to its own LS. The MFC contains a DMA controller that supports DMA transfers. DMA transfers are used to move instructions and data between the SPU's LS and main storage where main storage is the effective-address space that includes main memory, other SPEs' LS, and memory-mapped registers such as memory-mapped I/O registers. The MFC interfaces the SPU to the EIB and synchronizes operations between the SPU and all other processors in the system. It maintains and processes queues of DMA commands. After a DMA command has been queued to the MFC, the SPU can continue to execute instructions while the MFC processes the DMA command autonomously and asynchronously.

Each DMA transfer can be up to 16 KB in size. Only the MFC's associated SPU can issue DMA-list commands and there can be upto 2,048 DMA transfers, each one up to 16 KB in size. DMA transfers are coherent with respect to main storage. The SPEs provide a deterministic operating environment. They do not have caches, and hence cache misses are not a factor in their performance. Pipeline-scheduling rules are simple, so it is easy to statically determine the performance of code.

2.3 Element Interconnect Bus

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

Although the theoretical peak of the EIB is 96 bytes per cycle (384 Gigabytes per second) however, according to IBM [21] only about two thirds of this is likely to be achieved in practice. The EIB provides the Cell BE with an aggregate main memory bandwidth (at 3.2GHz) of about 25.6GB/s, I/O bandwidth of 35GB/s inbound and another 40GB/s outbound.

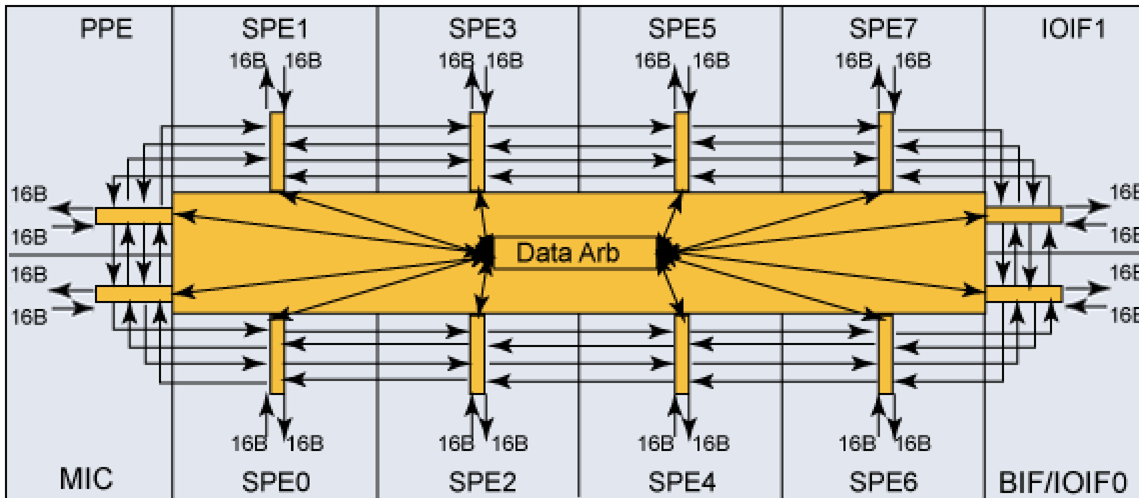


Fig 2.4: Element Interconnect Bus [2]

Within the four rings data can move down a ring only in one direction. For instance, a connection that allows data to move from the PPE to SPE1 cannot be used to move data from SPE1 back to the PPE. Two rings go clockwise, and two counterclockwise and all four rings have the components attached in the same order. Each ring can move 16 bytes at a time from any position on the ring to any other position. In fact, each ring can transmit three concurrent transfers, but those transfers cannot overlap. Data ports are not exposed to the ring interface, which is transparent to them. It would help to compare the Cells external memory bandwidth of 25.6 GB with that of a typical DDR2 memory bus which is about 6 – 11 GB/s.

The reason why the theoretical maximum bandwidth of an EIB cannot be attained is due to the fact that the 4 rings are a shared resource. While it is possible to have multiple transactions on a single ring, these transactions cannot be overlapped. If a given ring is being used for SPE to SPE communications, this might block communication from the PPE to the IOIF units, depending on the SPEs involved. Physical locality plays a significant role here and it is important to ensure that tasks SPE to SPE communication does not tie up an entire ring.

2.4 Neural Network Classifier

A Neural Network can be used to model any relationship between the predictor variables (inputs) and the predicted variables (outputs) even when that relationship is extremely complex. Key areas, apart from image classification, to which neural networks have been successfully applied, are

- Detection of medical phenomena. The onset of a particular medical condition could depend on a very complex (e.g., nonlinear and interactive) combination of changes on a subset of a number of variables. Neural networks have been used to recognize this predictive pattern so that the appropriate treatment can be prescribed. Cancer detection is a prime example where the network factors in a person's medical history as well.
- Stock market prediction. Stock prices and stock indices are another example of a complex, multidimensional, but in some circumstances at least partially-deterministic phenomenon. Neural networks are currently used by many technical analysts to make predictions about stock prices based upon a large number of factors such as past performance of other stocks and various economic and social change indicators.
- Credit assignment. After training a neural network on historical data like age, occupation, education neural network analysis can identify the most relevant characteristics and use those to classify applicants as good or bad credit risks.
- Monitoring the condition of machinery. Neural networks have been used to schedule the preventive maintenance of machines. A neural network can be trained to distinguish between the sounds a machine makes when it is running normally versus when it is on the verge of a problem. After this training period, the expertise of the network can be used to warn a technician of an upcoming breakdown, before it occurs.
- Engine management. Neural networks have been used to analyze the input of sensors from an engine. The neural network controls the various parameters within which the engine functions, in order to achieve a particular goal, such as minimizing fuel consumption.

Not every problem can be modeled using a neural network, there must exist a relationship (at least a weak one). Not factoring in certain predictor variables (which is quite probable) can lead to distorted or at least noisy results. Generally the relationship is unknown, for if it were known it would have been modeled randomly. There are two types of training used in neural networks namely supervised and unsupervised learning. In our implementation we have chosen the supervised learning approach.

In supervised learning, user assembles a set of training data. The training data contains examples of inputs together with the corresponding outputs, and the network learns to infer the relationship between the two. The neural network is then trained using one of the supervised learning algorithms which use the data to adjust the network's weights and thresholds so as to minimize the error in its predictions on the training set. If the network is properly trained, it will have learned to model the function that relates the input variables to the output variables, and can subsequently be used to make predictions where the output is not known.

Face detection has application in a wide variety of fields such as security systems, identification, tracking systems etc. It must be noted that face detection is different from face recognition. Face detection is often the predecessor to recognition, it is widely accepted that once the presence of a face has been detected the related problem of recognition becomes a far easier one. This is one of the fundamental steps in applications like Video surveillance, facial expression extraction, gender classification, Human computer interaction. Some recent cameras also use face recognition for autofocus. Recognition can be performed using a number of methods namely neural networks, feature extraction, Markov chain, template matching etc. Being a resource intensive task alternate platforms of computation are necessary for a real time implementation. GPUs and Cell Processors are ideally suited to the problem along with hardware implementations on FPGAs.

The Cell Broadband engine would be suited to the computations that are needed of a neural network however scalability needs to be addressed. Dynamically partitioning load always results in a less efficient implementation compared to a customized algorithm. Most of the

classifier implementations are customized versions which also allows for optimizations which would otherwise be impossible to make with a generic algorithm.

2.4.1 Least Mean Square Error

The least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior:

$$\{\mathbf{p}_1,\mathbf{t}_1\}, \{\mathbf{p}_2,\mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q,\mathbf{t}_Q\}$$

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k) - a(k))^2$$

Here P_q is an input to the network, and t_q is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and the network output. The goal is to minimize the average of the sum of these errors.

The LMS algorithm adjusts the weights and biases of the linear network so as to minimize this mean square error.

2.4.2 Scaled Conjugate Algorithm

Most training algorithms like gradient descent and gradient descent with momentum are often too slow for practical purposes. The basic backpropagation algorithm adjusts the weights in the steepest descent direction, the direction in which the performance function is decreasing most rapidly. It turns out that, although the function decreases most rapidly along the negative of the gradient, this does not necessarily produce the fastest convergence. In the conjugate gradient

algorithms a search is performed along conjugate directions, which produces generally faster convergence than steepest descent directions.

In most of the training algorithms discussed up to this point, a learning rate is used to determine the length of the weight update (step size). In most of the conjugate gradient algorithms, the step size is adjusted at each iteration. A search is made along the conjugate gradient direction to determine the step size that minimizes the performance function along that line. There are five different search functions included in the toolbox, and these are discussed in Line Search Routines. Any of these search functions can be used interchangeably with a variety of the training functions described in the remainder of this chapter. Some search functions are best suited to certain training functions, although the optimum choice can vary according to the specific application.

Most of the conjugate gradient algorithms require a line search at each iteration. This line search is computationally expensive, because it requires that the network response to all training inputs be computed several times for each search. The scaled conjugate gradient algorithm (SCG), developed by Moller was designed to avoid the time-consuming line search.

Chapter 3

Programming models for application partitioning

To achieve optimum performance, optimizations need to made to an implementation which fully exploit the underlying architecture. The distinguishing features of the Cell architecture are the large number of SPEs, independent local storage, vector processing, DMA model for data transfer. Main memory latency can be hidden by overlapping data transfers with computation. Programmers here have the advantage of making use of its features instead of having to rely solely on the compiler to do all the work, which although might result in greater turnaround time results in more optimized, efficient algorithms. However in order to choose an optimum model the following factors that are specific to the Cell must be kept in mind [1]

- Heterogeneous units – PPE/SPE
- Distributed memory between the SPE
- SIMD processors
- PPE slow for compute intensive tasks
- Software managed memory hierarchy
- Limited LS size
- Dynamic code loading
- High EIB bandwidth
- Coherent shared memory
- Large SPE context startup time

The general framework of an accelerator looks as in Fig 3.1

Live data	READ ONLY	READ WRITE		WRITE ONLY
State data	f1state	f2state	f3state	f4state
Services	f1()	f2()	f3()	f4()

Fig 3.1: Accelerator framework [1]

The state data is that which is persistent across multiple invocations and live data is the data that moves in and out of the accelerator for each invocation. It is necessary to understand and analyze these components to optimize the algorithm.

3.1.1 Function-Offload model

In the Function-Offload Model, the SPEs are used as accelerators for critical procedures. Specific functions are moved onto the SPE that are then called from the PPE. Minimal code change restructure is required and this model is sometimes called the RPC Model. This provides a way for programmers to use the asynchronous parallelism of the SPEs without dealing with the low-level workings of the DMA layer. This model uses proxy code called stub that the PPE calls. The stub then initializes the SPE and transfers parameters to it.

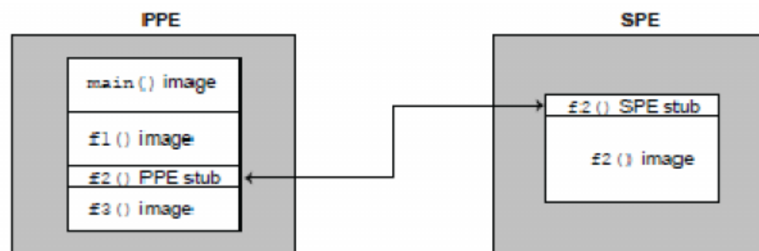


Fig 3.2: RPC Model [1]

This is a fork/join model, and therefore enough work needs to be given to an SPE to compensate for the startup time overhead. This is typically implemented with specialized workload libraries such as BLAS and FFT. One variation of this model is to have an accelerator program running on the SPE, sitting in a loop, awaiting requests from PPE to provide services. The existence of persistent threads on each SPE eliminates the startup time of SPE threads. The ALF (Accelerated Library Framework) framework uses such an implementation.

3.1.2 Device-Extension model

The Device Extension Model can be considered as a special case of the Function-Offload Model in which the SPEs act like I/O devices. Mailboxes can be used as command and response FIFOs between the PPE and SPEs. The SPEs can interact with I/O devices because all I/O devices are memory-mapped.

3.1.3 Computation acceleration model

In this model the work is partitioned manually by the programmer to have the most computationally intensive tasks in an algorithm performed by the SPE. The SPEs have to efficiently schedule DMA commands to hide memory latencies. This model either uses shared memory model among SPEs, or a message-passing model. One of the advantages is that this does not require a significant rewrite of the application.

3.1.4 Streaming Model

In a streaming model data streams through SPEs which are set up as either a serial or parallel pipeline. The SPEs act as stream data processors with each processor performing a dedicated task and passing data onto the next one. This model can be quite efficient because the data

remains inside the Cell for as long as possible. The PPE and SPEs support message-passing among them.

