ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Διπλωματική Εργασία

«Παραλληλοποίηση αλγορίθμου Bayesian Intrapersonal/Extrapersonal Classifier για εκτέλεση στον πολυεπεξεργαστή Cell Broadband Engine»

ΠΕΛΕΚΑΝΟΣ ΝΕΚΤΑΡΙΟΣ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Παπαευσταθίου Ιωάννης, Επίκουρος Καθηγητής Π.Κ. (Επιβλέπων)

Δόλλας Απόστολος, Καθηγητής Π.Κ.

Πνευματικάτος Διονύσιος, Αναπληρωτής Καθηγητής Π.Κ

SUMMARY

The slow progress in the performance of the traditional architectures let the high performance computing community to examine alternative architectures. These architectures are trying to deal with the limitations of the modern single core cache-based designs. In this work we examine a new and very promising multicore architecture, the *Cell Broadband Engine Architecture*.

The *Cell Broadband Engine* (Cell) is a multicore processor recently developed by Sony, Toshiba and IBM. It was originally designed for the Playstation 3 game console, but its capabilities also make it well suited for various other computation-intensive applications. The Cell processor is capable of achieving impressive levels of performance for complex scientific applications. These levels of performance can be reached by exploiting the several dimensions of parallelism that Cell provides.

In this thesis the computational power of the Cell processor was applied to the face recognition problem and more specifically to *Bayesian Intrapersonal/Extrapersonal Classifier* (BIC), a complex face recognition algorithm based on probabilistic matching techniques. In order to achieve better performance for the BIC algorithm we parallelized and optimized the most computation–intensive parts of the algorithm. During the porting procedure of the algorithm on the Cell processor many problems were encountered, mostly related with data motion, parallelism levels and scheduling.

In the final results a considerable improvement of performance was achieved, but with a significant overhead too, which prevents us from reaching the desirable results.

Table of Contents

List of Figures

List of Tables

"You do not really understand something unless you can explain it to your grandmother."

– Albert Einstein

ACKNOWLEDGEMENTS

It would not have been possible to write this thesis without the help and support of the kind people around me, to only some of whom it is possible to give particular mention here.

First of all I would like to express my sincere appreciation to my supervisor, professor I. Papaefstathiou, whose expertise, understanding, and patience, added considerably to the completion of my thesis.

Besides my supervisor, I would like to thank the rest of my thesis committee, professors A. Dollas and D. Pnevmatikatos for the assistance they provided at all levels of my five-year studies.

Very special thanks go to Mr. M. Kimionis, for his help and encouragement during all the time I was in Microprocessors & Hardware Laboratory.

I must also acknowledge the postgraduate students G. Chrysos and P. Christou for their important help they offered me in this research.

Many thanks and to all the other members of the Microprocessors & Hardware Laboratory, postgraduate and undergraduate students, for their help and all the good times we had together at the laboratory.

And finally I am grateful to Doris for some very special moments when the going was tough and for her personal support and great patience at all times.

CHAPTER 1

Introduction

During the last decade high performance computing became very popular; as the need for more computational power grows high performance computing is trying to serve these needs. More and more multicore processors are being designed to fulfill the demands of the market. More processors mean more complex problems for the processors designers, gate density, power consumption, and efficient memory hierarchies are some of the problems they are facing. The scientific and industrial communities are looking for alternative solutions that can keep up with the insatiable demand of computing cycles and yet have a sustainable market outside the scientific world.

A major trend in computer architecture is the design of multi-core-systems-on-a-chip processors which can integrate several identical independent processing units on the same die, together with network interfaces, acceleration units and other specialized units. This technological trend is driving the development of high performance processors that are holding enormous computational power on a single chip. The burden is now being shifted, from the architecture which is becoming simpler and more streamlined to the software. Software is now required to extract several forms of parallelism and directly coordinate a plethora of computational and communication activities across various levels of memories and functional units. This is exactly and the purpose of this thesis, to take advantage of the computational power that a multicore processor offers and use it for the needs of a specific application. Moreover we are trying to extract the computational power of a multicore processor through the partial redesign of an existing application in order to achieve better performance for the application.

The multicore processor that is being used in this project is the *Cell Broadband Engine* [1] processor that was jointly developed by IBM, Sony and Toshiba, is the new member of the IBM Power/PowerPC processor family. The initial target was the PlayStation 3 game console [2], but its capabilities also make it well suited for various other computation-intensive applications such as visualization, image and signal processing, bioinformatics etc [3], [4], [5], [6]. The Cell BE is a heterogeneous chip with nine cores capable of massive floating point processing, optimized for

compute-intensive workloads and broadband, rich media applications; for these characteristics Cell became very popular in the scientific community [7].

As the computational power of multicore processors is increasing, the computation complexity of the applications is also increasing. The databases are getting extremely large, as well as and their processing time. One problem that needs to process many data and is also computationally complex is the face recognition problem. The face recognition is very popular in many applications such as computer vision, image analysis, psychology, security, etc. In this thesis an existing application for face recognition is being used, the Colorado State University (CSU) Face Identification Evaluation System [8], [9]. This application is evaluating the performance of several face recognition algorithms, from this collection of algorithms the most complex was selected, the Bayesian Intrapersonal/Extrapersonal Classifier (BIC) [10], [11], [12].

The purpose was not the complete redesign of the application or the algorithm in order to be faster, but to execute the BIC algorithm on the Cell processor without changing the CSU application logic and achieve as much as possible performance. The approach that was followed in order to accomplish our purpose was to parallelize only the most time-consuming functions of the algorithm. This was maybe not the best approach but certainly was the fastest way to port the CSU application on Cell processor and get a respectable performance improvement.

The rest of this thesis is organized as follows: Chapter 2 introduces the Cell Broadband Engine processor and the Playstation 3 that was used in the implementation. Chapter 3 outlines the face recognition problem the CSU application and the BIC algorithm. Chapter 4 briefly describes the development process that was followed for the implementation. Chapter 5 presents the final results of the implementation and Chapter 6 has some conclusions from this work.

CHAPTER 2

Platform

This chapter describes the platform that was used for the implementation. The Cell processor, its architecture as well as and the Playstation 3 game console are described in this chapter. The purpose of the chapter is to introduce the reader to the architecture of Cell and to the hardware that was used in the implementation. Many aspects of the implementation that will be discussed later are based on concepts of this chapter.

2.1 Cell Processor

The *Cell Broadband Engine* (CBE) [13], known as *Cell* is a nine-core implementation of the *Cell Broadband Engine Architecture* (CBEA) [1], [14], [15], **Figure 1** gives an architectural overview of the Cell B.E. The CBEA is a new architecture that extends the 64-bit PowerPC Architecture, CBEA and CBE are the results of collaboration between Sony, Toshiba and IBM (STI), formally started in early 2001.

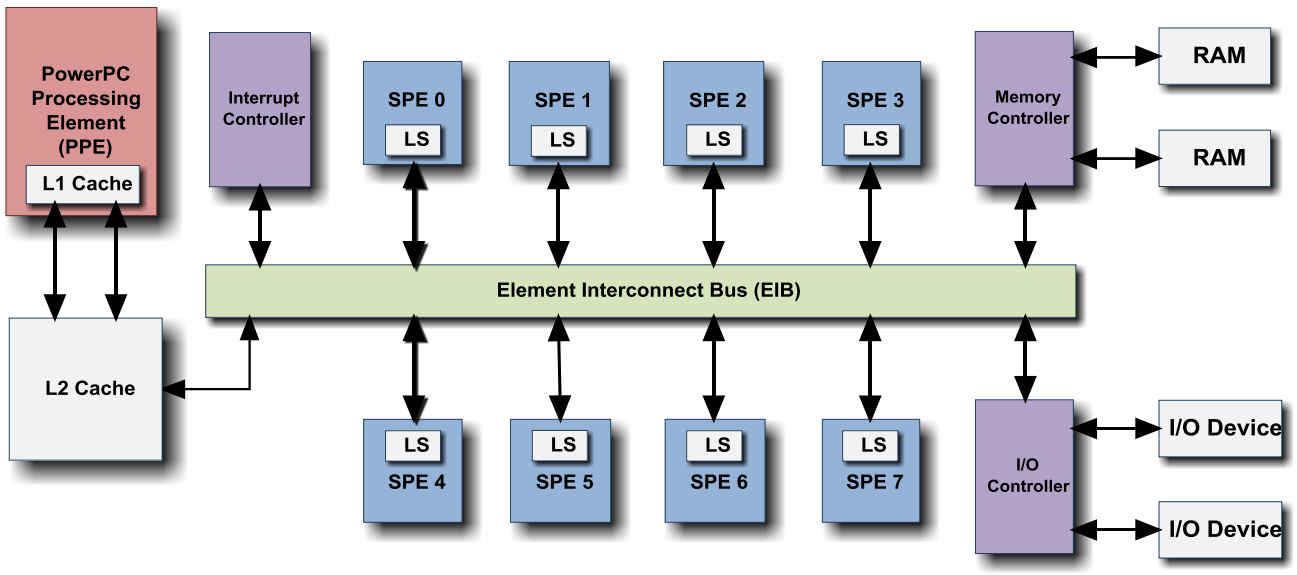

Figure 1: Cell Broadband Engine Architecture

It was initially designed for the *Sony PlayStation 3* (PS3) game console, but rapidly became famous in the scientific community for its computing capabilities. The Cell Broadband Engine (Cell B.E.) is a heterogeneous multicore chip that is significantly different from conventional multiprocessor or multicore architectures. It consists of a traditional PowerPC microprocessor called *Power Processor Element* (PPE) that controls eight SIMD co-processing units called *Synergistic Processor Elements* (SPEs), a high speed memory controller, and a high bandwidth bus interface, the *Element Interconnect Bus* (EIB), all integrated on a single chip.

2.1.1 Power Processor Element

The *Power Processor Element* (PPE) [1], [15] is a 64-bit processor representative of the Power Architecture, optimized for design frequency and power efficiency. The PPE consists of the *Power Processing Unit* (PPU) and a *Power Processor Storage Subsystem* (PPSS); **Figure 2** shows a simple block diagram of the PPE.

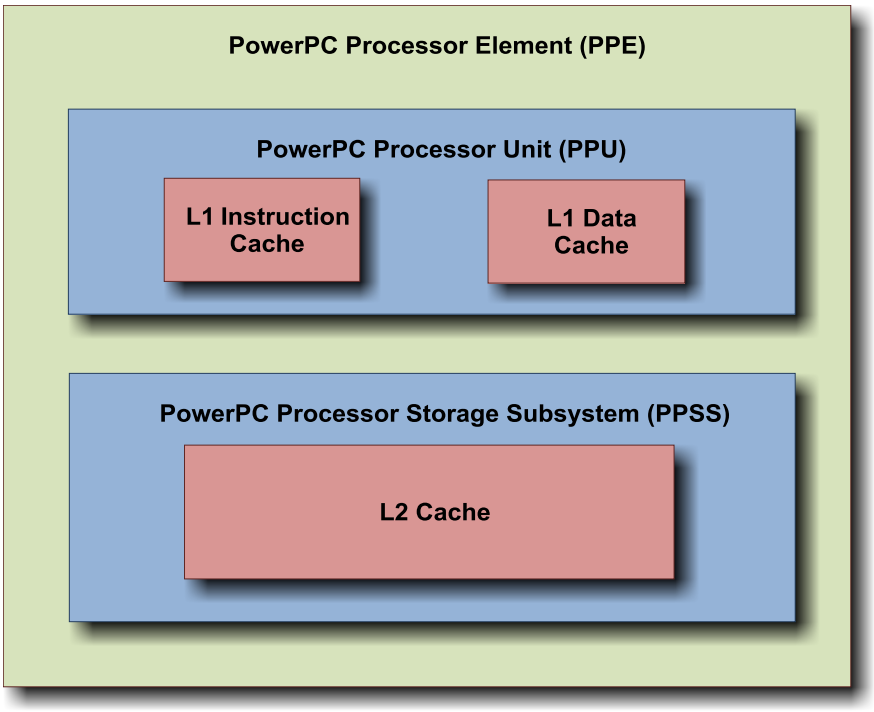

Figure 2: PowerPC Processor Element (PPE) block diagram

The PPU deals with instruction control and execution. It includes a 32 KB 2-way set associative instruction cache and a 32 KB 4-way set associative write-through data cache. Beside the standard floating point unit (FPU) the PPU also includes a short

vector SIMD engine, VMX [16], an incarnation of the PowerPC Velocity Engine or AltiVec, a branch unit and a virtual management unit. The PPE's register file is comprised of 32 64-bit general purpose registers, 32 64-bit floating-point registers and 32 128-bit vector registers.