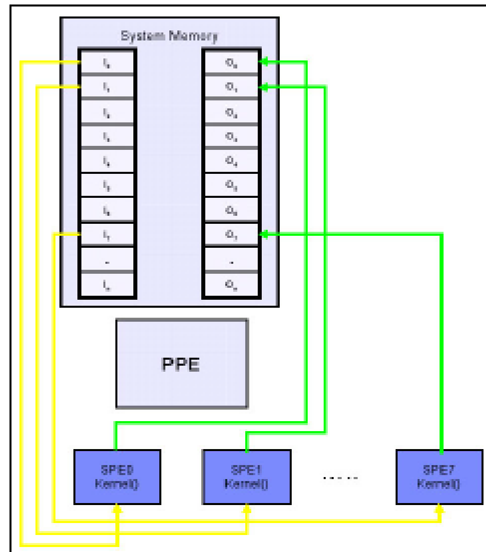


Fig 3.3: Streaming model [1]

3.1.5 Shared- Memory Model

The SPEs and the PPE operate in a cache-coherent Shared-Memory Multiprocessor Model. All DMA operations in the SPEs are cache-coherent. DMA operations are used for data transfer from shared memory to local store (LS), followed by a load from LS to the register file. The DMA operations use an effective addressing mechanism that is common to the PPE and all the SPEs. The SPE's DMA lock-line commands provide the equivalent of the PowerPC Architecture atomic update primitives

3.1.6 Asymmetric-Thread Runtime Model

Threads can be scheduled to run asymmetrically with the thread interacting with each other like in a conventional multiprocessor. Scheduling policies are applied to the PPE and SPE

threads to optimize performance. A single SPE can run only one thread at a time; it cannot support multiple simultaneous threads. This model is flexible and supports all of the other programming models described above and any thread created with spe_create_thread (). This is one of the fundamental models supported by the SPU Runtime Management Library.

3.1.7 User mode Thread Model

In this model one SPE thread manages a set of functions running in parallel. These user level functions are called microthreads, which are created and supported by the user software. However a set of microthreads can run across a number of different SPUs. The application schedules the tasks in shared memory and the tasks are processed by available SPUs. One of the advantages of this model is that since everything happens under the control of SPUs there is predictable overhead.

3.2 Task Parallelism

From a programmers perspective the SPU threads can be considered as Linux threads. The PPEs conform to the PowerPC architecture, hence most programs that run on a Linux based PowerPC architecture should work just fine on the PP which includes file system, using sockets and message passing interface (MPI) for communication with remote nodes, and managing memory allocation. Any use of operating system functionality needs to be implemented on the PPE , any such calls made from the SPE causes the stalling of SPE code till the PPE handles that request on behalf of the SPE and finishes it.

Coding for the Cell BE is done in a high level language and for the purpose of this thesis I have chosen C. C language intrinsic are provided for writing efficient highly optimized code. Intrinsics are essentially inline assembly language instructions in the form of function calls, however with the advantage of having a syntax that is already familiar to programmers. There are 2 kinds of intrinsic depending on the nature of the data that they work with.

3.2.1 Scalar intrinsics

These are available on the PPU to make the PPU instruction set easily accessible to the programmer from the C language. Almost every one of them have a one-to-one assembly language mapping. Some of the most useful intrinsics are related to shared memoy access and synchronization as they can assist in significantly improving code performance. Some of them provide access to the PPE internal registers and all the intrinsics are declared in the header file ppu_intrinsics.h

3.2.2 Vector intrinsics

The vector data types intrinsics supports the VMX instructions, which follows the AltiVec standard. The VMX provides a set of fundamental data types, called vector types. The vector registers are 128 bits and can have either sixteen 8-bit values (signed or unsigned), eight 16-bit values (signed or unsigned), four 32-bit values (signed or unsigned), or four single-precision IEEE-754 floating-point values. The vector instructions provide for a rich set of operations that can be performed on those 128 bit operands which include arithmetic operations, rounding and conversion, floating-point estimate intrinsics, compare intrinsics, logical intrinsics, rotate and shift Intrinsics, load and store intrinsics, and pack and unpack intrinsics. It must be noted that vector operations are usually performed on the SPUs because of the suitability of the SPU architecture. The Vector/SIMD intrinsics contains various basic mathematical functions that are implemented by corresponding SIMD assembly instructions, more complex mathematical functions are not supported by these intrinsics. The SIMD Math Library is provided with the Cell SDK and they include a set of functions that extend the SIMD intrinsics and support additional mathematical functions.

3.3 Multithreading and work scheduling on the SPE

In any program, the main thread is run on the PPE which creates sub-threads that run on the SPEs and off-loads and allocates work to be run on the SPEs. The programming model determines how the threads and tasks are managed, the data is transferred, and how the different processors communicate. The code running on the SPEs use the libspe library (SPE runtime management library) that is part of the SDK package. This library provides a low-level application API for DMA, register and memory access, synchronization etc. When the PPE creates a thread using a function such as spe_create_thread() it creates what is called a context. These SPE contexts are a logical representation of an SPE and hold all persistent information about a SPE. It must be noted that exiting the thread destroys all such information, in effect reinitializing the SPE everytime spe_create_thread() is called. Rescheduling an SPE and performing the context switching takes time because you must store most of the 256 KB of the local storage (LS) in memory and reload it with the code and data for the new thread.

When creating an SPE thread one can pass up to three parameters to this function. The parameters may be either 64-bit parameters or 128-bit vectors. These parameters can be used by the code that is running on the SPE. It is a common practice to pass the effective address of a block of data that might be larger and contains additional information. The SPE can use this address to DMA this block into its local storage memory. There are two ways to load SPE programs, either statically or it can be loaded during runtime dynamically. When the program is loaded statically, it is compiled with the PPE program and at run time the object is accessed using an external pointer is used to load the program into local storage. The loading of the program is implemented internally by the library API by using DMA. When the SPE program is loaded dynamically you compile the SPE as a stand-alone application. During run time the executable is mapped into the main memory, and then loaded into the local storage of theSPE. This method provides flexibility because you can decide, at run time, which program to load depending on the run time parameters. The disadvantage of this method, however is that the program is now a set of files, and not just a single executable.

3.4 Memory layout of the Cell architecture

The Cell utilizes a unique memory layout quite different from conventional processors. The MFC implements most of the Cells inter-processor communication mechanism. It includes the means to initiate data transfer, including DMA data transfers. The MFC interfaces can be accessed by both code on the SPE and PPE. The Cell Broadband Engine Architecture defines three types of storage domains: one main-storage domain, eight SPE LS domains, and eight SPE channel domains. The main-storage domain or the entire effective address space, can be configured by the PPE operating system to be shared by all processors in the system. The local-storage and channel problem-state domains are private to the SPE components. The SPE consists of the SPU, the LS and the MFC which handles the DMA data transfer. The LS of an SPE program is referenced by the Local Store Address, the LS can also be mapped within the systems memory using a real address. Privileged software on the PPE can be configured to access the EA space, where the PPE, other SPEs, and other devices that generate EAs can access the LS like any regular component on the main storage. Any transfer between the LS and main storage of the SPE are executed by using DMA transfers that are controlled by the MFC DMA controller for that SPE. The MFC of each SPE serves as a data-transfer engine. DMA transfer requests contain both an LSA and an EA. Therefore, they can address both the LS and main storage of an SPE and to initiate DMA transfers between the domains. The local storage can be mapped to the main storage; SPEs can use DMA operations to directly transfer data between their LS to the LS of another SPE. The DMA transfers go directly from SPE to SPE on the high performance local bus.

MFC commands can be initiated through either a Channel interface or MMIO interface. The SPU can use this interface to interact with the associated MFC through writes or reads to the various channels, which in response enqueue MFC commands. It has low latency since channel accesses are local to the SPE, this does not use the EIB bandwidth. MMIO interface can be used by PPE or other to interact with any MFC by accessing the Command-Parameter Registers of the MFC. These registers can be mapped to the system's real-address space so that the PPE or SPUs can access them using their effective addresses.

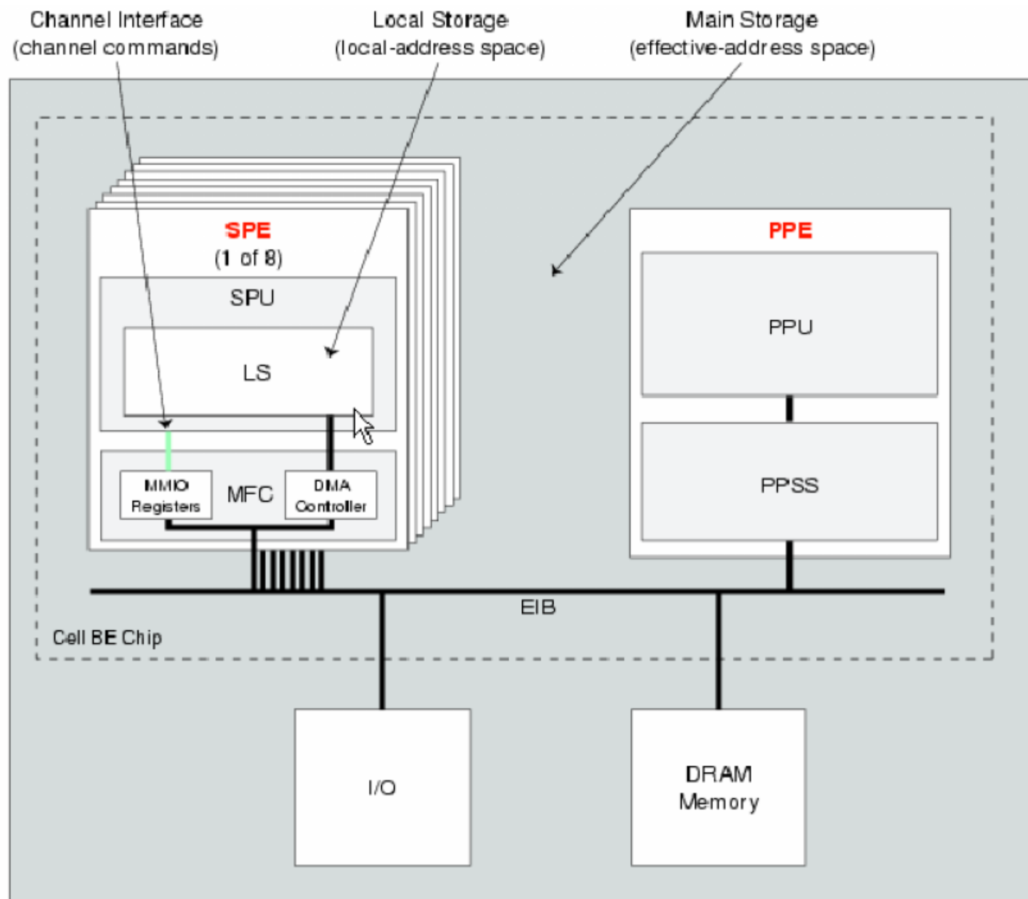


Fig 3.4: DMA access over the EIB [2]

Additionally each channel can be configured to be blocking or nonblocking. When the SPE reads or writes a nonblocking channel, it is immediately executed. However, when the SPE reads orwrites a blocking channel, the SPE might stall for an arbitrary length of time if theassociated channel count is 0. In case of a channel read the SPE stalls until there is something to read and in case of a write it stalls till space is available to write to it. The stalling mechanism reduces SPE software complexity and allows an SPE to minimize the power consumed by message-based synchronization. To avoid stalling SPE software must read the channel count to determine the available channel capacity. The MMIO interface is always nonblocking. If a command is written while the queue is full, then the last entry in the queue is overridden. Therefore care should be taken to first verify if there is available space in the queue by reading the queue status register. Waiting for available space by continuously reading this

register in a loop has a negative effect on the performance of the entire chip because it this happens over the local EIB. Reading from an MMIO register when a queue is empty returns invalid data. Therefore the corresponding status register should be read first and only if there is a valid entry the MMIO register itself should be read.

3.4.1 SPU access to MFC

There are number of ways in which the SPU can access the MFC.

1. MFC Functions
2. Composite intrinsics
3. Low-level intrinsics
4. Assembly language intrinsics

Although the simplest method for a programmer would be to use the MFC, from a performance viewpoint it is not desired. The DMA command to transfer data into the SPE mfc_get() has the following syntax

```
mfc_get(lsa, ea, size, tag, tid, rid);
```

where lsa is local-storage address, ea is the effective address, size is the DMA transfer size in bytes, tag is DMA tag group id, tid is transfer class identifier and rid is replacement class identifier.