The PPSS handles memory requests from the PPE and external requests to the PPE from other processors or I/O devices. It includes a unified (instruction and data) 512 KB 8-way set associative write-back cache, various queues and a bus interface unit.

While the PPE uses the PowerPC instruction set and is binary compliant with the PowerPC 970 architecture [17], its design is substantially different, it is not based on an existing design on the market today. The PPE is a *dual-issue, dual-thread, in-order* processor with a relatively simple architecture, which results in considerably smaller amount of circuitry than its out-of-order execution counterparts and lower energy consumption. This can potentially translate to lower performance, especially for applications heavy in branches. However, the high clock rate, high memory bandwidth and dual threading capabilities may make up for the potential performance deficiencies. The PPE seems to provide two independent execution units to the software layer. In practice the execution resources are shared, but each thread has its own copy of the architectural state, such as general-purpose registers.

Although clocked at 3.2 GHz PPE looks like a quite potent processor, its main purpose is to serve as a controller and supervise the other cores on the chip. In a Cell based system the PPE will run the operating system (OS) and most of the applications but compute intensive parts of the OS and applications will be offloaded to the SPEs. Thanks to the PPE's compliance with the PowerPC architecture, existing applications can run on the Cell out of the box, and be gradually optimized for performance using the SPEs (see *2.1.2*), rather than written from scratch.

2.1.2 Synergistic Processor Elements

One of the key architecture features that enable the Cell Broadband Engine's breakthrough performance is the *Synergistic Processor Element* (SPE) [18], [19], [15], [1]. Each of the eight SPE's consists of a *Synergistic Processing Unit* (SPU) and a *Memory Flow Controller* (MFC); **Figure 3** shows a simple block diagram of the SPE.

The SPU deals with instruction control and execution. It includes a single register file with 128 registers, each one 128 bits wide, a unified 256-KB *Local Store* (LS), an instruction-control unit, a load and store unit, two fixed-point units, one floating-point unit, and a channel-and-DMA interface. Each SPU is an independent processor with its own program counter and is optimized to run SPE threads spawned by the

PPE. All SPU instructions are inherently *SIMD* operations that can run at four different granularities: 16-way 8-bit integers, 8-way 16-bit integers, 4-way 32-bit integers or single-precision floating-point numbers, or 2-way 64-bit double-precision floating point numbers [20]. Like the PPU, SPU is an *in-order* processor with *two instruction pipelines*, odd and even. The even pipeline is being devoted to arithmetic operations and the odd is being devoted to data motion. Particularly the floating point and fixed point units are on the even pipeline while the rest of the functional units are on the odd pipeline. Each SPU can issue and complete up to two instructions per cycle - one per pipeline. For a wide variety of applications, the SPU can approach this theoretical limit.

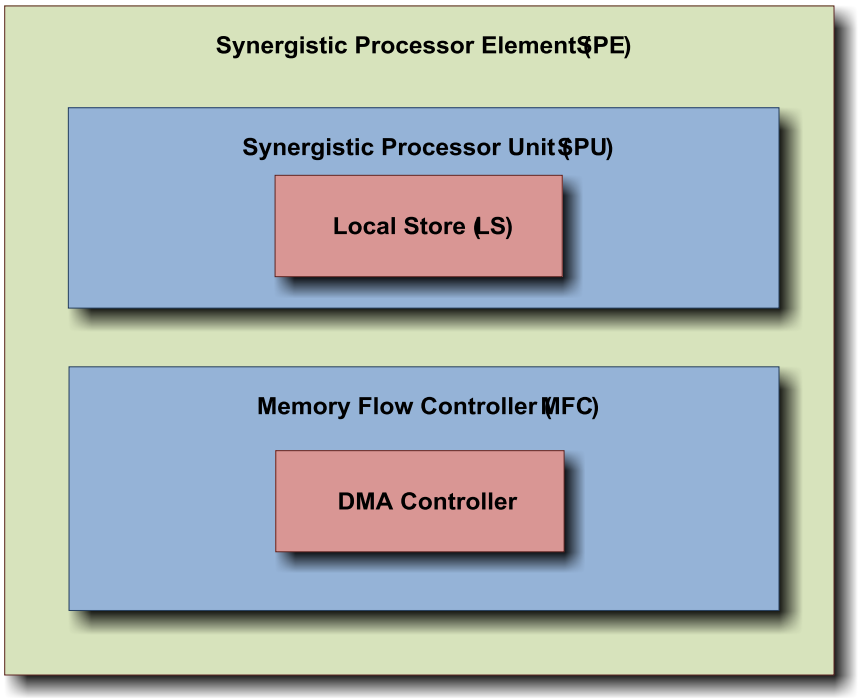

Figure 3: Synergistic Processor Element (SPE) block diagram

Unlike the PPE, the SPEs do not have caches. Instead, they have the LS that only they can see. All code and data for the SPU must be stored within this 256K local area. In fact, the SPUs cannot “see” the rest of the chip's address space at all. They can't access each others' local stores nor can they access the PPE's caches or other on-chip or off-chip resources. Each SPU fetches instructions from its own LS and it loads and stores data from and to its own LS, it cannot access main memory directly, but it has to issue DMA commands to the MFC to bring data into the LS or write results back to main storage (main memory, other SPEs’ LS, and memory-mapped registers). In effect LS works as a “second level” register file which provides a deterministic operating environment for the SPEs. The lack of caches and the presence of the LS

contribute to the deterministic performance because cache misses are not a factor in their performance.

Figure 4 shows the Synergistic Processor Element architecture.

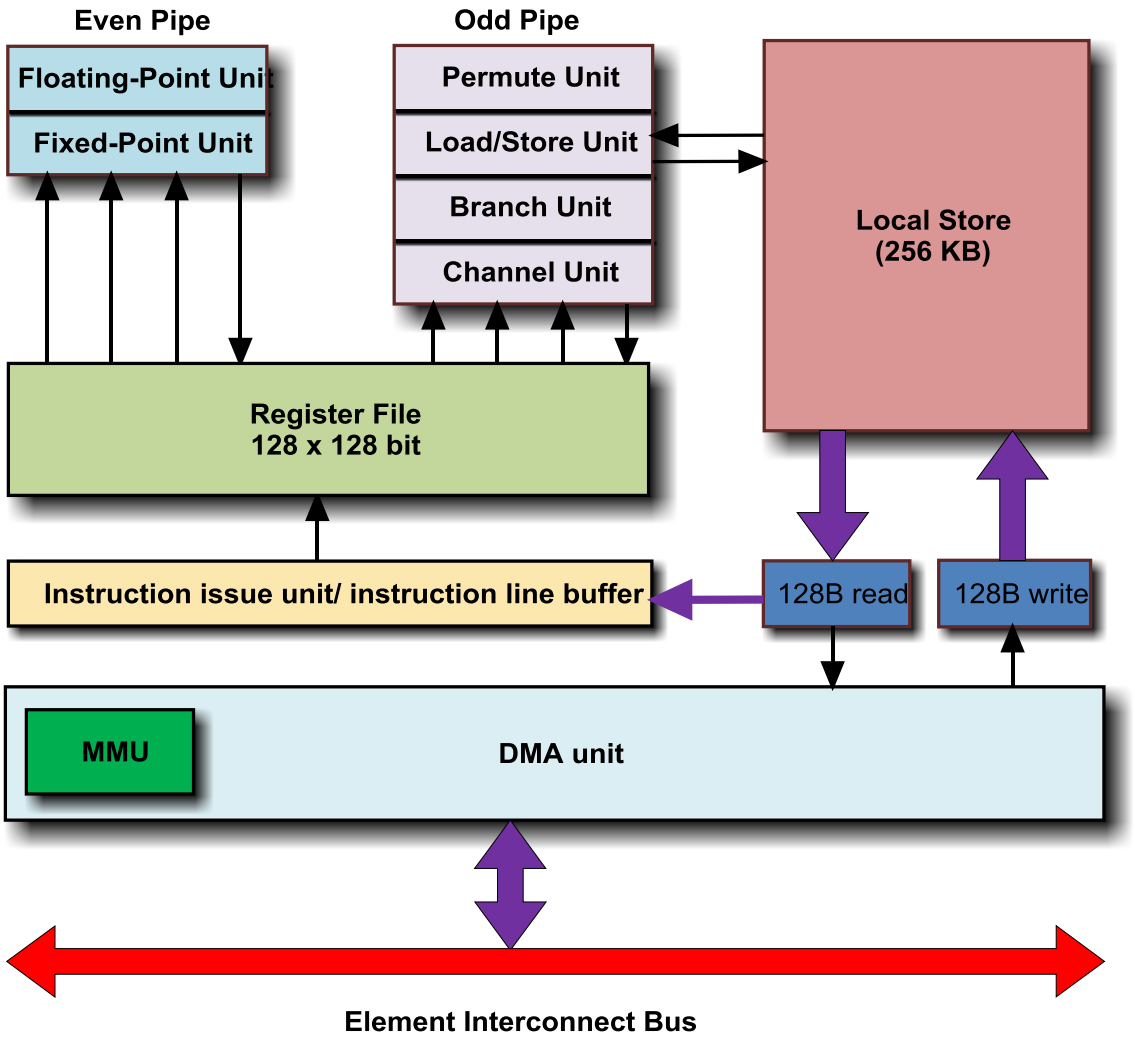

Figure 4: Synergistic Processor Element Architecture

The MFC contains a DMA controller that supports DMA transfers. Programs running on the SPU, the PPE, or another SPU, use the MFC's DMA transfers to move instructions and data between the SPU's LS and main storage. The MFC interfaces the SPU to the EIB (see *2.1.3*), implements bus bandwidth-reservation features, and synchronizes operations between the SPU and all other processors in the system. To support DMA transfers, the MFC 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. This autonomous execution of MFC DMA commands and SPU instructions allows DMA transfers to be conveniently scheduled to hide memory latency.

Synergistic processing clearly drives Cell's performance. Offloading as much as possible computations to the SPEs is the key to unleash the Cell computational power. But keeping all eight SPEs "fed" with data and parallelizing the code to run on all eight SPEs is the main challenge of programming the Cell processor.

2.1.3 Element Interconnection Bus

The *Element Interconnection Bus* (EIB) [21], [1], [15] is a communication bus internal to the Cell processor which connects the various on-chip system elements: the PPE processor, the memory controller (MIC), the eight SPE coprocessors, and two off-chip I/O interfaces, for a total of 12 participants. The EIB also includes an arbitration unit which functions as a set of "traffic lights", **Figure 5** shows the EIB.

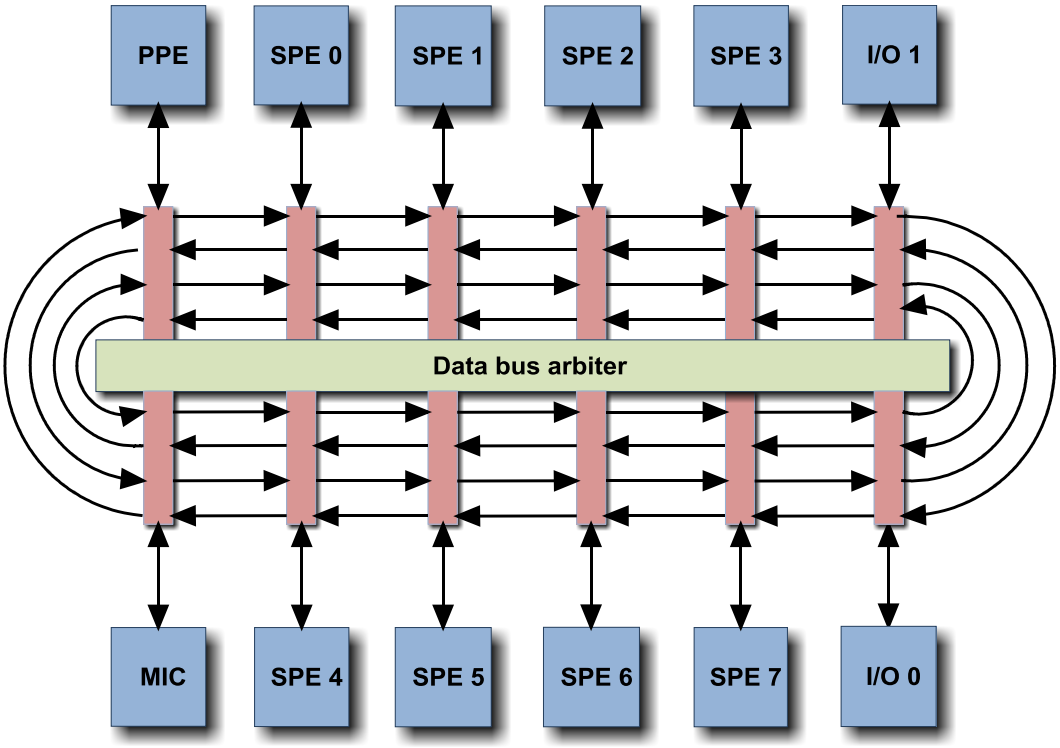

Figure 5: Element Interconnection Bus

The EIB data network consists of four 16-byte-wide data rings: two running clockwise, and the other two counterclockwise. Each ring potentially allows up to three concurrent data transfers, as long as their paths don't overlap. To initiate a data transfer bus elements must request data bus access. The EIB data bus arbiter processes these requests and decides which ring should handle each request. The arbiter always selects one of the two rings that travel in the direction of the shortest transfer. The arbiter also schedules the transfer to ensure that it won't interfere with other in-flight transactions. To minimize stalling on reads, the arbiter gives priority to requests coming from the memory controller.

The EIB operates at half the processor-clock speed. Each EIB unit can simultaneously send and receive 16 bytes of data every bus cycle. The EIB supports a peak bandwidth of 204.8 GB/s for internal transfers among the SPEs. The memory interface controller (MIC) provides a peak bandwidth of 25.6 GB/s to main memory. The I/O controller provides peak bandwidths of 25 GB/s inbound and 35 GB/s outbound.

It's clear that the EIB is one of the most important parts of the Cell design; it doesn't do processing itself but has to contend with potentially hundreds of Gigabytes of data flowing through it at any one time to many different destinations.

2.2 PlayStation 3

Currently the easiest and the cheapest way to gain access to a Cell processor is the Sony PlayStation 3 (PS3) [2]. As mentioned before, Cell processor was originally designed for PS3 and the vision was to achieve 1,000 times the performance of PlayStation 2. Due to the need of access to the Cell's computational power a Linux based operating system designed to run on PS3. The need for real Cell hardware mainly derives from the fact that IBM's Cell simulator is very slow. Today anybody can have access to the Cell processor by just installing an OS on PS3 and using it as a normal PC with high capabilities. Although PS3 is an easy solution it may not be the best, PS3 has some limitations on the performance of Cell. The main limitations are the small memory, only 256 MB and the availability of only six SPEs out of eight. One of the eight SPEs is disabled at the hardware level due to yield reasons and another SPE is reserved for use by the PS3's operating system. Apart from these limitations PS3 remains a good choice for anybody who wants to have its own Cell processor.

2.2.1 Operating System

The PS3 is shipped with an operating system called Game OS but is capable of running Linux OS if installed on the console's hard drive. The Linux operating system runs on the PS3 on top of a *virtualization layer*, also called *hypervisor*, the Game OS. This means that all the hardware is accessible only through the hypervisor calls. The hardware signals the kernel through virtualized interrupts. The interrupts are used to implement callbacks for non-blocking system calls. The Game OS permanently occupies one of the SPEs and controls access to the hardware. A direct consequence of this is larger latency in accessing hardware such as the network card. Even worse, it makes some hardware inaccessible like the accelerated graphics card.

At this point, there are numerous distributions that have official or unofficial support for PS3. The distributions that are currently known to work on PS3 (with varying levels of support and end-user experience) include:

- Fedora Core 7 [22],
- YellowDog 6.0 [23],
- Gentoo PowerPC 64 edition [24],
- Debian [25].

All the distributions mentioned include Sony-contributed patches to the Linux kernel-2.6.16 to make it work on PS3 hardware and talk to the hypervisor. However, the Linux kernel version 2.6.20 has PS3 support already included in the source code without the need for external patches.

2.2.2 Memory System

The memory system is built of dual-channel Rambus Extreme Data Rate (XDR) memory. PS3 provides a modest amount of memory of 256 MB, out of which approximately 200 MB is accessible to Linux OS and applications. The memory is organized in 16 banks. Real addresses are interleaved across the 16 banks on a naturally aligned 128-byte (cache line) basis. Addresses 2 KB apart generate accesses to the same bank. For all practical purposes the memory can provide the bandwidth of 25.6 GB/s to the SPEs through the EIB, provided that accesses are distributed evenly across all the 16 banks.

2.2.3 Network Card

The PS3 has a built-in GigaBit Ethernet network card. However, unlike standard PC's Ethernet controllers, it is not attached to the PCI bus. It is directly connected to a companion chip. The network card has a dedicated DMA unit, which allows making data transfer without PPE's intervention. One of many advantages of GigaBit Ethernet is the possibility of increased frame size – so called Jumbo Frames. It can increase available bandwidth by 20% in some case and significantly decreases processor load when handling network traffic.

2.2.4 Graphics Card

PS3 features special edition from NVIDIA and 256 MB of video RAM. Unfortunately, the virtualization layer does not allow access to these resources. At issue is not as much accelerated graphics for gaming as is off-loading of some of the computations to GPU and scientific visualization.

CHAPTER 3

Bayesian Intrapersonal/Extrapersonal Classifier Algorithm

This section presents the face recognition algorithm that has been ported on the Cell processor. The *Bayesian Intrapersonal/Extrapersonal Classifier* (BIC) is an algorithm based on the probabilistic matching techniques proposed by Moghaddam and Pentland for face recognition [10] . The design is using the ANSI C implementation of the BIC algorithm which is part of the *Colorado State University* (CSU) *Face Identification Evaluation System*, version 5.0. The methodology, the basic steps of algorithm as well as matters of algorithm input and output are introduced in this chapter.

3.1 CSU Face Identification Evaluation System

The Colorado State University (CSU) Face Identification Evaluation System [8], [9] provides standard face recognition algorithms and standard statistical methods for comparing face recognition algorithms. The system includes standardized image pre-processing software, four distinct face recognition algorithms, analysis software to study algorithm performance, and UNIX shell scripts to run standard experiments. All code is written in ANSI C. The four algorithms provided are *Principle Components Analysis* (PCA), a combined *Principle Components Analysis* and *Linear Discriminant Analysis* algorithm (PCA+LDA), a *Bayesian Intrapersonal/Extrapersonal Classifier* (BIC), and an *Elastic Bunch Graph Matching* (EBGM) algorithm. Our interested is only for the BIC algorithm, which is the most complex among the four algorithms; a detailed description of the BIC algorithm follows in the paragraph *3.3*.

3.2 The Face Recognition Problem

2D face recognition has been a popular and challenging research area since last decade. It arise general interests in computer vision, image analysis, psychology, etc.

The problem can be approached from two sides, identification and verification. The identification problem is: given a set of face images with labeled identity (the database) and a set of unlabeled face images (the probe), identify the person or persons in the probe images. The related verification problem is: given a novel image of specific person, confirm whether the person is or is not who they claim to be. Many works have been proposed for this problem till now. Below it is briefly introduced the method based on Bayesian theory.

3.3 BIC Algorithm

In traditional classifiers, face images are projected directly into a compressed subspace, under the assumption that images of a single person will map to a tight cluster of points. Conversely it is expected that projections of images of different subjects will be widely separated, Moghaddam and Pentland proposed an alternative [10], [11]. Their classifier defines the subspace in a different way, rather than treading face images as points in a face subspace, they look at the space spanned by the *difference* between two face images. The difference image for two face images is the signed arithmetic difference between respective pixels in the source images. Such difference images fall into two distinct classes:

- *Intrapersonal difference images* are those derived from two images of the same subject.
- *Extrapersonal difference images* are those derived from two images of different subjects.

Below **Figure 6** shows intrapersonal and extrapersonal difference images.

Figure 6: Intrapersonal/Extrapersonal difference images

Moghaddam and Pentland suggest that intrapersonal and extrapersonal difference images form distributions that are approximately Gaussian. As shown in **Figure 7** their classifier matches probe images to stored images by computing the likelihood that the corresponding difference images came from the subspace of intrapersonal rather than extrapersonal. The likelihood is computed in two ways, by using *Maximum Likelihood* (ML) method or *Maximum a Posteriori* (MAP) method.

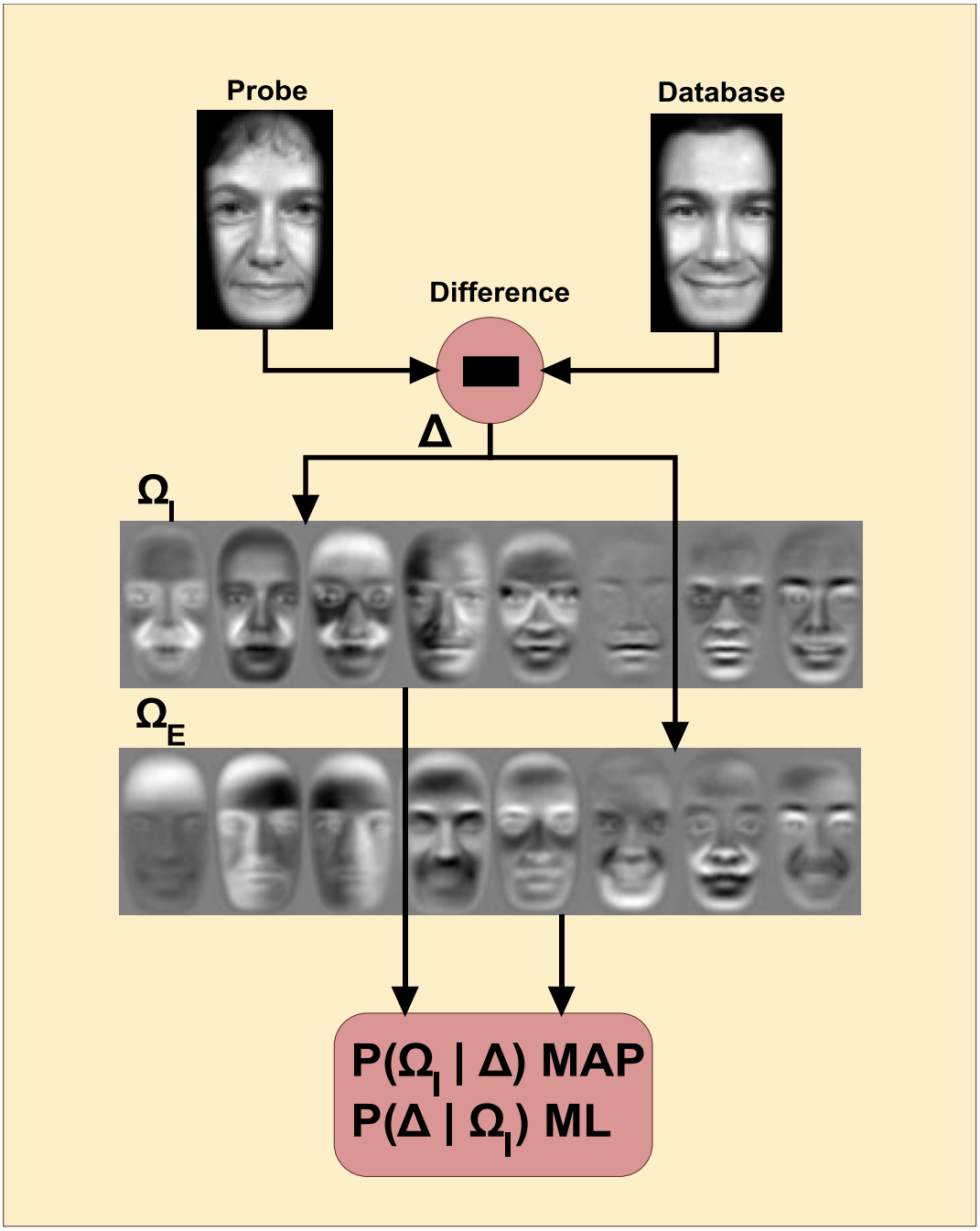

Figure 7: Bayesian similarity. The difference image is projected through both subspaces (intra/extra) in order to obtain the two likelihoods.

The CSU Face Identification Evaluation System implements the BIC algorithm in three main steps, *preprocessing, Bayesian training* and *Bayesian project* [12], [8]. The implementation uses the FERET database and it classifies all the images of the input set, that is to say it uses each image of the input set as a probe, it classifies it and goes to the next. The flow chart of the CSU implementation of the BIC is shown in **Figure 8**.