Each composite intrinsic handles one DMA command and is constructed from a series of low-level intrinsic. Example of the a composite intrinsic

```
spu_mfcdma64(lsa, eah, eal, size, tag, MFC_GET_CMD);
```

A series of a few low-level intrinsics should be executed in order to run a single DMA transfer. Each intrinsic is mapped to a single assembly instruction.The same get command can be performed using low level intrinsic as

```
spu_writech(MFC_LSA, lsa);
```

```
spu_writech(MFC_EAH, eah);
spu_writech(MFC_EAL, eal);
spu_writech(MFC_Size, size);
spu_writech(MFC_TagID, tag);
spu_writech(MFC_CMD, 0x0040);
```

Assembly-language instructions are similar to low-level intrinsics. In fact as mentioned above there is a one-to-one mapping to low level intrinsic. The same get command in assembly is

```
.text
.global dma_transfer
dma_transfer:
wrch$MFC_LSA, $3
wrch$MFC_EAH, $4
wrch $MFC_EAL, $5
wrch $MFC_Size, $6
wrch $MFC_TagID, $7
wrch $MFC_Cmd, $8
bi $0
```

3.4.2 PPE access to MFC

PPE can access the MFC facilities through the MMIO interface. This can be done either through MFC functions or Direct problem state access (or direct SPE access). However for PPE access MFC functions are not recommended. MFC functions are simpler for a programmer however Direct problem state access gives the programmer more flexibility. The syntax for a get command is similar to the one for SPE. However the tag ID that is used for the PPE-initiated DMA transfer is not related to the tag ID that is used by the software that runs on this SPE. Eachtag is related to a different queue of the MFC. There are no mechanisms available for

allocating tag IDs on the PPE side, such as the SPE tag manager. Therefore, the programmer should use a predefined tag ID. Since tag IDs 16 to 31 are reserved for the Linux kernel, the user should stick to tag IDs 0 to 15 .Direct problem state access has significantly better performance in cases such as when writing to the inbound mailbox. One reason for the reduction in performance for the MFC functions is the call overhead. Therefore in cases where the performance, for example latency, of the PPE access to the MFC is important, Direct problem state access is preferred, which may have significantly better performance over the MFC functions.

In order to use the Direct problem state access one has to map the corresponding problem state area of the relevant SPE to the PPE thread address space. The programmer can do this by using the spe_ps_area_get function in the libspe library. Then use one of the inline functions for direct problem state access that are defined in the cbe_mfc.h header file. Direct memory load or store instructions can also be used to access the relevant MMIO registers. After the problem state area is mapped, direct access to this area by the application does not involve the kernel, and therefore, has a smaller latency than the corresponding MFC function. The PPE program must set the SPE_MAP_PS flag when creating the SPE context (in the spe_ create_thread function) of the SPE whose problem state area is to be mapped using the spe_ps_area_get function.

3.4.3 DMA Access

MFC supports a set of DMA commands that provide the main mechanism that enables data transfer between the LS and main storage. It also supports a set of synchronization commands that are used to control the order in which storage accesses are performed and maintain synchronization with other processors. Each MFC has an associated memory management unit that holds and processes address-translation and access-permission information that is supplied by the PPE operating system. This MMU is distinct from the one used by the PPE, but in order to process an effective address provided by a DMA command, the MMU uses the same method as the PPE memory-management functions.

MFC supports a set of DMA commands which can initiate or monitor the status of data transfers. Each MFC can maintain and process up to 16 in-progress DMA command requests execution. This happens separately from code execution. The MFC can autonomously manage a sequence of DMA transfers in response to a DMA list command from its associated SPU. DMA lists are a sequence of eight-byte list elements, stored in the LS of an SPE, each of which describes a single DMA transfer. The DMA command is tagged with a 5-bit Tag ID and the software can use this identifier to check or wait on the completion of all queued commands in one or more tag groups.

There are a number of guidelines for DMA commands that must be followed. The supported transfer sizes are 1, 2, 4, 8, or 16 bytes, and multiples of 16-bytes. The maximum transfer size is 16 KB. The peak performance is achieved when the transfer size is a multiple of 128 bytes. The source and destination addresses should have the same four least significant bits. When transfer size is less than 16 bytes, the address must be naturally aligned. For transfer sizes of 16 bytes or greater, the address must be aligned to at least a 16-byte boundary. Bits 28 through 31 must be 0. Peak performance is achieved when both the source and destination are aligned on a 128-byte boundary. Bits 25 through 31 must be cleared to 0. Address and alignment violations are not noted during compilation however during runtime the program terminates with a "Bus error". The MFC checks the validity of the effective address during the transfers.

After a DMA is initiated the software might choose to wait for completion of the command. The mfc_write_tag_mask function writes the tag mask that determines to which tag IDs a completion notification is needed. The mfc_read_tag_status_any function waits until any of the specified tagged DMA commands is completed. The mfc_read_tag_status_all function waits until all of the specified tagged DMA commands are completed.

A DMA list is a sequence of transfer elements that, together with an initiating DMA list command, specify a sequence of DMA transfers between a single continuous area of the LS and possibly discontinuous areas in the main storage. DMA lists can be used to gather data from disparate locations in main memory and group it in LS. Data transfers that are issued in a single DMA list command have the same tag ID. The DMA list is stored in the LS of the same SPE.

After the list is stored in the LS, the execution of the list is initiated by a DMA list command, such as getl or putl. To initialize a DMA list transfer, the SPE program can call one of the corresponding functions of the spu_mfcio.h header file. These functions are nonblocking in terms of issuing the DMA command. The software continues its execution after enqueueing the commands into the MFC SPU command queue but does not block until the DMA commands are issued on the EIB.

DMA programming can be organized by using multiple different tags. The tag is an integer between 0 and 31. The use of tags reduces the grain size of communication compared to DMA transfers always using the same tag. When checking the completion of some transfer with some given tag, we can only look at the state of all pending transfers using this given tag. Each DMA engine has its own independent tag set. Two DMA engines could issue a DMA request with the same tag concurrently. Suppose if both the PPU and SPU uses the same tag, if either the PPU or the SPU checks the completion of this tag, the completion of its of own DMA request is checked, and not both requests. One example of a DMA request command is mfc_get(). An mfc_get() takes the following parameters

- Volatile void *ls - LS address of data to be transferred
- Unsigned long long ea – Effective address of the external data location
- Unsigned int size – Number of bytes to be transferred
- Unsigned int tag – Value identifying a DMA request, it can be used to group DMAs
- Unsigned int tid - Transfer class identifier
- Unsigned int rid - Replacement class identifier

The first 3 parameters are self explanatory and tags were discussed above. Tid affects how the EIB assigns bandwidth to data transfers and rid influences the L2 caches replacement scheme. However utilizing these requires privileged access to internal device registers.

The SPU program can access the LS of another SPE in the chip. The LS is mapped to the effective address in the main storage that allows SPEs to use ordinary DMA operations to transfer data to and from this LS. LS-to-LS data transfer is efficient because it goes directly from SPE to SPE on

the internal EIB without involving the main memory interface. The internal bus has a much higher bandwidth than the memory interface and lower latency.

3.4.4 Overlapped DMA and computation

Overlapping DMA and computation can dramatically improve program performance. Consider the following scenario, where the program accesses incoming data using DMA from the main storage to the LS. The program waits for the transfer to complete. Once transfer is complete data in buffer B is processed. It is obvious why this sequence is not efficient because it wastes a lot of time waiting for the completion of the DMA transfer and has no overlap between the data transfer and the computation.

We can significantly accelerate the previous process by allocating two buffers, and overlapping computation on one buffer with data transfer in the other. This technique is called double buffering [1] . Double buffering is kind of multi-buffering, which uses multiple buffers in a circular queue instead of only the two buffers of double buffering. In most cases, usage of the two buffers in the double buffering case is enough to guarantee overlapping between the computation and data transfer. However, if the software must wait for completion of the data transfer, more buffers might be allocated.

3.5 Interprocessor Communication

The Cell/B.E. has a number of mechanisms for communication between the PPE and SPEs and between the SPEs. These mechanisms are mainly implemented by the MFCs. Mailboxes and signals are mechanisms that can be used for sending short messages between the different processors. While they have a lot in common, there are differences between the two mechanisms. In general, a mailbox implements a queue for sending separate 32-bit messages, while signaling is similar to interrupts, which can be accumulated when being written and reset when being read.

3.5.1 Mailboxes

The mailbox mechanism sends 32-bit messages between the local SPU and the PPE or local SPU and other SPEs. The mailboxes are accessed from the local SPU by using the channel interface and from the PPE or other SPEs by using the MMIO interface. Mailboxes can also be used as a communications mechanism between SPEs. This is done by DMAing data into the mailbox of another SPE by using the effective addressed problem state mapping. Local SPU access to the mailbox is internal to an SPE and has small latency. However PPE or other SPE access to the mailbox is done through the local memory EIB which has larger latency and overloading of the bus bandwidth, especially when polling to wait for the mailbox to become available. Each SPE contains three mailboxes divided into two categories, Outbound mailboxes and Inbound mailboxes. Two mailboxes are used to send messages from the local SPE to the PPE or other SPEs, the SPU Write Outbound mailbox (SPU_WrOutMbox) and SPU Write Outbound Interrupt mailbox (SPU_WrOutIntrMbox). One mailbox, SPU Read Inbound mailbox (SPU_RdInMbox), is used to send messages to the local SPE from the PPE or other SPEs.

When new data is written when the buffer is full overruns the last entry in the FIFO. The SPU program that writes new data when the buffer is full blocks until space is available in the buffer. The SPU program blocks when trying to read an empty buffer and continues only when there is a valid entry. The PPU program never blocks, writing to the mailbox when full overrides the last entry, and the PPU immediately continues. The SPU program blocks when trying to write to the buffer when it is full and continues only when there is an empty entry. Reading from the mailbox when it is empty returns invalid data, and the PPU program continues.

In order to overcome stalling the program on a blocking read the programmer can explicitly read the mailbox status by calling the *_stat_* functions for the SPU program. It must be noted that Local SPU access is internal to the SPE and has small latency. PPE or other SPE access is done through the local memory EIB, they tend to tie up the bus ans hence use up bus bandwidth, especially when polling the mailbox counter.

3.5.2 Signal notification registers

It enables a PPU program to signal an SPE by using 32-bit registers. It also enables an SPU program to signal a program that is running on another SPU by using the signal mechanism of the other SPU. Each SPE contains two identical signal notification registers: Signal Notification 1 (SPU_RdSigNotify1) and Signal Notification 2 (SPU_RdSigNotify2). Unlike mailboxes, the signal notification registers can only be used to send information to the SPUs. An SPU program signals another SPE by using special signaling command like sndsig, sndsigf, and sndsigb. When the local SPU program reads a signal notification the signal's register is reset to 0. When writing to the registers two different modes can be configured: OR mode (many-to-one) and Overwrite mode. In OR mode the MFC accumulates several writes to the signal-notification register by using a logical OR operation to combine all the values that are written to this register. The register is reset when the SPU reads it. In Overwrite mode writing a value to a signal-notification register overwrites the value in this register. This mode is similar to using an inbound mailbox and has similar performance. The signaling mode can be configured by the PPU when it creates the corresponding SPE context. Using the OR mode requires no synchronization between the SPEs as signal producers can send their signals at any time and independently of other signal producers.

Similar to the mailboxes, the signal notification register maintains a counter. The counter in this case behaves differently from that for mailboxes. The counter indicates only whether there are pending signals and not the number of writes to the register that have taken place. A value of 1 indicates that there is at least one event pending, and a value of 0 indicates that no signals are pending. The counter can be accessed from the local SPU, PPU, or other SPUs.

3.5.3 Atomic Caches

Atomic operations are implemented by the SPE using an atomic unit inside each MFC, which contains a dedicated local cache for cache-line reservations. This cache is called the atomic cache. The atomic cache has a total capacity of six 128-byte cache lines, of which four are

dedicated to atomic operations. When all the SPEs and the PPE perform atomic operations on a cache line with an identical effective address, a reservation is made in one of the MFC units. When this occurs, the cache snooping and update processes are performed by transferring the cache line contents to the requesting SPE or PPE over the EIB, without requiring a read or write to the main system memory. Hardware support is essential for efficient atomic operations on shared data structures that consist of up to 512 bytes, divided into four 128-byte blocks mapped on a 128-byte aligned data structure in the local storage of the SPEs.

In order to lock on a shared variable perform the reservation for the cache line designated to contain the part of the shared data structure that is to be updated by using mfc_getllar. This operation causes data transfer from the atomic unit that contains the most recent reservation or from the PPU cache to the requesting atomic unit of the SPE over the EIB. The data structure mapped in the SPU local storage now contains the most up-to-date values hence the values can be copied to a temporary buffer. You can update the cache line by using mfc_putllc but this can happen only if there is no lock on the said variable.

For PPU to update or access a shared variable perform the reservation for the cache line designated to contain the part of the shared data structure that is to be updated by using __lwarx or __ldarx. This operation triggers the data transfer from the atomic unit that contains themost recent reservation to the PPU cache over the EIB. The data structure that was contained in the specified effective address, which resides in the PPU cache, now contains the most current values. Hence you can copy the values to a temporary buffer and update the structure with modified values. Attempt the conditional update for the updated cache line by using __stwcx or __stdcx. To be noted is the difference between the behavior of the PPE and SPE in managing atomic operations. While both use the cache line size (128 bytes) as the reservation granularity, the PPU instructions operate on a maximum of 4 bytes (__lwarx and __stwcx) or 8 bytes (__ldarx and __stdcx) at once while the SPE atomic functions update the entire cache line contents.

3.6 Shared storage synchronization

Because the Cell BE uses DMA transfers for any movement of data, it can be considered to be following a weakly consistent storage model. Storage accesses can be reordered dynamically and hence there is the likelihood of real time bugs especially when code is ported from one system to another. Hence the programmer is burdened with the task of explicitly ordering access by using synchronization instructions whenever storage occurs in program order. Over usage of the synchronization instructions can significantly reduce the performance because they take a lot of time to complete [4].

To ensure that access to the shared storage is performed in program order, the software must place memory-barrier instructions between storage accesses. The term storage access refers to main storage that is caused by a load, a store, a DMA read, or a DMA write. There are two orders to consider, order of instructions execution and order of shared storage access. The Cell/B.E. processor is an in-order machine, which means that, from a programmer's point of view, the instructions are executed in the order specified in the program. The order in which shared-storage access is performed might be different from both the program order and the order in which the instructions that caused the access are executed.

PPU ordering instructions enable the code on the PPU to order the PPE data transfers on the main storage with respect to all other elements, such as other SPEs, in the system. The ordering of storage access and instruction execution can be explicitly controlled by the PPE program by using barrier instructions. These instructions can be used between storage-access instructions to define a memory barrier that divides the instructions into those that precede the barrier instruction and those that follow it.

One example of such a synchronization command is sync(). It is known as the heavyweight sync, and it ensures that all instructions that precede the sync have completed before the sync instruction completes and that no subsequent instructions are initiated until after the sync instruction completes. This does not mean that the previous storage accesses have completed before the sync instruction completes. Another command is the lwsync() or lightweight sync. It

creates the same barrier as the sync instruction for storage access that is memory coherence. Therefore, unlike the sync instruction, it orders only the PPE main storage access and has no effect on the main storage access of other processor elements.

SPU ordering instructions enable the code on the SPU to order SPU data access to the LS with respect to all other elements that can access it, such as the MFC and other elements in the system that access the LS through the MFC (for example PPE, other SPEs). They also synchronize the access to the MFC channels. An LS can have asynchronous interactions from a number of sources namely instruction fetches by the local SPU , data loads and stores by the local SPU, DMA transfers by the local MFC or the MFC of another SPE, loads and stores in the main-storage space by other processor elements. In order execution is guaranteed only with respect to that SPU, external accesses to that LS is not guaranteed in order execution.

An SPE might write data to the LS and immediately generate an MFC put command that reads this data. In this case without synchronization instructions, it is not guaranteed that the MFC will read the latest data, since it is not guaranteed that the MFC reading the data is performed after the SPU that writes the data. However executing the six commands for issuing the DMA always takes longer than executing the former write to the LS. In the absence of external LS writes, an SPU load from an address in its LS returns the data written by that most-recent store of the SPU to that address. However when instruction is fetched from that address, it may not guarantee the return of that recent data [4].

Chapter 4

Neural network and the Development environment

4.1 Introduction to the Mercury CAB

For this thesis we have used the Mercury Cell Accelerator Board which is a 16x PCIe accelerator card based on the Cell Broadband Engine™ (Cell BE) processor and intended for use in high-performance computing applications. The CAB has one Cell BE processor directly mounted on the board and 1 GB of XDR™ DRAM memory. A companion chip provides the high-speed bridge from the processor to a PCIe x16 channel, a dual channel DDR2 interface to 4 GB of DDR2 SDRAM memory, and a Gigabit Ethernet (GbE) external connector.

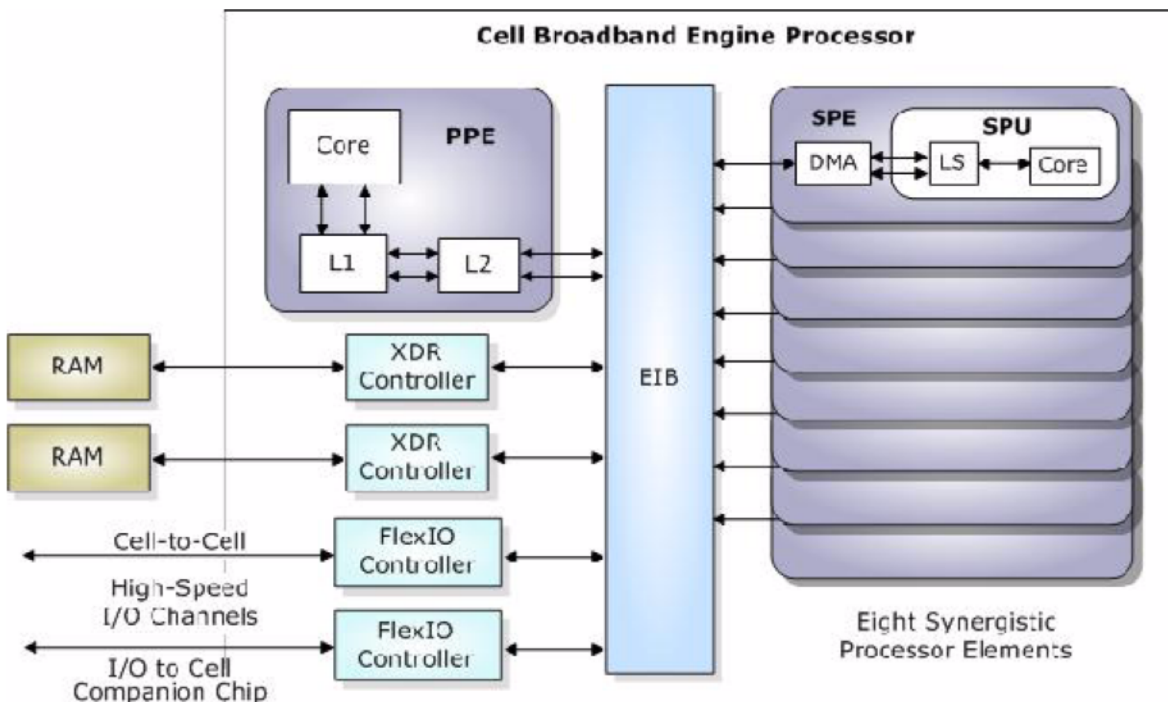


Fig 4.1: CAB Overview [3]

The CAB does not use the Cell-to-Cell FlexIO interface. The FlexIO I/O, based on Rambus technology, to companion chip interface on the CAB is active and connects to the Cell companion chip. The FlexIO connections can be configured for data rates between 400 Mhz to 8 Ghz . In the Playstation 3 the I/O is connected to Nvidias RSX graphic processor. The IOIF can only be accessed by privileged applications.

4.1.1 Cell Companion Chip

The Cell companion chip provides the high-speed interface between the Cell BE processor, the dual-channel DDR2 memory, and the following communications channels

• PCI Express Channel: The PCI Express x16 provides a high-speed serial interface with a packet-based protocol.

• Gigabit Ethernet Controller: The companion chip connects to a serializer/deserializer (SerDes) that transmits (and receives) a 1-Gbit Ethernet signal to (and from) an external source.

• Universal Asynchronous Receiver-Transmitter (UART): The CAB supports two full-duplex UART channels. One of these channels provides a communication path from the Cell companion chip to the H8 service processor and is kept at 3.3v levels. The other channel is converted to RS-232 and can be used for external I/O, primarily for factory test and debug

• External bus controller (EBC): the CAB provides a 32MB Flash boot device and battery-supported NVRAM and real time clock (RTC). These devices are connected to the Cell companion chip. The companion chip also has a DMA engine for moving data between the Cell BE, DDR2, and PCIe channels.

4.1.2 Memory

The CAB is provided with 1GB of XDR DRAM with a peak data rate of 24 GB/s. XDR DRAM allows us to have the highest sustained bandwidth for multiple, interleaved randomly addressed memory transactions, yielding over 95% utilization while allowing fine access granularity [3] [6]. The CAB also has 4 GB of DDR2 SDRAM attached to the companion chip with data rates of 2.5 to 3.0 GB/s. Both the XDR and DDR2 memory can be accessed through the Cell BE processor's Memory Flow Control (MFC) DMA engines, or through the companion chip's high-performance DMA engine, or through an external PCIe device or host. The CAB also has 32MB of flash EPROM for CAB initialization and reboot, battery-supported NVRAM (1MB) for operating system boot parameters and other user information. There is also 64KB of serial EEPROM for vital product data (VPD), Cell companion chip initialization, and board error-logging.

4.1.3 Booting the CAB

The CAB software is installed into the /opt directory on the host machine . The CAB does not have a disk, so the /opt directory needs to be mounted on the CAB as an NFS file system to make the software available to the CAB. The following sequence is needed to run an executable on the CAB

1. Log into the host workstation

2. Ssh to the CAB board using the IP address allocated for it

3. Make the directory /opt on the CAB using mkdir -p /opt

4. Mount the opt directory from host using the following command: mount -n -t nfs -o nolock,tcp your_host_IP_address:/opt /opt

5. Navigate to the executable on the opt folder and execute it.

4.2 Neural Network – Matlab implementation

A sample dataset of 38000 samples each with a 20x20 image. An image of size MxN is scanned using a sliding window of size 20x20 pixels. Each pixel is a grayscale value on a 256 point scale. The goal was to implement a neural network based classifier to determine whether a face exists in the said image. Out of the given dataset 20000 images are used for training the network. The network is trained using MATLABs NPR toolbox. The NPR Toolbox is a GUI based neural network pattern recognition wizard. It gives us a three layer MLP neural network with sigmoid output neurons. The structure is then stored in a file for later extraction of parameters. What we end up with is a perceptron for detecting the presence of faces in an image.

In our case we have a 3 layer MLP neural network with 400 inputs going to each of the 20 neurons in the first layer. There is a single neuron in the output layer. The network was trained using a scaled conjugate gradient algorithm. The following parameters were used for training

epochs: 1000

time: Inf

goal: 0

max_fail: 6

in_grad: 1.0000e-006

sigma: 5.0000e-005

lambda: 5.0000e-007

performFcn: 'mse'

trainFcn: 'trainscg'

The parameters were then extracted to work with the MATLAB script that was written to do the classification on the test dataset.

MATLAB program for classification

```
tstart = tic;
load('input_20000.mat')
load('layerbias_20000.mat');
layerbias = xt;
load('inpbias_20000.mat')
inpbias = xt;
load('inpweights_20000.mat');
inpweights = xt;
load('layerweights_20000.mat');
layerweights = xt;
 for j = 1 : 4000
 a = c_i(j,:);
a = (2*c_i(j,:)/(max(a) - min(a))) - 1;
for i = 1 : 20
    b = inpweights(i,:);
    c = a.*b;
    c_sum(i) = sum(c);
    level1_neuron(i) = c_sum(i) + inpbias(i,1);
    level1_neuron(i) = tansig(level1_neuron(i));
end
 c2 = level1_neuron.*layerweights;
c2_sum = sum(c2);
```

```
level2_output(j) = c2_sum + layerbias;

level2_output(j) = tansig(level2_output(j));

if (level2_output(j) > 0 ) level2_output(j) = 1 ;

else level2_output(j) = 0 ;

end

%a = 2 ./ (1 + exp(-2*n)) - 1;

end

telapsed = toc(tstart)
```

We use the MATLAB 'TIC'and 'TOC' functions to measure elapsed time for the program on the computer. TIC and TOC functions work together, TIC by itself saves the current time that TOC uses later to measure the time elapsed between the two. For eg: TSTART = TIC saves the time to an output argument, TSTART. The value of TSTART is only useful as an input argument for a subsequent call to TOC.