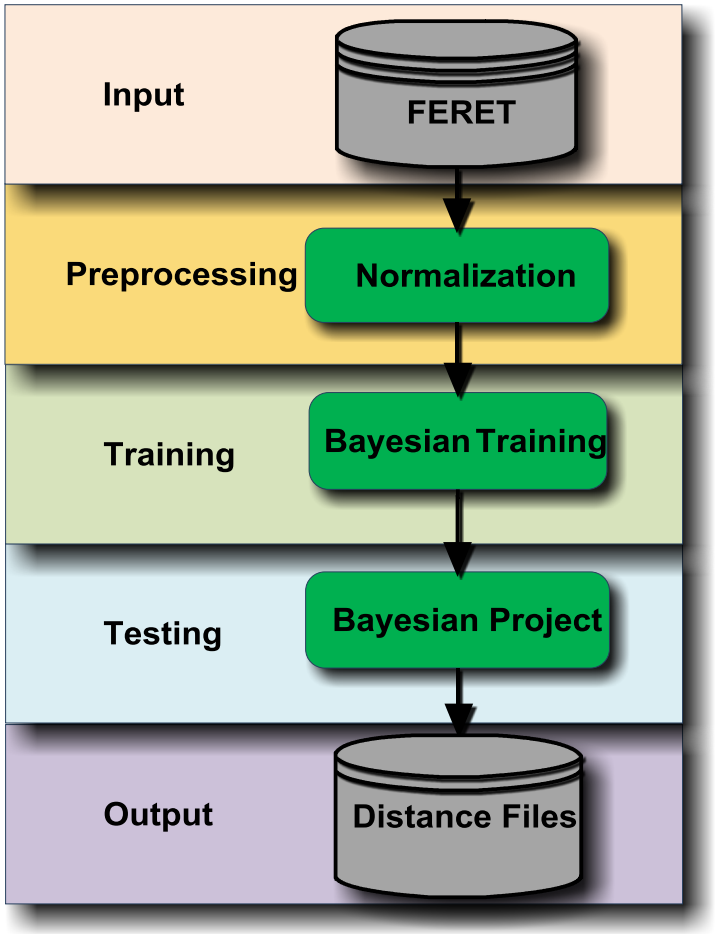

Figure 8: Bayesian Intrapersonal/Extrapersonal Classifier

3.3.1 Preprocessing

Preprocessing is conducted at the first step of the algorithm. The process performs five steps in converting a FERET image (*see 3.4*) to a normalized image [26]. The normalization schedule is:

- *Integer to float conversion* - Converts 256 gray levels into floating point equivalents.
- *Geometric normalization* – Lines up human chosen eye coordinates.

- *Masking* – Crops the image using an elliptical mask and image borders such that only the face from forehead to chin and cheek to cheek is visible.
- *Histogram equalization* – Equalizes the histogram of the unmasked part of the image.
- *Pixel normalization* – scales the pixel values to have a mean of zero and a standard deviation of one.

For an example see **Figure 9**

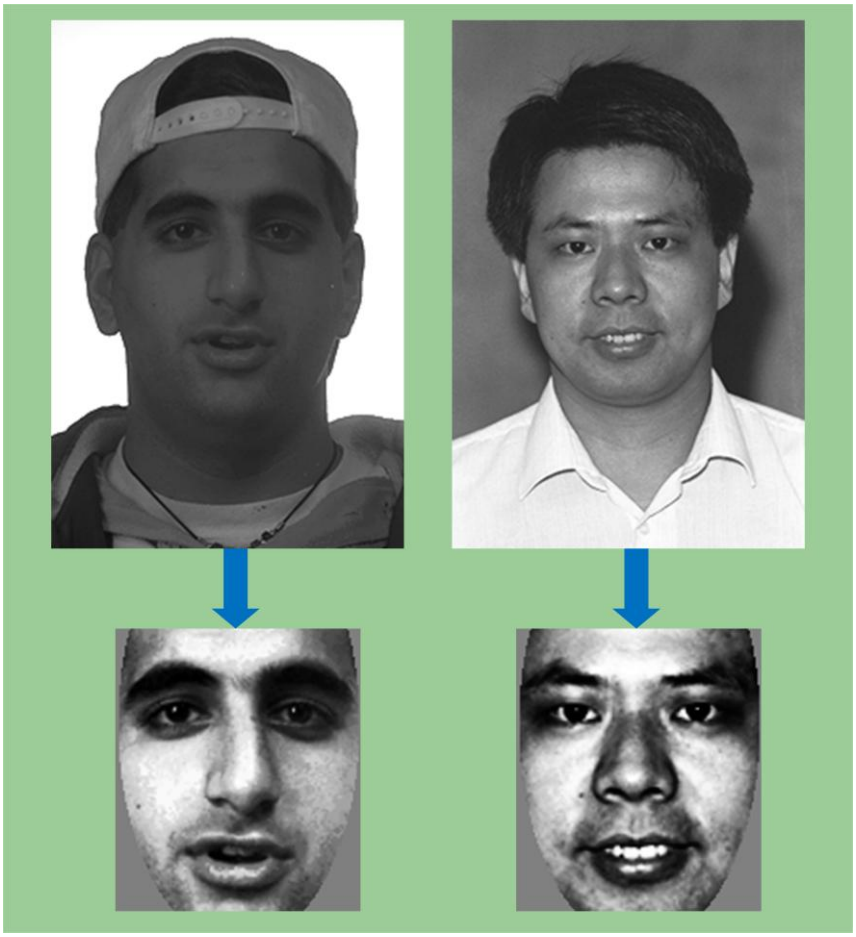

Figure 9: Image Preprocessing

3.3.2 Bayesian Train

Before the algorithm can actually be used, parameters for the intrapersonal and extrapersonal densities must be estimated using training data. The training step outputs two training files, one for each subspace. The training file contains a description of the training parameters, the mean of the training image, the eigenvalues and a set of basis vectors for the subspace [27]. This training is

performed twice: once with the Intrapersonal subspace, and once with the Extrapersonal subspace. At the end, the subspaces are stored into two different files: *bayesian.intra* and *bayesian.exta*.

3.3.3 Bayesian Project

The main step of the algorithm is the Bayesian projection. The code projects the feature vectors onto each of the two sets of basis vectors and then computes the probability that each feature vector came from one or the other subspace. The output is a set of distance files containing the distance from each image to all other images. As we mentioned before the similarities may be computed using the maximum a posteriori (MAP) or the maximum likelihood (ML) methods. From a practical standpoint, the ML method uses information derived only from the intrapersonal images, while the MAP method uses information derived from both distributions. The below equations are extracted from Marcio Luis Teixeira's thesis on the Bayesian Intrapersonal/Extrapersonal Classifier [12], [10], [27].

First, let us introduce the probability of distance image *Δ* belonging to a class:

$$\hat{P}\ \Delta\,|\,\Omega\ \equiv \frac{\exp\left(-\frac{1}{2}\sum_{i=1}^{M}\frac{y_i^2}{\lambda_i}\right)}{2\pi^{\ \mathrm{M}/2}\prod_{i=1}^{M}\lambda_i^2} * \frac{\exp\left(-\frac{1}{2\rho}\varepsilon^2\ \Delta\right)}{2\pi\rho^{\ m-M\ /2}}$$

Where the parameters are:

λ_i	i[th] eigenvalue (from training)
y_i	Projection of a difference image onto the corresponding subspace
M	Number of eigenvalues kept
m	Dimension of the original data
$\rho = \frac{1}{T-M}\sum\lambda_i$	Mean of eigenvalues
T	Number of training images used to estimate $\hat{P}\ \Delta\,\|\,\Omega$
$\varepsilon^2\ \Delta$	Reconstruction error

The Maximum Likelihood classifier is then defined by:

$$S_{ML} = \frac{\varepsilon^2(\Delta)}{\rho} + \sum_{i=1}^{M} \frac{y_i^2}{\lambda_i}$$

The Maximum a Posteriori classifier uses Bayes rule to estimate the a posteriori probability of Δ to the Intrapersonal or Extrapersonal class. This probability is:

$$S_{MAP} = \hat{P}(\Delta|\Omega_I) = \frac{\hat{P}(\Delta|\Omega_I) \cdot \hat{P}(\Omega_I)}{\hat{P}(\Delta|\Omega_I) \cdot \hat{P}(\Omega_I) + \hat{P}(\Delta|\Omega_E) \cdot \hat{P}(\Omega_E)}$$

Where

Ω_I Intrapersonal subspace

Ω_E Extrapersonal subspace

3.4 Algorithm Input

The main input of the algorithm is a set of frontal facial images from the *Face Recognition Technology* (FERET) [26] database of the *National Institute of Standards and Technology* (NIST). The set of images from the database to be used from the algorithm are stored in an image list file with *.srt* extension. The algorithm is initially reading and processing all the FERET database images included in the image list file. The preprocessing step reads the *.pgm* extension files from the FERET database and creates the normalized images with *.sfi* extension. These set of normalized images consist the input of the Bayesian Train and Bayesian Project steps.

3.5 Algorithm Output

The algorithm produces a distance matrix for all of the images in the testing list. This matrix is split up into distance files. One file is produced for every image in the list. Each line in these file contain the name of another image and the distance to that image. The file has the same name as the probe image and is placed in a distance directory. The algorithm assumes that *smaller distances are a closer match*. Two sets of distances files are created by the algorithm, one for each method it uses to compute the distance, maximum a posteriori (MAP) or the maximum likelihood (ML) method.

CHAPTER 4

Implementation

In this chapter it is briefly described the process of enabling the CSU implementation of the BIC algorithm to the Cell processor. It refers with details in the data partitioning procedure, the levels of parallelism, the data transfers and the code optimizations. It also describes some of the problems that were encounter during the development and the solutions that were provided. The main purpose of this chapter is to explain the overall development flow that was followed in our implementation.

4.1 The Programming Model

The programming model that was chosen for our implementation was the *function offload model* [28], [29] .The function offload model is the quickest way to effectively use the Cell processor with an existing application. In this model, the main application runs on the PPE and calls selected procedures to run on one or more SPEs. In this programming model, the SPEs are used as *accelerators* for certain types of performance-critical functions, hotspots. This model replaces complex or performance-critical functions invoked by the main application with functions offloaded into one or more SPEs, without changing the main application logic at all. The original performance-critical function is optimized and recompiled for the SPE environment and an SPE-executable program is being created.

Currently, the programmer statically identifies which functions should execute on the PPE and which should be offloaded to SPEs by utilizing separate source and compilation for the PPE and SPE components. It is also programmer's responsibility to manually partition and schedule the work to one or more SPEs. This model was selected because we already had an implementation of the BIC algorithm and we just wanted to improve the performance of the algorithm with parallelism. The quickest and possibly easiest way to do this was to speed-up the computation-intensive functions of the algorithm with the function offload model.

4.2 The Application Enablement Process

The process of enabling an application on Cell processor can be incremental and iterative [29]. It is incremental in the sense that the hotspots of the application should be moved progressively off the PPE to the SPE. This process is iterative as for each hotspot, the optimization can be refined at the SIMD, synchronization and data movement levels until satisfactory levels of performance are obtained. As for the starting point, a thorough profiling of the application on a general purpose system will give all the hotspots that need to be looked at. Then, for each hotspot, a multi-SPE implementation can be written with all the data transfer and synchronization between the PPE and the SPE. Once this first implementation is working, the tuning is turned to the SPE code. The last two steps can be repeated in a tight loop until the performance is good enough. The same process can be repeated for all the major hotspots. This enablement process is shown in **Figure 10**.

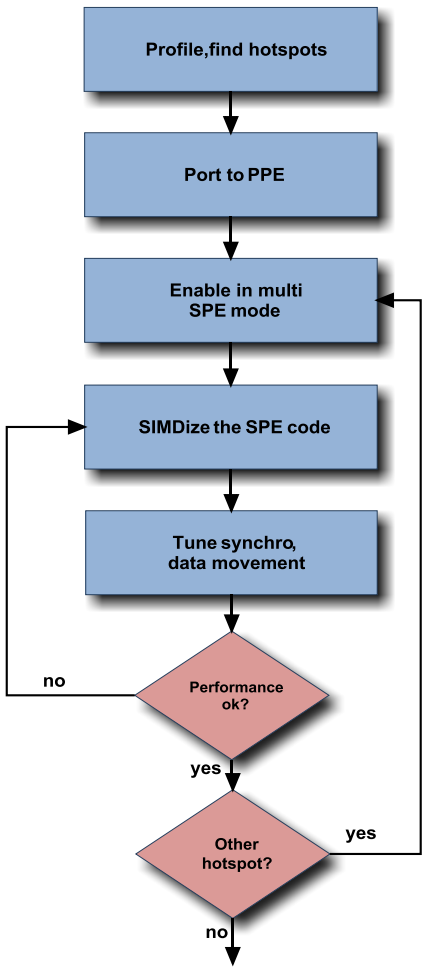

Figure 10: Application enablement process

4.3 Profiling

As it was mentioned above the first step for porting the application to the Cell processor was the profiling of the original code, to locate the computation-intensive functions. *Intel's Vtune Performance Analyzer 9.0 for Linux* was used to determine the most time-consuming functions for each of the three applications of BIC algorithm. **Table 1, Table 2** and **Figure 11** shows the execution profile for each of the three applications, preprocessing, Bayesian train and Bayesian project, the profiling was made on a P4 machine at 2.66GHz, with Ubuntu 7.10 OS. These results indicate that two of our applications spend more than *90%* of their execution time in two functions only, *multiplyMatrix* and *tranposeMultiplyMatrixL*. On the other hand, preprocessing distributes its execution time to many functions with none of them exceeding the *30%* of the execution time.