In order to compare this with our cell implementation a few changes needs to be made to the parameter files. The Cell is big endian and hence, conversion is necessary before the parameters from MATLAB can be handled by the Cell. This conversion is done in a MATLAB script cell_write.m which uses the MATLAB function swapbytes() to convert little endian floating point values to big endian floating point values. This is then written into a file to be transferred to the Cell environment. Since the CAB board is located remotely the files are transferred over SSH to the Host machine.

```
load layerbias_20000;

a = xt;

b = single(a);

fid = fopen('layerbias', 'w');

d = swapbytes(b);

d = d';

fwrite(fid, d, 'single');

fclose(fid)

load inpbias_20000

a = xt;

b = single(a);

fid = fopen('inpbias', 'w');

d = swapbytes(b);

d = d';

fwrite(fid, d, 'single');

fclose(fid)

load inpweights_20000;

a = xt;

b = single(a);

fid = fopen('inpweights', 'w');

d = swapbytes(b);

d = d';

fwrite(fid, d, 'single');

fclose(fid)

load layerweights_20000;

a = xt;

b = single(a);
```

```
fid = fopen('layerweights', 'w');

d = swapbytes(b);

d = d';

fwrite(fid, d, 'single');

fclose(fid)
```

Matlab code for preprocessing

For a fair architectural comparison a C implementation of the code was also compared against the Cell. The C implementation was done using the Visual Studio environment and code was timed in the same way.

4.2.1 Scaled Conjugate Gradient

Although a basic backpropagation algorithm adjusts weights in the direction of steepest descent it does not result in the fastest implementation. In a conjugate gradient approach the search is performed along conjugate directions [18].

Most of the training algorithms use a learning rate to determine the length of the weight update (step size). In most of the conjugate gradient algorithms, the step size is adjusted at each iteration. A search is made along the conjugate gradient direction to determine the step size, which minimizes the performance function along that line.

Most of the conjugate gradient algorithms require a line search at each iteration. It requires that the network response to all training inputs be computed several times for each search, which is computationally expensive. The scaled conjugate gradient algorithm (SCG), developed by Moller was avoids the time-consuming line search.

Trainscg [18], the MATLAB function, can train any network as long as its weight, net input, and transfer functions have derivative functions. The trainscg routine can require more iterations to

converge than the other conjugate gradient algorithms, but the number of computations in each iteration is significantly reduced because no line search is performed

Training stops when any of these conditions occur:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time has been exceeded.
- Performance has been minimized to the goal.
- The performance gradient falls below mingrad.
- Validation performance has increased more than max_fail times since the last time it decreased.

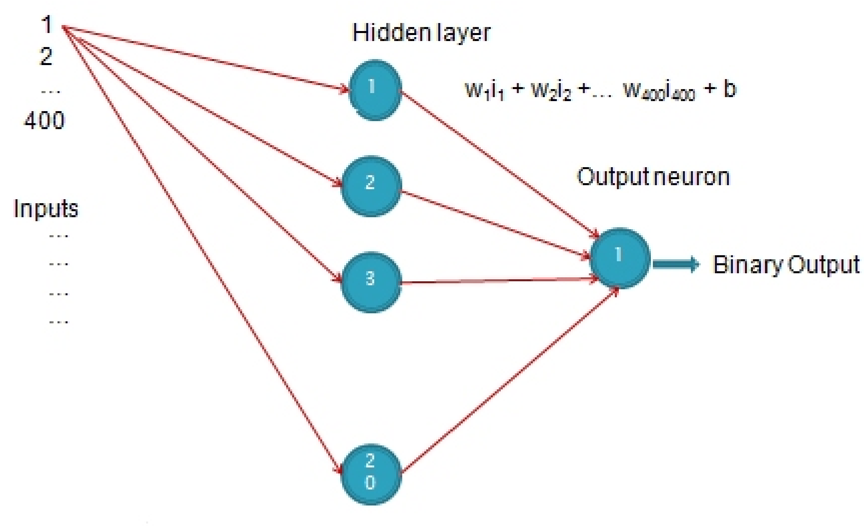


Fig 4.2 Overview of the network structure

4.3 Development environment for the Cell BE

Most of the components in the SDK were created by IBM but the basic build tools were developed by Sony. Sony chose to base its tools on the GCC toolchain which has gained wide

support and because of the fact that it is released under the GNU public license. The PPU and SPU have different instruction sets and hence the need for separate tools for the two architectures

Currently the SDK is supported only on Linux operating systems, either Fedora or Red Hat Enterprise Linux. For the purpose of the thesis I have chosen a Fedora 6 installation for the SDK. The machine that hosts the CAB board however runs Red Hat Enterprise Linux 5. As of this writing the latest version of the SDK is 3.0 however we have worked with SDK 1.1 on the CAB. SDK 3.0 provides additional domain specific libraries along with a whole new API using the libspe2 library. The following files are necessary for compiling code to run on the Cell and they are found under the folder /usr/bin

- ppu-gcc - compiles C files into assembly code
- ppu-g++ - compiles C++ files into assembly code
- ppu-as - assembles assembly files into object code
- ppu-ld - links object code to form an application

The corresponding SPU files predictably begins with spu instead of ppu.

4.3.1 Debugging applications for the Cell

The SDK provides two debuggers, one for the PPU (ppu-gdb) and one for the SPU (spu-gdb). These are available in the IBMs Full system simulator, called SystemSim [13], for the Cell Processor as well to make application development easier. The SystemSim simulator gives users nearly absolute oversight of the PPUs and SPUs. Like a debugger the simulator displays the contents of registers and memory. It also keeps track of bus usage, address translation, stack pointer location and DMA communication.

You can start SystemSim with no operating system, this is called standalone mode however there are a few restriction associated with this. You have no access to virtual memory operations and all applications must be statically linked. In the case of file operations the files have to be copied into the simulator environment or else the simulator reports a File error

during runtime. SystemSim can be configured to provide cycle accurate timing. SystemSIm was originally created by IBM to simulation for its PowerPC line of processors. In fact SystemSim reads a configuration file that declares a machine type to configure it. When SystemSim starts up it read the configuration file systemsim.tcl in /opt/ibm/systemsim-cell/lib. This file defines the resources and operation of the Cell processor. Initially the size of the simulated memory system is set to 256MB.

In Linux mode after the simulator is loaded, it 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. 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. 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. There are, however, limitations that apply when building an application to be loaded and run by the simulator without an operating system. One such instance is address-translation support, which is usually provided by the operating system. Since an operating system is not present in this mode, the simulator loads executables without address translation, so that the effective address is the same as the real address. Hence all addresses referenced in the executable must be real addresses.

SystemSim recognizes Tcl and custom scripts can be written in Tcl to automate processes. The simulator is located in /opt/ibm/systemsim-cell/bin and it can be started with the command

```
systemsim –g –q
```

The –g option tells it to open a GUI and –q option tells it to run in quiet mode. Once the simulator starts up 2 panels 2 panels can be seen. Hitting Go on the mysim window starts the simulation. The Mode button can be used to toggle between Fast, Simple or Cycle mode. Cycle mode provides cycle accurate simulation however requires a great amount of processing power. The following SPU modes are available; instruction mode, pipeline mode, fast mode. Instruction mode is used for checking and debugging the functionality of a program. Pipeline

mode is used for collecting performance statistics on the program, fast mode for fast functional simulation only.

The functional-only mode models the effects of instructions, without accurately modeling the time required to execute the instructions. In functional-only mode, a fixed latency is assigned to each instruction which can be 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.

The cycle-accurate 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-accurate mode allows you to gather and compare performance statistics on full systems, including the PPE, SPEs, MFCs, PPE caches, bus, and memory controller. You can determine precise values for system validation and tuning parameters, such as cache latency. You can characterize the system workload and forecast performance at future loads, and fine-tune performance benchmarks for future validation.

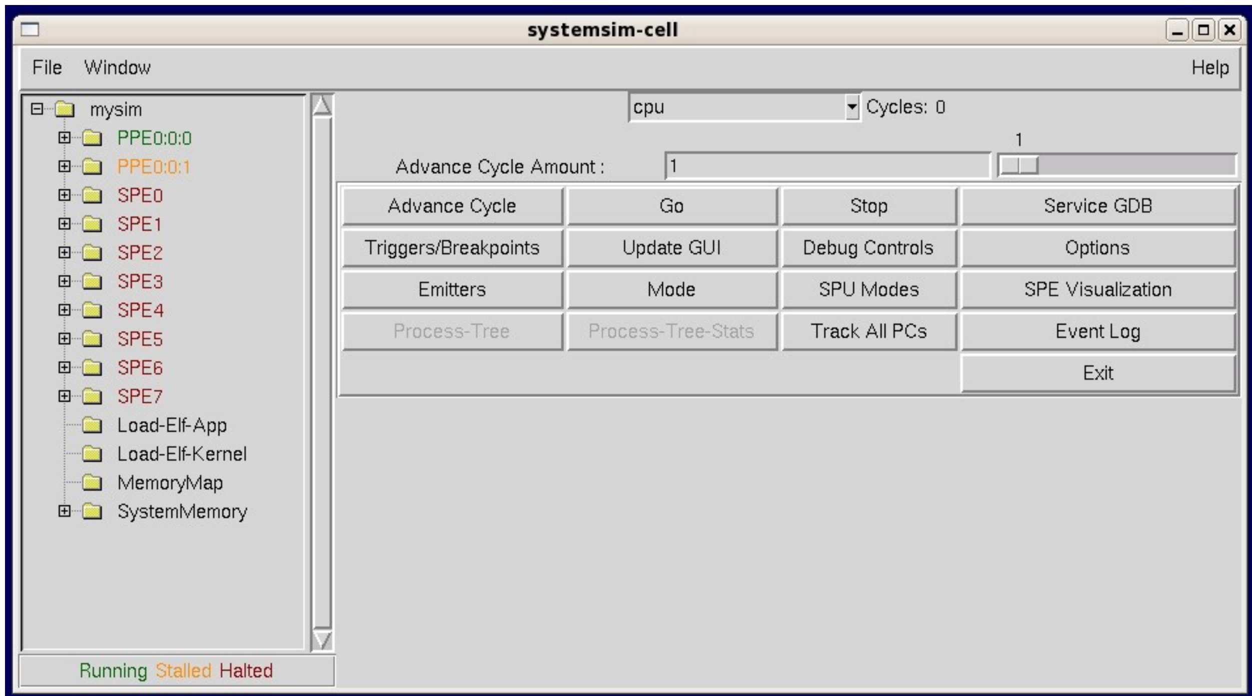


Fig 4.3: SystemSim main window

Some of the important performance statistics that are collected are

- Total cycle count
- Count of branch instructions
- Count of branches taken
- Count of branches not taken
- Count of branch-hint instructions
- Count of branch-hints taken
- Contention for an SPE's local store
- Stall cycles due to dependencies on various pipelines

The following steps can be used to collect and display simple performance statistics

1. Start the simulator by entering the following command: PATH=/opt/ibm/systemsin-cell/bin:$PATH; systemsim This command starts the simulator in command-line mode, and displays the simulator prompt.

systemsim %

2. In the command window, set the SPUs to pipeline mode. An SPU must be in pipeline mode to collect performance statistics from that SPU. In instruction mode, it will only report the total instruction count. Use the mysim spu command to set those processors to pipeline mode, as follows:

```
mysim spu 0 set model pipeline

mysim spu 1 set model pipeline

mysim spu 2 set model pipeline
```

3. In the command window, boot Linux. Boot the Linux operating system on the simulated PPE by entering

```
mysim go
```

4. In the console window, load the executables. Load the PPE and SPE executables into the simulated environment, and set their file permissions to executable, using the following commands

```
callthru source sourcename > sourcename

chmod +x sourcename
```

5. In the console window, run the PPE program. Run the PPE program in the simulation by entering the name of the executable file, as follows:

```
./sourcename
```

6. In the command window, pause the simulation and display statistics. When the program finishes execution, select the simulator control window. Pause the simulator by entering the Ctrl-c key sequence. To display the performance statistics for the three SPEs, enter the following commands

```
mysim spu 0 display statistics

mysim spu 1 display statistics
```

4.3.2 Profile Checkpoints

You can use profile checkpoints to capture such as prof_clear , prof_start and prof_stop 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. A SPE header file, profile.h, provides the interface to invoke these instructions. In addition to displaying performance information, certain performance profile checkpoints can control the statistics-gathering functions of the SPU. For example the following code represents how the above functions can be used to profile code

```
prof_clear(); // clear performance counter

prof_start(); // start recording performance statistics

... <code_to_be_profiled>

prof_stop(); // stop recording performance statistics
```

Chapter 5

Cell Implementation of the classifier

Using the MATLAB model and the scripts that I had written for the classifier, the classifier is ported onto the Cell. The key to an efficient implementation on the Cell requires attention to the load partitioning model. Giving the SPEs too little or too much to perform compromises the performance of the system [4] [22]. If the data that is sent over to the SPEs is too little the DMA overhead for data transfer becomes larger than the processing time [21]. Such is system would be quite inefficient. On the other hand sending one SPE too much data would tantamount to wasting the processing power of the other SPEs which could have shared the workload.