Process	Clockticks %
csuBayesianProject	53,10%
csuBayesianTrain	7,70%
csuPreprocessNormalize	4,62%
Other OS processes	34,58%

Table 1: BIC profiling

Function	Bayesian Train	Bayesian Project	Preprocessing
multiplyMatrix	60,88%	0%	14,79%
transposeMultiplyMatrixL	33,85%	90,90%	0%
centerThenProjectImages	0%	3,02%	0%
interpLinear	0%	0%	28,44%
histEqualMask	0%	0%	10,67%
Others	5,27%	6,08%	46,1%

Table 2: BIC applications profiling

This is a useful fact for an implementation on the Cell processor, as it implies that significant speed up might be obtained for these applications by only offloading these functions to the SPUs. Especially the speed-up of the *tranposeMultiplyMatrixL* function is what concerns us more because is the most time-consuming part of the total BIC application. The rest of the enablement process is based on this results and the main objective is the speed-up of *tranposeMultiplyMatrixL* and *multiplyMatrix* functions.

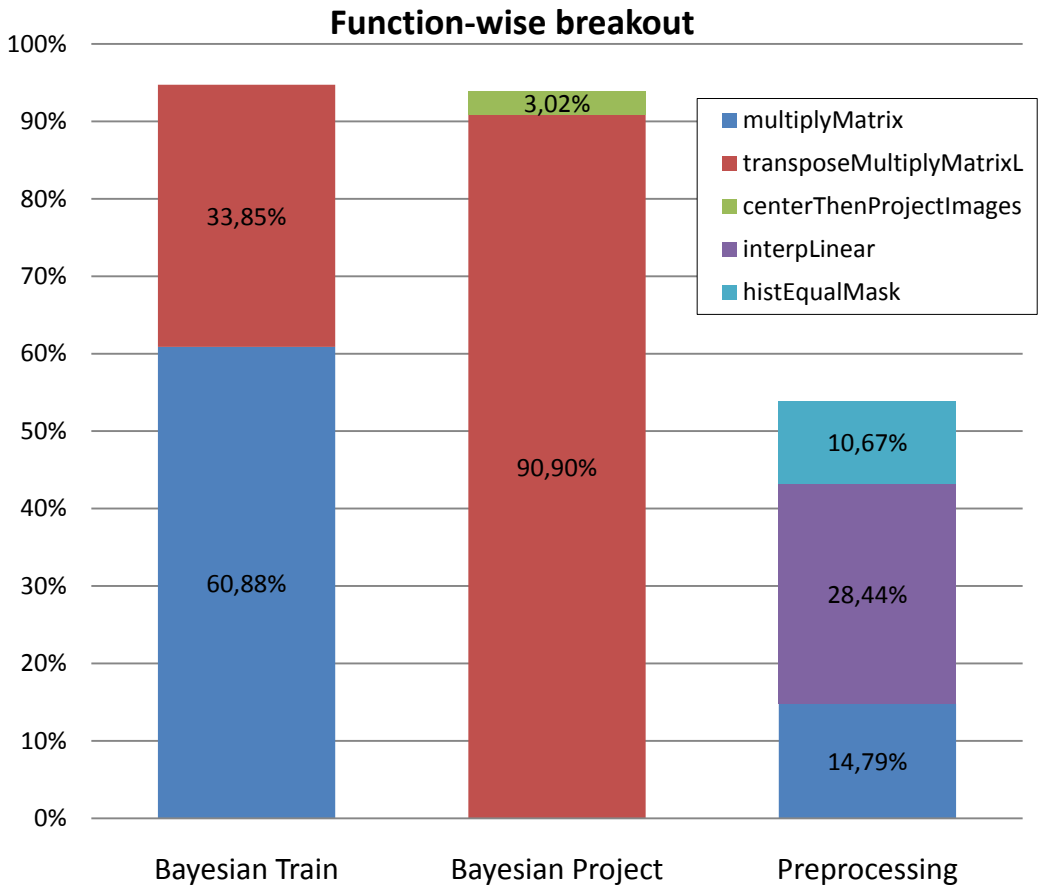

Figure 11: Function-wise breakout of BIC applications

4.4 Matrix Multiplication

Matrix multiplication and transpose matrix multiplication is a common operation in ML and MAP classifiers and also a computation-intensive procedure for the processors. The hard part of matrix multiplication (and transpose multiplication) is the evaluation of the triple loop, for example a multiplication of the form *C = A x B*, where A and B are *n x n* matrices can be structured as follows:

```
for (i=1; i<=n; i++){
  for (j=1; j<=n; j++){
    for (k=1; k<=n; k++){
      c[i,j]=c[i,j]+a[i,k]*b[k,j];
    }
  }
}
```

This structure involves $2n^3$ *operations*, carried out by a total of $6n^3$ *instructions*, three loads, an addition, a multiplication and a store are required each time the computation is evaluated. Another important thing that makes even harder the matrix multiplication is that all the elements of the matrices are *double precision* (64-bit) floating-point numbers.

The implementation of these functions in the CSU BIC algorithm uses *column major* form for storing the data in the main memory. That is to say that the elements of a column are stored sequentially in the main memory, as shown in **Figure 12**. As the CSU BIC algorithm implementation uses matrices not only in these two functions, the implementation for the SPEs depends on the structures and type definitions that CSU implementation uses. This makes a lot easier the compatibility of the two offloaded functions with the rest of the code.

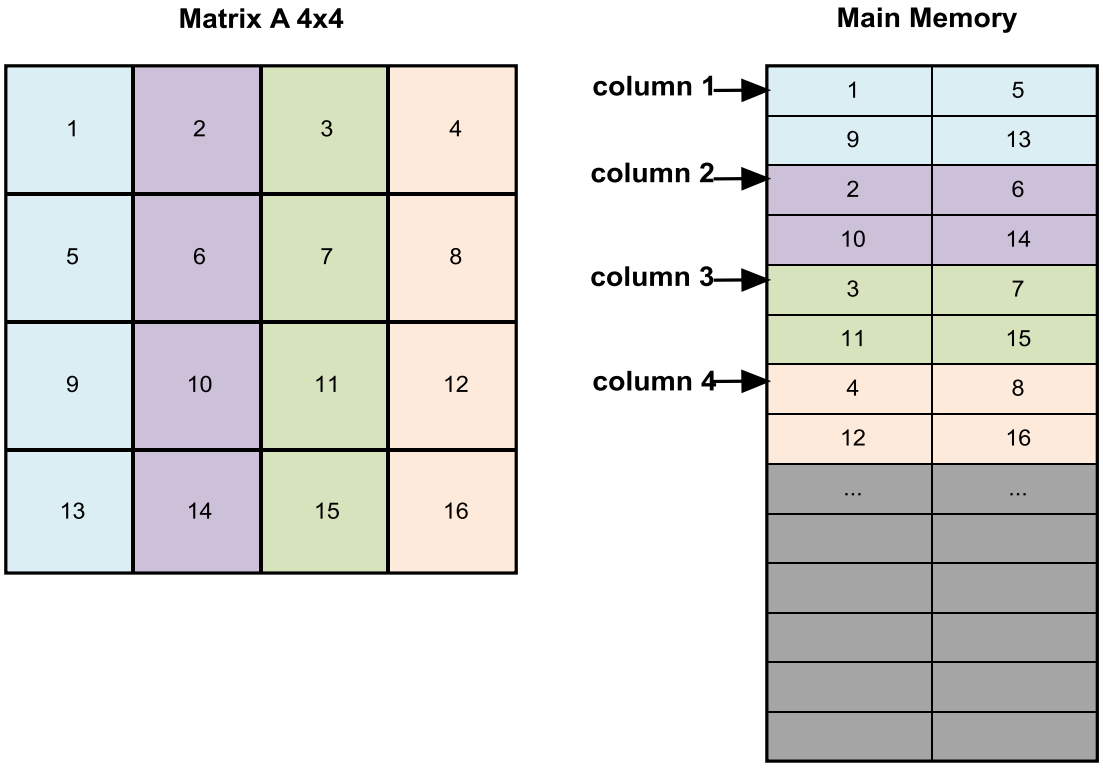

Figure 12: Column-major form

4.5 Data Flow Analysis

After the functions that would run on SPEs were located, a data flow analysis was required to determine the amount of data that had to be transferred to the SPEs LS. Both functions have two input arguments, matrix A and matrix B and return the result into another matrix C. The size of matrices A, B and C is not always the same, all the function calls in the original source code were studied carefully and was discovered that there are three kinds of multiplications and transpose multiplications. The following table, **Table 3** shows the dimensions of matrices for each case and the amount of space needed. Besides the multiplication types, **Table 4** shows which application uses each type of multiplication. The application of preprocessing is not included in the next table because the original implementation of preprocessing was not changed.

Matrix Multiplication	A	B	C	Data size (MB)
Transpose	19500x100	19500x100	100x100	31.28
Transpose	19500x59	19500x1	59x1	9.36
Normal	19500x100	100x100	19500x100	31.28

Table 3: Matrices dimensions and data size

Matrix Multiplication	A	B	C	Application
Transpose	19500x100	19500x100	100x100	Bayesian Train
Transpose	19500x59	19500x1	59x1	Bayesian Project
Normal	19500x100	100x100	19500x100	Bayesian Train

Table 4: Matrices dimensions and applications

From the previous data analysis it was obvious that was unable to perform at once these multiplications at the SPEs due to the limited size of LS at the SPEs. This observation led us to the conclusion that in order to execute the multiplication at the SPEs it was necessary to partition the data into smaller pieces. Moreover the matrices had to split into submatrices of the appropriate size, calculate the partial results at the SPEs and combine them to estimate the final result of the initial matrices. This necessity for data partitioning offered us and the *first level parallelism* of the implementation, parallel submatrices multiplication on the SPEs in order to estimate one of the large matrix multiplications. More details on the implementation of data partitioning and the first level parallelism follows in the next paragraph.

4.6 Development

According to the data layout that was defined in the previous paragraph the development started from the data partitioning problem that occurred. Apart from the two main functions *tranposeMultiplyMatrixL* and *multiplyMatrix*, auxiliary functions were created for partitioning the matrices and merging the submatrices. As the implementation was based on an existing application and on existing source code, these two functions were implemented separately from rest application. When the correctness of the two functions was verified they were combined with the rest source code of the application. This approach has the benefits of fast development and easy debugging as it has small size of source code. After the data partitioning problem was solved we had to deal with the data transfer to the SPEs in order to execute the two functions. At first only one SPE was used and then the implementation was extended on multiple SPEs. A detail development flow chart is shown on **Figure 13**.