Several models are presented here that look at how effectively one can go about this task. Whenever a choice of a model is to be made it must be kept in mind that a DMA transfer takes at least 8 cycles on the EIB. Hence to make optimal use of it, it is not recommended to transfer less than 128 bytes at a time. In all the models attention has been paid to keep processing to a minimum in the PPE and control structures to a minimum in the SPEs.

5.1 Intra-frame and Inter-frame frameworks

Accelerating classification can be dealt with from an intra-frame perspective or inter-frame. Intra-frame splits the process of classifying an image among the 8 SPEs, thereby ensuring that the one task gets completed as soon as possible. This approach is best when an isolated task needs to be performed. An inter-frame framework however takes a different approach and focuses on throughput. It can be visualized as multiple tasks being completed in certain duration of time effectively speeding up a task. Parallelism here is at a much higher level. This is best suited to applications where multiple numbers of processes need to be carried out simultaneously. We will look at the effectiveness of each approach and do a performance

analysis of each methodology. We conclude with why we need either one or why a ‘one size fits all’ approach does not work here.

5.2 Intra-frame Load partitioning models

In this first model I investigate the effects of inter SPE synchronization and DMA data transfer. A set of 400 input values (20x20 image) is sent over to SPE0 and SPE1. The 2 SPEs process data and distribute the outputs from them to the rest of the SPEs. As is obvious, in this model the SPEs have to synchronize data transfers since each could finish processing at any point of time. The model is graphically represented in Figure 5.1. Figure 5.2 represents a mailbox based SPE synchronization protocol. Consider the following scenario where SPE2 needs to get data from SPE0’s LS. In order for that to occur SPE0 should wait till SPE2 can import it into its LS. It does that with the aid of the PPE, SPE0 once done with its processing waits on a blocking read for the PPE to write to its mailbox which it does when it receives a signal from SPE2 that it is ready. This would be a streaming pipeline model where the assumption of uniform dataflow is made.

Transferring data between SPE Local Stores straightforward if both the SPEs have the same binary since the offsets of the variables would be the same in both the SPEs. For example, if you want to get the variable target, take the LS address of the other SPE plus the offset of target within yours to find the effective address of the variable. To load that value into your variable result, the mfc_get() command would be

```
mfc_get(&result, src_spe_ls + (uint64_t)(&target), sizeof(target), tagid, 0, 0)
```

Here result is the local buffer; src_spe_ls is the LS address of the other SPE. The PPE can get this with spe_get_ls() and pass it along to the SPE as a parameter. SPEs are share the same code and static buffer addresses are the same for all SPEs. As is required with all DMAs, "result" and "target" must be naturally aligned with the same quadword offset. The spe local store address should be stored or passed in a thread and not as an unsigned int.

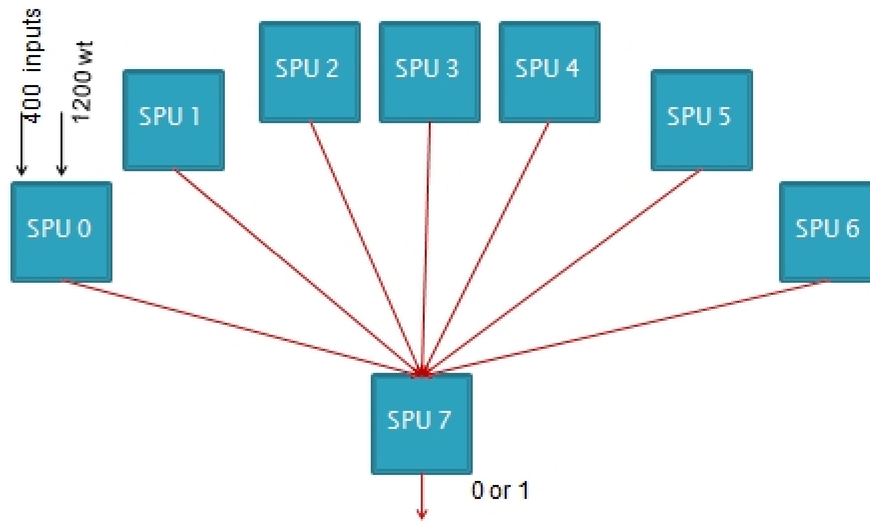


Figure 5.1 Dataflow diagram

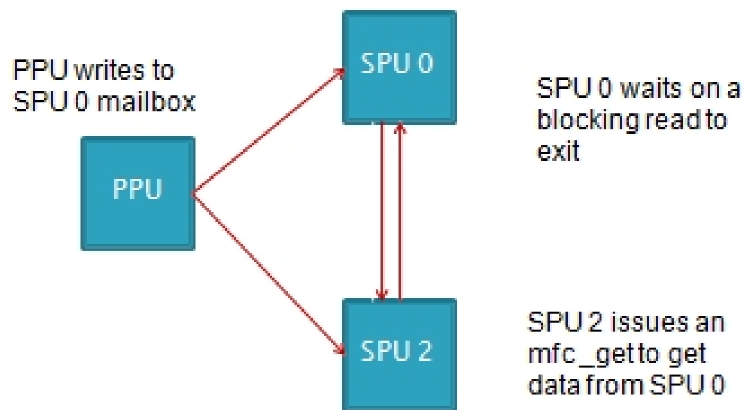


Fig 5.2: Inter SPE synchronization protocol

Given the task at hand, a model was formulated keeping the structure of the neural network in mind. Considering that the network fans in, it starts with 400 inputs going to each of the 20 neurons, it is logical to assign more SPEs to handle the hidden layer computations.

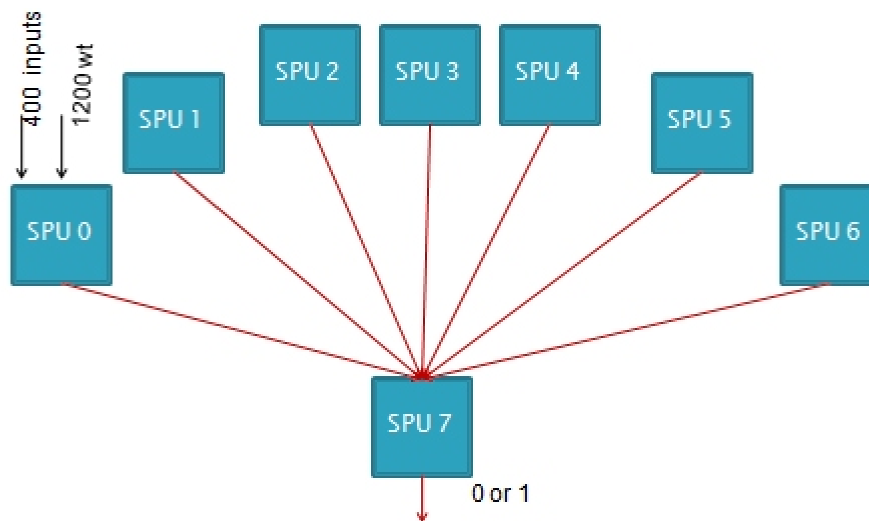


Fig 5.3: Intra-frame partitioning scheme

The average number of cycles a 16K transfer takes is 4738. The above model works as follows. Each of the 7 SPEs deal with 3 hidden layer neurons except for SPE 6 which takes 2 neurons. The inputs to each one of the above are 400 inputs, 1200 weights and the respective bias values. Each one of them processes the inputs simultaneously. SPE 7 synchronizes with the other SPEs and DMAs in the output of each hidden layer neuron to its Local Store. SPE 7 then calculates the output of the final layer from the received output hidden layer values. Synchronization is an essential part of this process, as without it invalid values can be read into the SPE 7 Load Store.

However it was seen that although the synchronization protocol worked like it should have, it was noticed that a better model can be implemented utilizing the principles of data locality.

For a 400 input model the channel stalls (due to waiting on DMA completion) were as high as 80%. It is easy to understand that waiting on DMA from the 7 SPEs have resulted in the high figures.

Figure 5.3: Performance Statistics from SystemSim

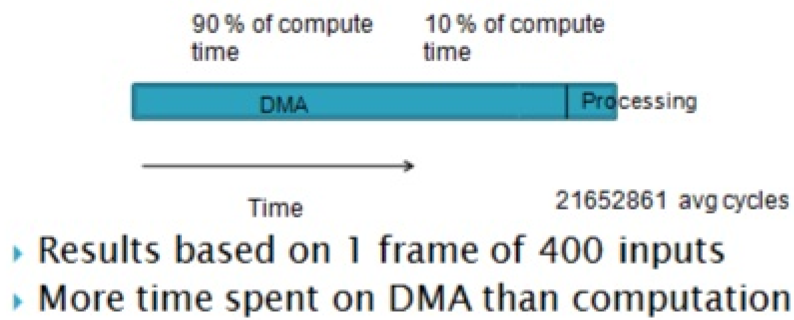


Figure 5.4: Intra-frame timing diagram

If we reuse the data sent out from the SPE within the same SPE, it ends up being a local LS access rather than a LS-to-LS transfer over the EIB. This would also help us better utilize the SPEs as well. As a rule of thumb the more time the data spends in a SPE the higher the performance benefits from work offloading.

5.3 Inter-frame SPE model

Here we have tried to isolate SPEs in order to keep interaction between them to a minimum. If each SPE is allowed to operate independently it is possible to avoid synchronization altogether. Each SPE takes a set of 400 inputs along with the weights and biases needed for classifying one set of inputs. Based on the network that is 8000 weight values and 21 bias values. These are read from the parameters file and transferred via DMA to the SPEs. Upon completion of processing the output is written to console and the thread exits. It repeats with another set of inputs read from the file and sent over. This is an iterative process until the data set is exhausted. Care has been taken to overlap file reads with DMA transfers on the PPE. It must be noted that multiple DMA transfers are needed to transfer the necessary parameters to the SPEs. The maximum size of a data transfer is 16K bytes, however this would not affect performance of the application as DMA transfers are scheduled to overlap with computation.

As noted earlier a triple buffering mechanism is utilized here. The following structure is passed to each SPE

```
typedef struct
{
int neuron_id;
int layer_id;
float input[400];
float weight[2800];
//float* input;
//float* weight;
float layer2_weight[20];
float layer2_bias;
float bias[20];
unsigned int pad;
}con_pl;
```

Fig 5.5: Performance Statistics for an isolated SPE model

As can be seen from the performance statistics for the model Channel Stalls have dropped to 12% which is a significant improvement compared to the 80% stalls that we had observed for the previous model. Compared to a MATLAB implementation the following performance numbers were obtained for the Cell implementation.

Sample configuration and results
Test inputs: 38000 Number of inputs used for training in the MATLAB NPR toolboox : 20000 Implementation on a Cell board for 8 classifications - .04s Implementation on a Cell board for 4000 classifications - 4s Implementation on MATLAB for 8 classifications - .4s Implementation on MATLAB for 4000 classifications - 28s

The above model suffers from the pitfall of having the SPEs exit every time it finishes processing. Instead the SPEs could wait in a loop for tasks to be assigned to them or better yet DMA in data till the PPE signals it to stop. This keeps PPE interaction to a minimum. The SPE continues operation as long as there is no data in its inbound mailbox. Since this operation is a local access for the SPE, it does not use up EIB bandwidth. In code this can be represented on the SPE side as

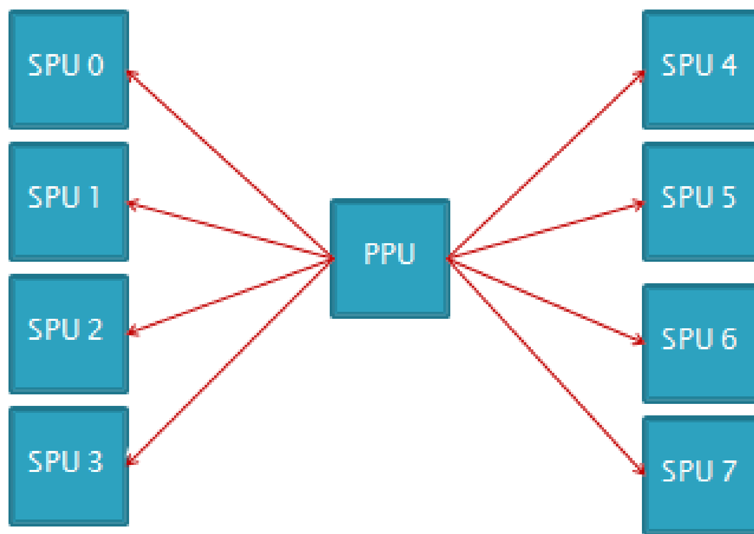
```
While (!spu_stat_in_mbox())
{
mfc_get(parameters);  // DMA in data from buffer on PPE
....... // Process Data
}
```

While on the PPE side u send a message across to the SPEs inbound mailbox signaling them that processing is done

```
If (done)
{
Spe_write_in_mbox (spe0_id,data) ;
Spe_write_in_mbox (spe1_id,data) ;
Spe_write_in_mbox (spe2_id,data) ;
Spe_write_in_mbox (spe3_id,data) ;
Spe_write_in_mbox (spe4_id,data) ;
Spe_write_in_mbox (spe5_id,data) ;
Spe_write_in_mbox (spe6_id,data) ;
Spe_write_in_mbox (spe7_id,data) ;
}
```

5.3.1 Independent SPE model

If the recurring overhead of recreating a thread is taken away, one will end up with an even better implementation. This can be done if the SPE that is once created stays alive and waits on data. It goes on in a loop until all the data is processed or until it receives a signal from the PPE instructing it to suspend processing. Such a message can be passed through mailboxes while the SPE DMAs in input data without PPE intervention. The advantage of such a system is that it takes the PPE out of the equation and the SPEs need minimal control signals from it. The SPE continues to read in data based on configuration information it reads in from a file. However if this parameter is not set correctly invalid data will be processed.



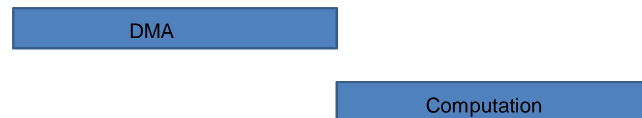
5.6: An Independent SPE model

In the above system each SPE holds a complete network thereby avoiding any need for data to be transferred outside an SPE's Local store and thereby avoiding an expensive DMA operation. Each SPE receives 400 inputs, 8000 weights (400 x 20) along with 21 bias values along with

other configuration information during setup. Once setup is completed the SPEs settle into a read-process cycle where it continuously reads in data from a buffer in the PPE and utilizes the weights stored in its Local store to compute the classifier outputs. It must be noted that the initial 8000 weights are DMAed into 3 buffers. These DMAs are overlapped with computation hence masking the DMA latencies. Also a single DMA cannot exceed a size limit of 16K byte. Although not a repetitive process it still serves to utilize the data already available.

Eight buffers are available on the PPE for data to be read from by the SPEs. The PPE partitions the input file into 8 segments which are then read into buffers in order to facilitate independent reads. The use of 8 buffers avoids contention at a single buffer which could slow down DMA transfers. A shared variable would only add onto the Channel stalls degrading performance.

DMA followed by processing



5.7 No Buffering for DMAs

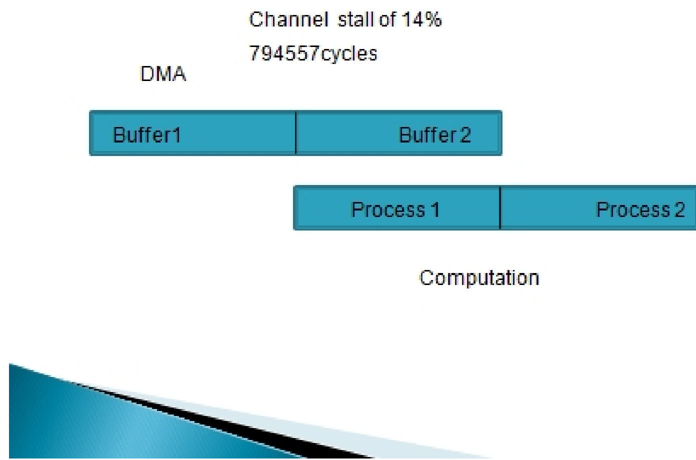


Fig 5.8: Double Buffering

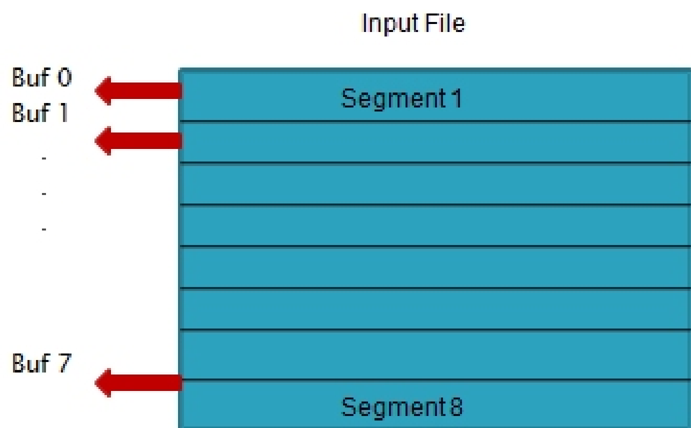


Fig 5.9: File partition

The structure of the Input file is shown above. The file read follows a standard file stream read in C and is sequential. Each SPE is assigned a buffer to operate on. The address of this buffer is passed to the SPEs along with the configuration info. Once a read cycle is completed the SPE checks to see if there are more inputs to be processed, if so it increments the pointer to the buffer to point to the next set of 400 inputs and proceeds to read from that location.

```
While (data available in buffer)
    { //DMA in data
  mfc_get((void *)&inp_str,(unsigned int)pl_ct.layer_id + ((done -1)*1600),1600,30,0,0);
  mfc_write_tag_mask(1<<30);
      Process data
    }
```

A typical compute cycle looks like the following

```
prod1[l] = spu_insert(pl_ct.weight[l*4],prod1[l],0);
prod1[l] = spu_insert(pl_ct.weight[l*4 +1],prod1[l],1);
prod1[l] = spu_insert(pl_ct.weight[l*4 +2],prod1[l],2);
prod1[l] = spu_insert(pl_ct.weight[l*4 +3],prod1[l],3);
prod2[l] = spu_insert(inp_str.inputs_t[l*4],prod2[l],0);
prod2[l] = spu_insert(inp_str.inputs_t[l*4+1],prod2[l],1);
prod2[l] = spu_insert(inp_str.inputs_t[l*4+2],prod2[l],2);
prod2[l] = spu_insert(inp_str.inputs_t[l*4+3],prod2[l],3);
prod[k] =  spu_madd(prod1[l],prod2[l],prod[k]);
prodsum[k] =
(float)spu_extract(prod[k],0)+(float)spu_extract(prod[k],1)+(float)spu_extract(prod[k],2)+(float)spu_extract(pr
od[k],3)  +(float)pl_ct.bias[k];
```

```
prodsum[k] = tanh(prodsum[k]);
```

5.3.2 Memory Mapped Files

Memory mapping the input files guarantees simultaneous DMA transfers and it also takes the PPE out of the equation. Reading data into a PPE buffer before the SPE can access it can be avoided if the SPE gets direct access to the file. However SPEs do not have file read capabilities. The solution would be to map it to the Cells main memory.

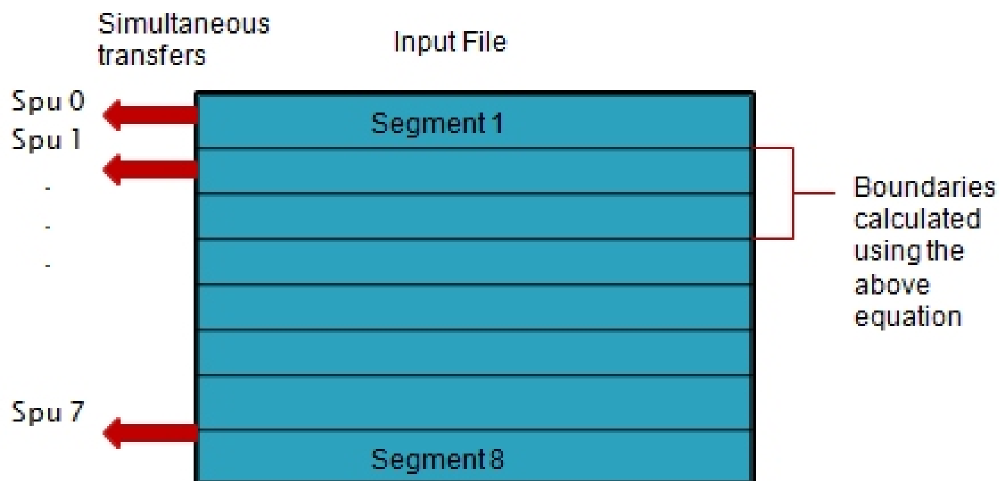


Fig 5.10: Memory mapped File partition

It also avoids the limitation of a buffer size on the PPE due to its small cache size (32 KB). In this scenario the whole segment (1/8th of a file) need not be buffered in at once. The SPE can DMA in chunks of 400 inputs as and when needed.

Memory map the file in PPU using :

```
comptx = open ("scaled_inputs", O_CREAT | O_RDWR, 0755));
```

```
float  *map;

map = mmap(0,120000,PROT_READ,MAP_SHARED,comptx,0);
```

File segment boundary for each each SPU calculated based on neuron_id as

```
pl_ct.mmap_address[0] + 12800+ (pl_ct.neuron_id - 1)*boundary + (done - 1)*1600
```

The SPE has access to memory space and this streamlines dataflow by avoiding an extra transfer to the PPE.

The configuration information that is passed to the SPE takes the following form.

```
typedef struct{
unsigned int mmap_address[4];
int neuron_id;
int layer_id;
int address_buf[4];            //  Structure padded to be
float input[400];              //a multiple of  128 bytes
float weight[2800];
float layer2_weight[20];
float layer2_bias;
float bias[20];
unsigned int pad;
}con_pl;
```

It must be noted here that I have used the libspe library as opposed to the newer libspe2 library which uses a similar function spe_in_mbox_write() which takes a different set of parameters. The outputs can be printed onto the console or into a file.

	Memory mapped	File read
Avg CPI	1.75	2.14
Avg Cycle count for 1200	794557	986899
Avg Channel Stalls	14%	27%

Fig 5.11 Performance comparison of PPE buffer vs. Memory mapped models

	Cycles	Time	CPI
Computation and DMA	281138	8.7856e-005	1.78
Computation alone	213233	6.6635e-005s	1.35
DMA overhead	67905	2.1220e-005	NA

Fig 5.12 Performance cycles split-up for DMA and computation

5.4 Hardware Implementation

The implementation on the Cell is compared with the MATLAB version. The machine that hosts MATLAB has a Core 2 Duo processor at 2.6 Ghz with 4GB RAM and a 800 Mhz FSB. Run times are recorded for different number of classifications and as can be seen the Cell scales admirably for larger datasets.

In order to accommodate the volume of data transfers from memory to the SPEs certain tweaks are necessary on the CAB. The CAB software provides two Linux kernel images for the CAB one that uses a 4K base page size and a version that uses a 64K base page size. The performance advantages pertain to the translation lookahead buffer (TLB) maintained by the Cell BE processor hardware. The TLB acts as an address translation cache and provides a fast method for converting the effective addresses on the internal Cell BE bus to the real physical addresses that are needed for memory access. The Cell BE has a fixed number (256) of these TLB entries. Each entry in the TLB corresponds to the address range of one base page of 4K or 64K. When an address is not in the TLB, the Cell BE processor's memory management hardware must find the correct address by performing a "table walk" of the virtual memory data structures. This is a rather slower process and has a significant performance effect on the data transfer. The 4K base page allows 256 x 4096 or 1MB of data to be accessed before the TLB is exhausted while the 64K base page allows access to 16MB of data.

SPE to SPE local store transfers also require address translation, and hence these addresses must reside in the TLB for high-performance applications. The 64K base page size allows the TLB to cover all of the SPEs' local stores as well as an additional 14MB of XDR memory.

Another mechanism which has been utilized in this application called HUGETLBFS, allows the use of 16MB pages. One can have the Linux kernel to map a specified number of these HUGE pages into the application's memory, and each 16MB takes only 1 TLB entry.