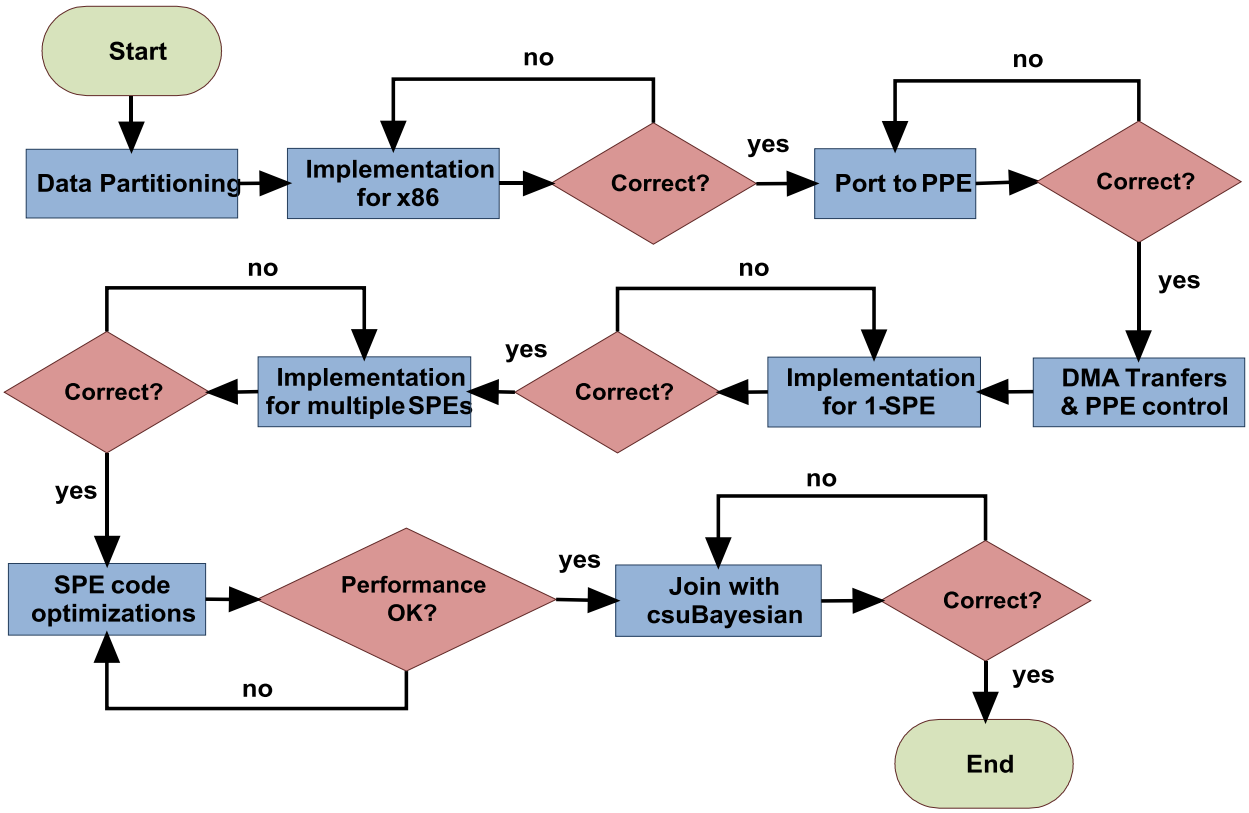

Figure 13: Development flow

All the code was developed with the use of IBM's *Cell SDK 2.1* and *Full-System Simulator*. For the final stage of the development it was used a PS3 with Yellow Dog Linux 6.0 OS.

4.6.1 Data Partitioning

From the data flow analysis it was discovered that data partitioning consisted an absolute necessity. A way should be found to resolve this problem in order to execute multiplications at the SPEs. The main idea is that a matrix multiplication of large matrices can be calculated by doing the partial multiplications of the appropriate submatrices and combine the partial results to a final result. Two methods were implemented for data partitioning because the first implementation had a significant overhead and it was necessary to avoid it.

At first two functions were implemented at the PPE side to partition the matrices to smaller submatrices. The two functions are *partitionMatrixVertical* and *partitionMatrixHorizontal*, these functions break up the matrices in submatrices in vertical or horizontal direction. The dimensions of the submatrices are defined by the user, but the matrix row dimension must be a multiple of the submatrix row dimension, the same limitation holds and for the column dimension. The following figure, **Figure 14**, show an example of horizontal and vertical partitioning for submatrices dimensions 5x5.

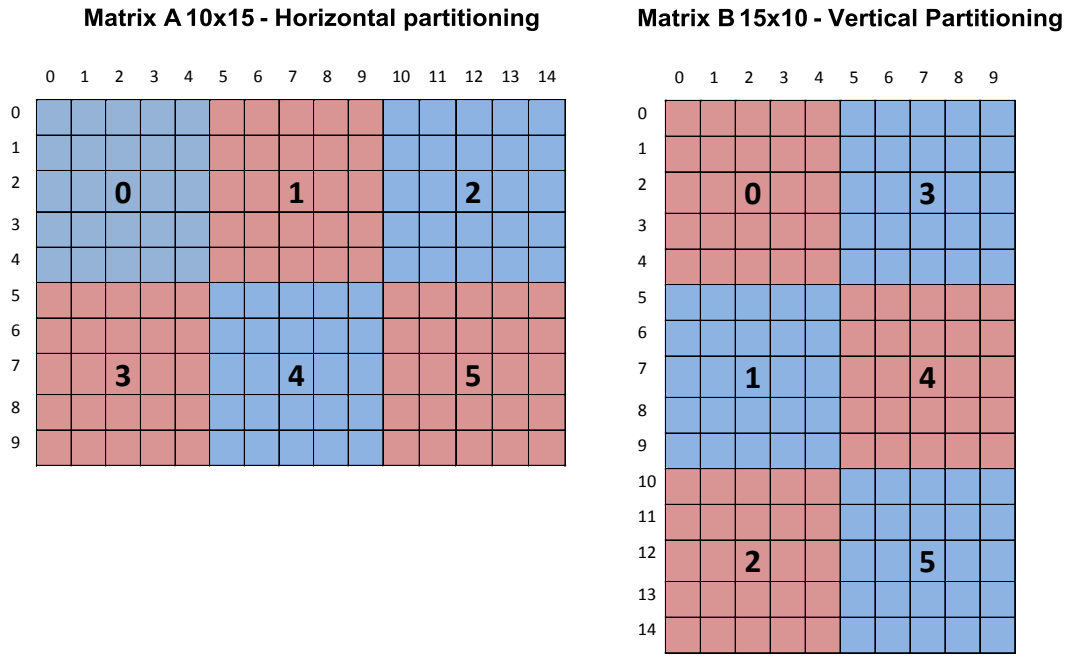

Figure 14: Horizontal and vertical partitioning

Submatrices of proper dimensions can be created for both matrices with the use of functions *partitionMatrixVertical* and *partitionMatrixHorizontal*. The problem with this approach is that in order to create the submatrices, blocks of memory had to be copied from multiple positions of the original matrix, for each submatrix. This copy procedure was adding an important amount of time to the final execution time and this overhead had to be avoided. More specifically the *memcpy* function was used for the data copy and it proved to be very slow on the PPE and we had to find an alternative method.

To avoid this overhead it was decided not to partition the data at the PPE but to construct the appropriate submatrices at the SPEs. This implementation has the advantage of not using the *memcpy* function at the PPE side and the disadvantage that it needs more DMA transfers to be initiated from the SPEs. The partitioning procedure of the second method is in fact the same with the first but instead of being implemented at the PPE side and temporary storing the submatrices is implemented directly to the SPEs. The first method was creating the submatrices at the PPE side and was doing sequential DMA transfers starting from the address of each submatrix. In the second method the data of each submatrix were not grouped together, but they were transferred to the SPEs column by column (because of the column-major form, figure 4.2) from the main memory until the submatrix was constructed in the LS. This means that as many DMA transfers as the columns of each submatrix had to be done. A more detailed analysis for the data partitioning method and the data transfers is discussed in paragraph 4.6.5.

The next step was to multiply these submatrices and calculate the overall result. This procedure is iterative and takes place to the SPEs; it multiplies the appropriate submatrices and stores the partial results back to the main memory. These partial results are added and merged properly to create the final result. For the merging process the functions *mergeMatrixVertical* and *mergeMatrixHorizontal* where created, these function implement the exact opposite of the partitioning process. Finally, the most suitable dimensions for the submatrices were defined for each case of multiplication that was showed previously on **Table 4** these dimensions are shown at **Table 5**

Matrix Multiplication	Submatrix A	Submatrix B	Submatrix C	Data Size (KB)
Transpose	100x100	100x50	100x50	160.00
Transpose	250x59	250x1	59x1	120.47
Normal	50x100	100x100	50x100	160.00

Table 5: Submatrices dimensions and data size

4.6.2 Implementation on x86 Architecture

The above procedure was first implemented on P4 machine at 2.66 GHz; this implementation can be considered as a *model* of the final implementation. A small program was developed only for matrix multiplication and transpose matrix multiplication with data partitioning as described. At this stage the SPEs are not being used so the first method for data partitioning was followed. This model program was mainly used to verify at this level the correctness of the data partitioning procedure. The program performs a multiplication of two matrices with the classic method (without data partitioning) and the same multiplication with data partitioning and subtracts the two result matrices to verify the result. A correct calculation should give a zero matrix after the subtraction of the two results. Unfortunately because the multiplication with submatrices changes the order of the additions, in comparison with the normal multiplication, the final result has a small difference from the classic method. This difference is located after the tenth decimal digit of the result. The precision of the implementation and how this inaccuracy affects us will be discussed in more depth in the next chapter.

4.6.3 Port to PPE

The next step of the development was the porting of the original CSU BIC algorithm and the model program that was described above to the PPE. This porting to the PPE was made to confirm the correct execution of the model program and of the complete algorithm, too, on the PPE. There were some problems with the porting of the CSU BIC algorithm implementation because the program was designed for little endian architecture and PPE is big endian. Some changes were made to resolve this problem and the application executed correctly to the PPE. This part of the development was important in order to gain familiarity with the *gcc-ppu* compiler and the necessary *makefiles* for the Cell processor [28], [30]. After this step the application was running on PPU and the next step was to begin offloading the functions to the SPEs. We let aside the CSU BIC algorithm implementation for PPE to continue the function offloading process with our model program for easier development.

4.6.4 PPE Control

Before it was able for the code to execute on SPEs the design had first to deal with matters of control and synchronization [29], [30]. As it was mentioned before the

function offload model uses the PPE to schedule and control the threads running on SPEs. These controls are based on *a fork-join* model; more specifically, the PPE creates the number of SPE threads that needs (fork) it also initiates the execution of the threads and when the threads are done, it joins all the independent execution flows and destroys the threads. While the threads are running the PPE can either continue execution, *asynchronous execution* or can wait the SPE threads to finish, *synchronous execution*. In the implementation synchronous execution was the only way because there were dependencies between the results from the SPEs and the following PPE code. A top level view of this procedure is shown in **Figure 15**.

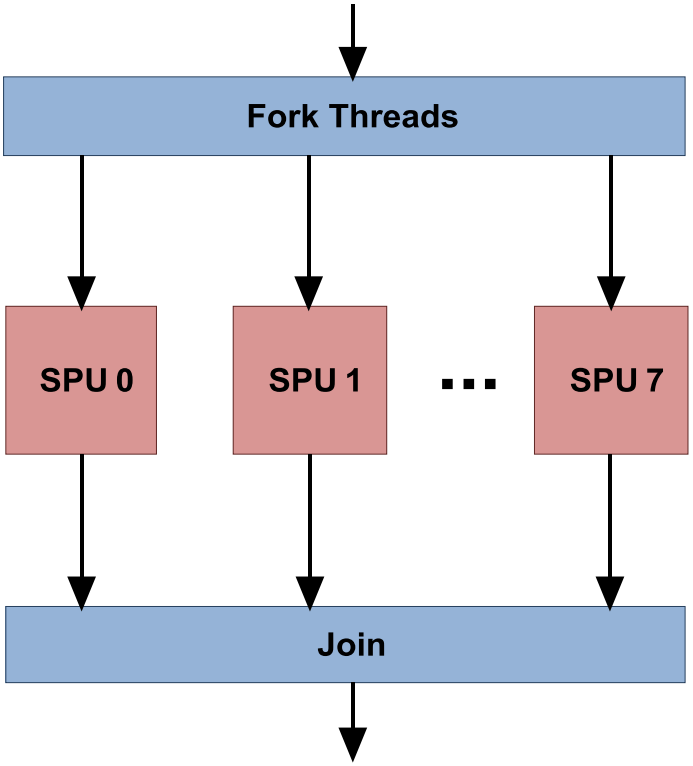

Figure 15: Fork-Join procedure

The fork-join procedure is implemented with the use of specific library calls provided by the SPE runtime management library [31] [32]. The following figure, **Figure 16**, shows very simply how PPE creates SPE context, loads a SPE program and executes the program from the current thread.