Cell vs. X86 platform (Core 2 Duo at 2.66 Ghz)

Hardware comparison – Cell vs Intel Core 2 Duo at 2.66 Ghz

Classifications	Cell	C program	MATLAB (Intel)
208	.012	.025	1.592
808	.018	.077	7.098
1208	.021	.135	11.961
1608	.0256	.174	17.605
2408	.033	.25	31.934
4808	.056	1.15	97.08
16008	.16	3.768	1006
32008	.315	7.306	4010

Fig 5.12 Performance comparison Cell and X86 runtimes (all times in seconds)

Cell vs Core 2 Duo @ 2.66 Ghz

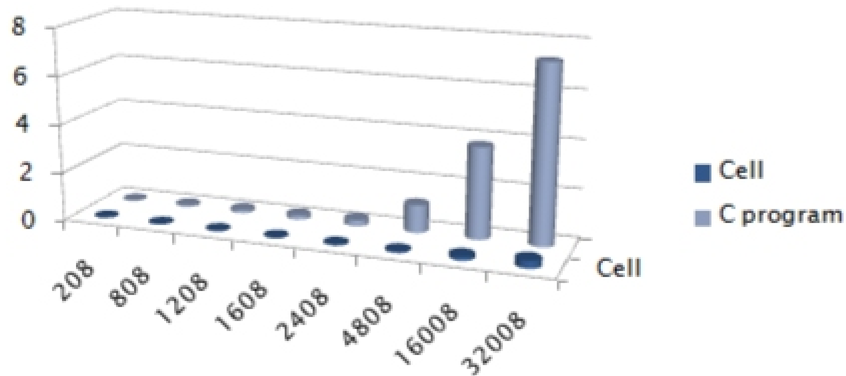


Fig 5.13 A comparison of the run times in seconds of the algorithm

5.5 Conclusion

The cell achieves a speedup of approximately 23 times for about 32000 classifications. For smaller quantities of data the differences are marginal, however as data volume scales the benefits are obvious. This is a result of the overhead associated with SPE thread and state creation. Larger datasets hide this latency and cell performance picks up.

"Autonomous, asynchronous computing is critical in maximizing computational throughput" [19]. This is the principle I have tried to follow in choosing a suitable framework for this project. Comparison of intra-frame and inter-frame techniques reveals unusually high stalls on DMA. It is a result of inter SPU access of data and the synchronization overhead that it necessitates.

Although an inter-frame scheme does not accelerate an individual classification, given a set of frames to process, throughput is greatly increased. In most classification applications this is likely the case. The ideal scenario to maximize throughput is to have each SPE working independently [19], directly DMAing in data that it needs from separate locations. Data should be 'pulled' into the SPEs rather than 'pushed' in by the PPE. Also by directly transferring data from main memory to SPE, a redundant PPE load is avoided. The memory restriction of having to transfer an input file (which is usually large) into the PPE cache is also bypassed by a direct main memory access. Independent accesses from distinct locations ensure that there is no contention at a memory location resulting in unnecessary stalls. The inter-frame model hence provides the best fit for the problem at hand.

Appendix

Accessing the CAB board

The following steps document the process flow that was used to develop code for the Cell Processor as well as steps to accessing and running the code. Since a remote CAB board was used, almost all development was done an X86 machine. I have utilized a Fedora Core 6 installation for the IBM SDK installation. The version of the SDK installed was IBM SDK 2.0, however since the remote CAB uses SDK 1.1. SDK 2.0 ships with the older libraries, hence all that is necessary is to point to the required libraries in the Makefile.

The source files are compiled and debugged on the local machine. IBM has stopped shipping SystemSim with release 3.0 onwards and hence must be purchased separately. Once the code is functionally verified, it is transferred onto the machine (Orlith.cse.psu.edu) that hosts the remote CAB board via ssh. It should be noted that the same can be done on the host machine if remote access is not available or in the case of large files which have to be transferred over the network.

Ssh to the CAB, using the IP allocated to it, from Orlith as root. Now mount the /opt directory from host if it is not visible.

```
mount -n -t nfs -o nolock,tcp 192.168.40.20:/opt /opt
```

where 198.168.40.20 is the address of the host Orlith. Navigate to the folder

```
/opt/ibm/cell-sdk/prototype/src/samples/tutorial/neural_pll
```

Change permissions on the executable using

```
chmod +x executablename
```

Timing the classifier code

The code can be timed using the following function. Calling the function from any point in the code with an argument of 0 starts the timer and a second call with an argument 1 ends the timer.

```
void time_int(int print)
{
static struct timeval t1; /* var for previous time stamp */
static struct timeval t2; /* var of current time stamp */
struct timezone tzp;
double elapsed_seconds;
if(gettimeofday(&t2, &tzp) == -1) exit(0);
if(print == 1)
{
elapsed_seconds=(double)(t2.tv_sec - t1.tv_sec) + ((double)(t2.tv_usec - t1.tv_usec))/1000000;
printf("Time spent [%.2fs] \n", elapsed_seconds);
}
t1 = t2;
}
```

Cell Performance Tips

The Cell BE processor has eight SPEs and one PPE. Even if the computations do not seem well suited to the SPEs (for example, branchy scalar code) might be best offloaded. Use the PPE as the control processor to co-ordinate SPE execution and assisting with exceptional events such as virtual memory management (VMM) misses.

Autonomous, asynchronous computing is critical in maximizing computational throughput. If the workload is variable the strategy is to let the SPEs algorithmically partition the work. the control processor can place work requests into a work queue or a set of work queues. If the work requests are non-predictable, the SPEs can self-arbitrate for work from a single queue .When these tasks are predictable in duration, the control processor can distribute the work amongst separate queues for each SPE which is the approach we have used in the PPE buffer model.

The Cell BE architecture encourages SPE programmers to manually initiate all transfers in and out of the SPE's local store (LS). This forces the programmer to be aware of all data accesses. To achieve efficient SPE data accesses, programmers should consider data alignment, access patterns, and location.

The SPE's memory flow controller (MFC) supports transfers of 1, 2, 4, 8, and n*16 (up to 16k) bytes, transfers less than 16 bytes must be naturally aligned and have the same quad-word offset for the source and destination addresses. The Element Interconnect Bus (EIB) overhead is minimized if transfers are at least 128 bytes, and transfers greater than or equal to 128 bytes should be cache-line aligned (aligned to 128 bytes). Avoid PPE pre-accesses to large data sets if possible, so that most SPE-initiated data transfers come from system memory, instead of the PPE's L2 cache. Transfers from system memory have high bandwidth and moderate latency, whereas transfers from the L2 have moderate bandwidth and low latency.

Instead of the PPE pushing data to the SPE using the SPE's proxy command queue, let the SPE pull the data using SPE to initiate the DMAs. There are eight times more SPEs than PPEs and the SPE command queue is twice as deep as the proxy command queue. The number of cycles

to initiate a transfer from the SPE is smaller than the number of cycles to initiate the same transfer from the PPE.

The SPE only accesses local store a quad-word at a time. Hence scalar (subquad-word) loads and stores require several dependent instructions. This is due to the fact that scalar loads are rotated into the vector element location, and stores require a read, scalar insert, write operation. Change the scalars to quad-word vectors. Cluster scalars into groups and load multiple scalars at a time using a quad-word memory access. Manually extract or insert the scalars on an as-needed basis. This will eliminate redundant loads and stores.

Cell programming introduces the concept of 'intrinics'. When programmers write code in a high-level language (for example, C or C++), they rely on compiler technology to auto-vectorize the code to exploit the SIMD capabilities. However it is difficult to obtain optimized code for many applications using this approach. This has resulted in programmers having to resort to using assembly for performance-critical sections to achieve the results they desire. Writing assembly on the SPE can be a daunting task for large, complicated code. A compromise solution is the use of 'intrinsics'. Intrinsics are essentially inline assembly with C function call syntax. They provide the programmer explicit control of the instructions used, but eliminate many of the optimization tasks that compilers are good at.

Loops are the foundation in nearly all programs and if the number of loop iterations is constant, then consider removing the loop altogether. If the number of loop iterations is variable, consider unrolling the loop as long as the loop is relatively independent. The SPE has a large register file, and significant loop unrolling can be accomplished before register spilling occurs. That occurs when the instantaneous number of active variables exceeds the size of the register file.

Branches are relatively expensive (up to 18-19 cycles when mis-predicted). Not only are they expensive from their issuance and stalls due to mis-predicts, they create a boundary for scheduling optimization. The programmer can explicitly direct branch prediction using the __builtin_expect language extension which tells the compiler that one is more likely than the other.

The SPE contains only a 16x16 bit multiplier. Therefore, performing a 32-bit integer multiply takes five instructions -- three 16-bit multiplies and two adds -- to accumulate the partial products. To avoid extraneous multiply cycles, if the operands are less than 16 bits in size, cast them to unsigned shorts prior to multiplication to take advantage of the native multiplier. Constants should also be cast since they have an implicit type of int.

Programmers have been improving application performance by developing tables of pre-computed values. For SPE programs, this is typically not ideal. Tables do not SIMDize well and consume valuable LS space. It would be better to exploit the SPE's huge computational capacity by instead computing values on the fly. It should be noted that if this happens to be computed in a loop you might want to pre-allocate depending on the number of recurring cycles it takes up.

The SPE local store is a limited resource. 256Kbytes is available for program, stack, local data structures, and DMA buffers. Not all of the optimization techniques mentioned above can be implemented in an application.

Bibliography

[1] IBM, IBM Cell Broadband Engine Programming Handbook.

http://www.ibm.com/developerworks/power/cell/

[2] IBM, IBM Cell Broadband Engine Architecture.

http://www.ibm.com/developerworks/power/cell/

[3] Mercury, Cell Accelerator Board, Mercury Hardware Device Note DOC-DVN-CAB-1.6

http://www.mc.com/products/boards/accelerator_board2.aspx

[4] IBM, Programming the Cell Broadband Engine Architecture, Examples and Best Practices

http://www.redbooks.ibm.com/abstracts/sg247575.html?Open&pdfbookmark

[5] IBM, Cell Broadband Engine Programming Handbook including the PowerXCell 8i Processor

http://www.ibm.com/developerworks/power/cell/

[6] Mercury, Cell Workstation Development System, Users Guide

http://www.mc.com/products/productdetail.aspx?id=9434

[7] IBM, C/C++ Language Extensions for Cell Broadband Engine Architecture, Version 2.5

http://www.ibm.com/developerworks/power/cell/

[8] IBM, Cell Broadband Engine SDK Libraries Overview and Users Guide, Version 1.1

http://www.ibm.com/developerworks/power/cell/

[9] IBM, SPE Runtime Management Library, Version 1.1

http://www.ibm.com/developerworks/power/cell/

[10] IBM, Synergistic Processor Unit Instruction Set Architecture, Version 1.2

http://www.ibm.com/developerworks/power/cell/

[11] Sony, SPU C/C++ Language Extensions, Version 2.1

http://train.psp3d.com/downloads/PS3/PDFs/CBE/SPU_language_extensions_2.1.pdf

[12] IBM, Performance Analysis with the IBM Full-System Simulator

http://www.alphaworks.ibm.com/tech/cellsystemsim

[13] IBM, IBM Full-System Simulator User's Guide, Version 0.02

http://www.alphaworks.ibm.com/tech/cellsystemsim

[14] IBM, IBM Full-System Simulator Command Reference, Version 0.01
http://www.alphaworks.ibm.com/tech/cellsystemsim

[15] IBM, PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual, Version 2.06c
http://www.alphaworks.ibm.com/tech/cellsystemsim

[16] MATLAB Neural Network Toolbox™ 6 User's Guide
www.mathworks.com/access/helpdesk/help/pdf_doc/nnet/nnet.pdf

[17] F. Smach, M. Atri, J. Miteran, M. Abid, "Design of a neural networks classifier for face detection", Journal of Computer Science, March, 2006

[18] MATLAB Neural Network Toolbox documentation
http://www.mathworks.com/access/helpdesk/help/toolbox/nnet/index.html

[19] Daniel Brokenshire (IBM), "Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance."

[20] Jed Scaramella , Mathew Eastwood, " Solutions for the Datacenter's Thermal Challenges-IDC"

[21] Thomas Chen (IBM), Ram Raghavan (IBM), Jason Dale (IBM), Eiji Iwata (IBM), "Cell Broadband Engine Architecture and its first implementation, a performance view "

[22] Programming the Cell Processor for Games, Graphics and Computation, Mathew Scarpino, 2009 Prentice Hall