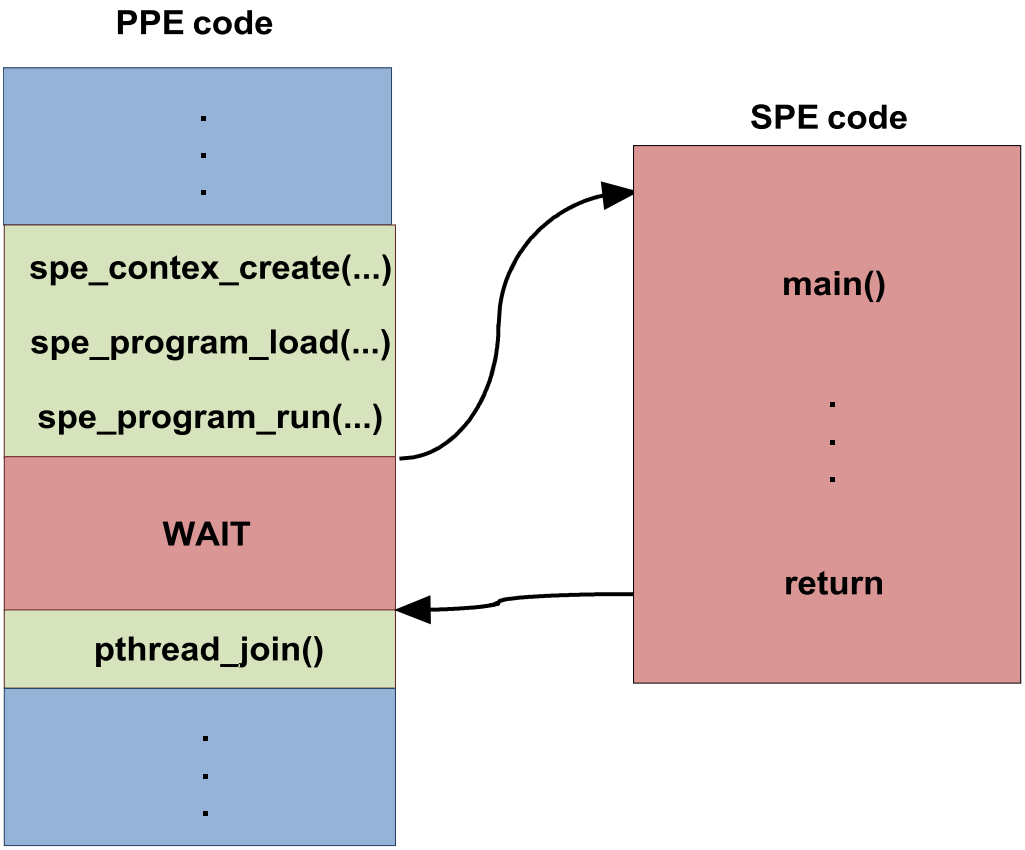

Figure 16: PPE control

The PPE control is a very important part of the design, as the total scheduling procedure consumes a significant amount of time compared with the total execution time. Some overhead from the scheduling process is unavoidable but it was managed to be reduced as much as possible. In order to reduce the overhead the context was created only once and the program was loaded to the SPEs only once, when it was possible. In the case of CSU Bayesian Project this was an important optimization because only one specific function was offloaded to the SPEs, so the context was created and the program was loaded to the SPEs only once. But in the case of CSU Bayesian Train there were two different programs to be executed on the SPEs so the appropriate program was load to the SPEs each time. Despite all the optimizations that were tried in the CSU Bayesian Project the overhead from the scheduling procedure remained very high mainly due to the fact that the program had many calls of the multiplication function and so many re-entrances to the SPEs. More details for the scheduling of CSU Bayesian Train and CSU Bayesian Project are presented in paragraph *4.6.8*.

4.6.5 DMA Transfers

According to the Cell architecture, SPEs can only use data from their LS; for the implementation this means that the two submatrices had to be transferred to the LS to be able executing the multiplication. In paragraph *4.6.1* it was explained how the data were partitioned in order to fit in the LS, now it will be explained how these data are transferred to the LS.

Cell supports two kinds of DMA transfers, PPE initiated and SPE initiated, in our implementation only SPE initiated DMA transfers were used. This choice was made because there are eight times more SPEs than PPEs and the number of cycles to initiate a transfer from the SPEs is smaller than the number of cycles to initiate the same transfer from the PPE. The data transferring process can be described from the following steps:

- SPU needs data.

1. SPU initiates DMA request for data.
2. DMA requests data from the memory.
3. Data is copied to the LS.
4. SPU can access data from the local store.

- SPU operates on data and then copies data from LS back to main memory in a similar process.

The above process looks very simple but there are two important limitations for the DMA transfers. The size of a single transfer is limited to *16KB* and the size can only be *1, 2, 4, 8, 16,* or a *multiple of 16 bytes*. So as it looks, for large data many DMA transfers had to be done and if the size of our data is not a multiple of 16, again extra DMA transfers had to be initiated. There are two categories of DMA commands the *put* and the *get*

- *put* commands move data from LS to main storage.
- *get* commands move data from main storage to LS.

The following figure, **Figure 17** describes the total procedure for data transfer to and from the LS. The circled numbers shown in the figure correspond to the steps of the data transferring process as it was defined above. The black arrows are for data transfer from the main memory to the LS and the red arrows are for the opposite process, data transfers from the LS to the main memory.

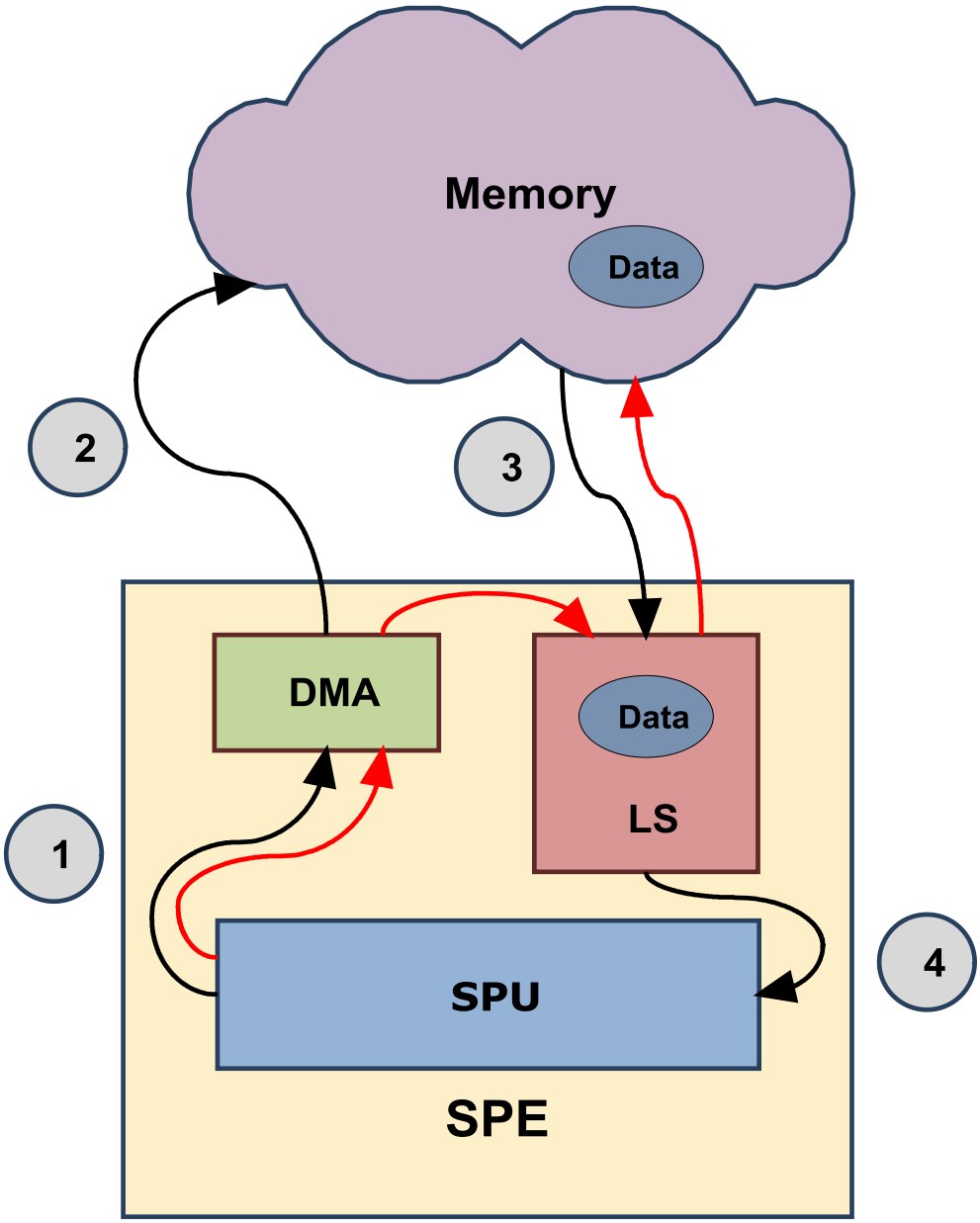

Figure 17: Data transfer to and from the LS

In our matrix multiplication problem the data that needed to be transferred were two submatrices from the main memory and the result of the multiplication back to the main memory. Multiple get commands were needed to fetch the submatrices from the main memory and multiple put commands to send the result back to the main memory. **Table 6** shows a summary of the required DMA transfers for each case of multiplication of the form *C = A x B.*

Submatrix	Type	Data Size (KB)	DMA – get (KB)	DMA – put (KB)
100 x 100	A	80	100 x 0,8	-
100 x 50	B	40	50 x 0,8	-
100 x 50	C	40	-	2 x 16 1 x 8
250 x 59	A	118	59 x 2	-
250 x 1	B	2	1 x 2	-
59 x 1	C	0.472	-	1 x 0,464 1 x 0,008
50 x 100	A	40	50 x 0,8	-
100 x 100	B	80	100 x 0,8	-
50 x 100	C	40	-	2 x 16 1 x 8

Table 6: DMA transfers summary

The data that should be transferred to each SPE are depended on the matrix multiplication type and the dimensions of the submatrices. Due to the limited LS size a multiplication of two submatrices must be completed and write the result to the main memory before the allocated space can be available for the new data.

Figure 18 shows the sequence of the DMA transfers for a simple matrix multiplication. The figure clearly shows the parallelism through the data partitioning; the two SPEs execute multiplications in parallel. The execution between the SPEs is independent but each SPE has to free its LS before getting new data and starting the next multiplication. In this case first C0 and C6 are calculated then C1 and C7 and so on. After all the calculations finish, the submatrices are properly joined to construct the final result matrix. In the previous example, the addition of submatrices C0 – C2 creates the upper left part of the result, C3 – C5 the upper right, C6-C8 the lower left and C8-C11 the lower right. The construction of the result matrix is very simple and it takes place at the PPE side after all the SPEs have finished.

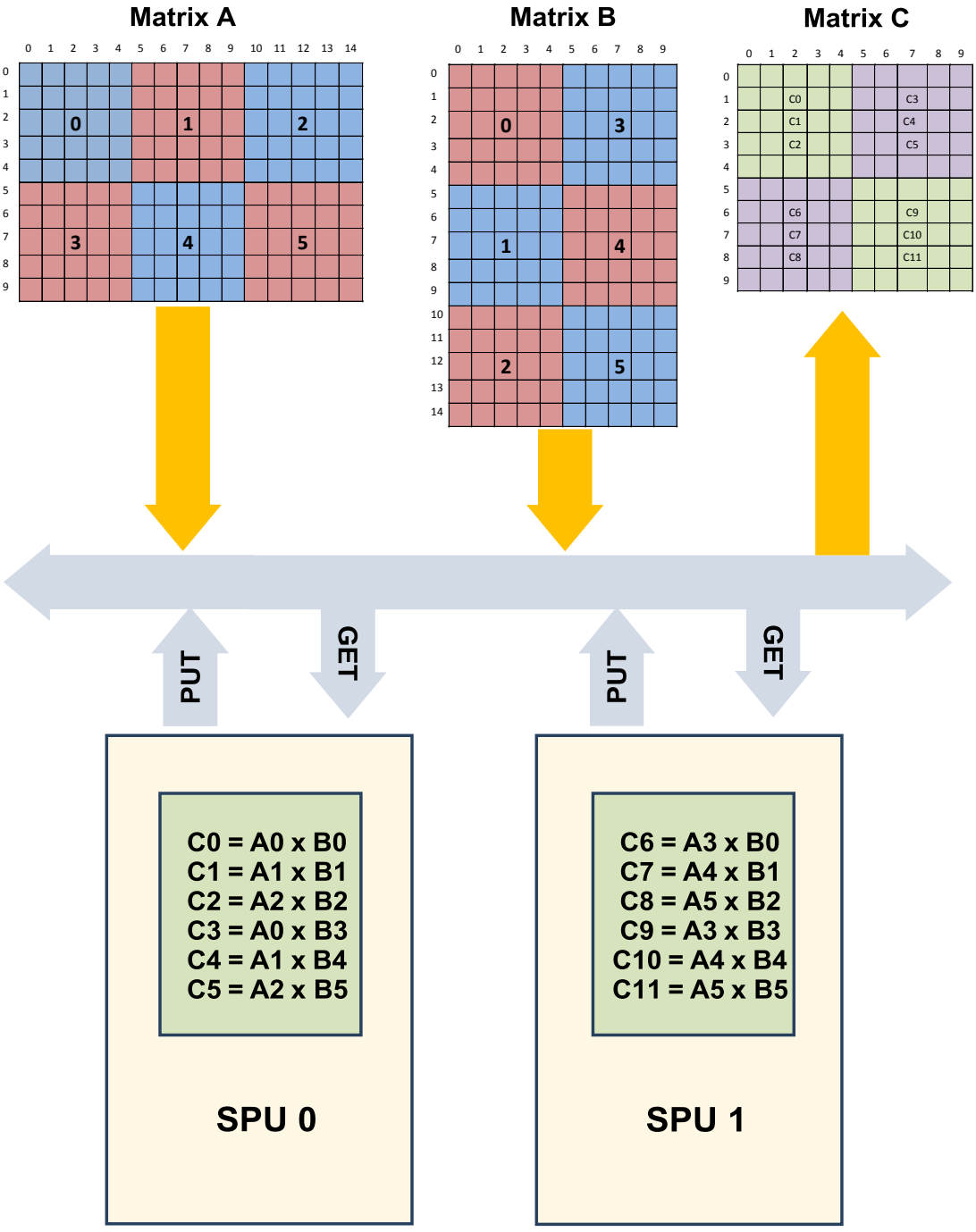

Figure 18: Data transfer for matrix multiplication

4.6.6 Implementation with One & Multiple SPEs

In this section, the overall execution of the model program will be presented considering all the design aspects and limitations that were described in the previous paragraphs. The transfer of all the required data to the SPEs was resolved and a complete multiplication can now take place. At first only one SPE it was used and then the implementation was extended to multiple SPEs.

Until this point the model program was executing a matrix multiplication on the PPE with the data partitioning method that was described in paragraph *4.6.1*. Next the appropriate scheduling control was added in this program and the multiplication function was moved to the SPE code. We developed the code for DMA transfers and the program was ready to be executed. The model application on the PPE side creates two matrices, starts an SPE thread and waits for it to finish. On the SPE side, it transfers the appropriate data for a pair of submatrices, calculates the multiplication and sends back the result; this procedure is repeated until all the submatrices multiplications are completed. At this stage there is no parallelism since only one SPE it was used, which executes the partial multiplications sequentially.

The design was extended on two and finally on six SPEs (and eight SPEs on simulation only), the procedure remains the same as in the case of one SPE. The advantage now is that the total number of iterations that are required for one multiplication is distributed on multiple SPEs. To mange executing one multiplication on multiple SPEs, each SPE had to be informed about how many iterations should execute, which data must fetch and where should store the result submatrix of each iteration. This information is passed once at each SPE through a structure called *control block*. The control block contains information such as, matrices addresses, iterations number, result address and information related to the pairs of submatrices that should be multiplied.

So far the data partitioning problem was resolved, a model application was implemented for our design; the model application and the total application were ported to the PPE and with the use of PPE control and DMA transfers the model application was executed on one and multiple SPUS. At this stage the correctness of the implementation on the SPEs was also verified with the subtraction method that was described in paragraph *4.6.2*. The next stage of the development was the optimizations of the SPE code. Next section describes a number of optimizations made on the SPE code in order to increase its efficiency.

4.6.7 Code Optimizations

Writing code for the Cell is, in many ways, different than programming most of the common modern architectures (in particular the x86 processors family). The main differences come from the fact that, on the Cell architecture, the user has full control over the processor behavior and all the hardware details are exposed to the programmer. This puts a serious burden on the programmer who has to take care of many aspects while writing code. A number of general, but important, programming rules must be followed, where possible. What the programmer gets for this price is extremely predictable performance and the possibility to get really close to the peak speed whenever these rules can be applied. These programming rules are in effect a number of code optimizations that the programmer must apply [33], [28], [30]. There are many possible code optimizations that are mainly related with the SPE code and less with the PPE code. The most of them try to exploit the advantages of Cell's architecture, such as SIMD, wide registers, LS and some of them try to hide the disadvantages of the architecture like data transferring and branch prediction.

From all the optimizations some of them were selected and applied to the design, these optimizations are the following:

- Function Inline
- Code SIMD Vectorization
- Loop Unrolling

The optimizations follow the same philosophy as the rest of the development process. First the optimizations were applied on the model application and when the performance and the correctness of the code were verified, it was joined with the CSU Bayesian application.

Function Inline

The first step of the optimizations was the use of the compiler in such way to produce optimized code. A specific flag was used, the *-Winline*, in the makefiles to force the compiler produce code with function inlining. Function-inlining eliminates the two branches associated with function-call linkage. These include the branch and set link for function-call entry, and the branch indirect for function-call return. SPEs can only do static branch prediction, since these prediction schemes are rather inefficient on programs that have a complex execution flow, reducing the number of branches in the code usually provides performance improvements. The compiler by

default has the optimization level set to three, *-O3*, which does some function inlining so the extra inlining that was applied didn't had significant increase in performance.

SIMD Vectorization

The next and most important optimization was the SIMD vectorization of the SPE code. As mentioned in Chapter 2, each SPE is an SIMD processor which means a single instruction can be applied to multiple data elements in parallel. SIMD processing exploits data-level parallelism, this is the second level of parallelism in the implementation. Data-level parallelism means that the operations required to transform a set of vector elements can be performed on all elements of the vector at the same time. Vectorization is also supported and in the PPE through the vector unit [32], [16], but only the computation-intensive part of the SPE code was vectorized.

In both the PPE and SPEs, vector registers hold multiple data elements as a single vector. The data paths and registers supporting SIMD operations are 128 bits wide, corresponding to four full 32-bit words or eight half-words, or 2 double-words. Both the vector unit of PPE and SPE instruction set have extensions that support C-language intrinsics [28], [30]. Intrinsics are C-language commands, in the form of function-calls that are convenient substitutes for one or more inline assembly-language instructions. In our design all the data were double, so the vector registers could only hold two 64-bit values and the data parallelism offered by the SIMD vectorization is reduced to only two simultaneous operations. This means that once the data were promoted from double to vector type it was able to execute two loads, two multiplications, two additions simultaneously and so the iterations were reduced by a factor of two. Next figure, **Figure 19,** explains the SIMD vectorization procedure followed in the implementation.

As shown in **Figure 19,** instead of loading one element into one 128-bit register we use the data type *vector* and load two matrix elements into one register. After this the vector command is being executed for multiplication and addition, *spu_madd* which executes in parallel the operations C1 = A1 x B1 + C1 and C2 = A2 x B2 + C2. Next the two elements C1 and C2 are added and stored to the appropriate place of the result matrix. This is how the multiplication code on the SPEs was vectorized; the code vectorization on its own was not enough, more performance was needed so the loop unrolling technique was applied.

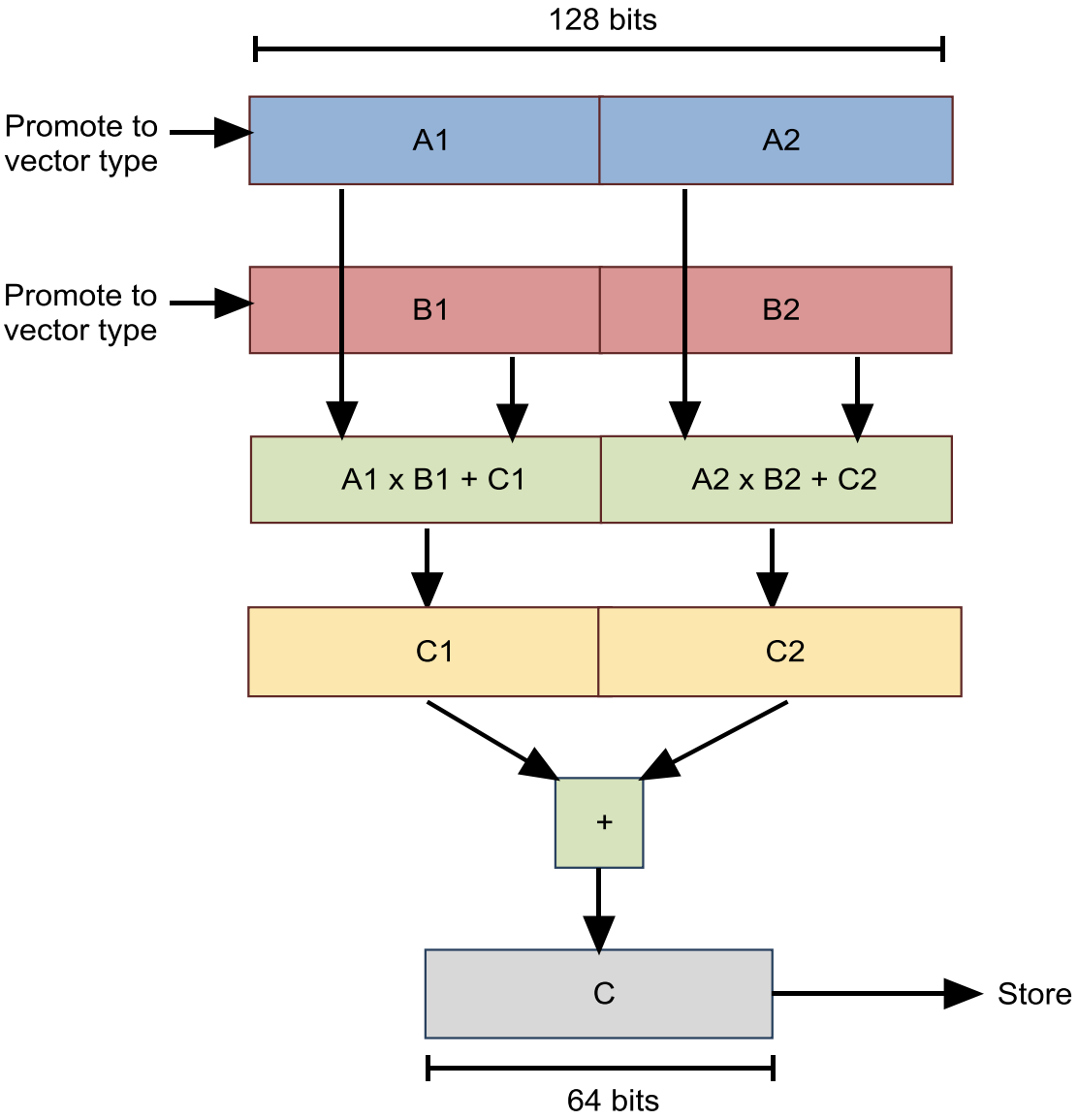

Figure 19: SIMD vectorization procedure on SPEs code

Loop Unrolling

The final optimization that was applied to the design was the loop unrolling technique. This technique is often used to increase the size of *basic blocks* (sequences of consecutive instructions without branches), which increases scheduling opportunities. Loop-unrolling eliminates branches by decreasing the number of loop iterations. Loop unrolling can be manual, compiler directed, or

compiler automated. Typically, branches associated with looping are inexpensive because they are highly predictable. However, if a loop can be fully unrolled, then all branches can be eliminated, including the final non-predicted branch. Due to the high number of registers on the SPEs and to the simplicity of SPEs architecture, no register renaming, no speculative execution, no dynamic branch prediction etc., explicit unrolling provides considerable improvements in performance. In our implementation the code was manually unrolled many times in order to achieve a significant improvement in performance. The SPE code was gradually unrolled until the improvement was not important; finally the code was unrolled twenty-five times. Next chapter provides detailed results for 5x and 25x unrolling.

4.6.8 Join with CSU Bayesian

The final step of the development was the join of the offloaded functions with the rest of the CSU Bayesian application. Until now all the development may be considered as modeling since we were working on a simple model program. At this point all the work that was done for the model program must be joined with the main application.

Up to this stage the model application was executing one or more matrix multiplications on multiple SPEs with optimized SPE code, the correctness and the performance of the model were also verified. All the calls of matrix multiplication or transpose matrix multiplication were located and replaced in the CSU Bayesian application. Instead of executing the default code for the multiplication functions, the program flow was redirected to our code. As shown in **Figure 20** the program after the matrix multiplication function-call executes a series of scheduling operations and initiates the execution of the SPE code. When the SPE code has been executed the program return to the PPE side, destroys the SPEs context, creates the result matrix and returns to the normal execution flow until the next matrix multiplication function-call.

The above procedure works fine with the CSU Bayesian Train because it is executing two different functions at the SPEs, the number of the function-calls is small and the execution time at the SPEs is much greater than the scheduling overhead. In the case of CSU Bayesian Project, as it was referred in section *4.6.4* this procedure was proved to be wrong because only one function was repeatedly running on SPEs thousands times with small execution time. In order to improve the performance of CSU Bayesian Project a different approach was followed. Since the SPEs were running the same program it wasn't necessary to create new context and load the program each time. Instead of this, the context was created and the program was

loaded only once, so the only operation of scheduling that was repeated each time was the initiation of the SPE program, this alternate approach is shown in **Figure 21**.

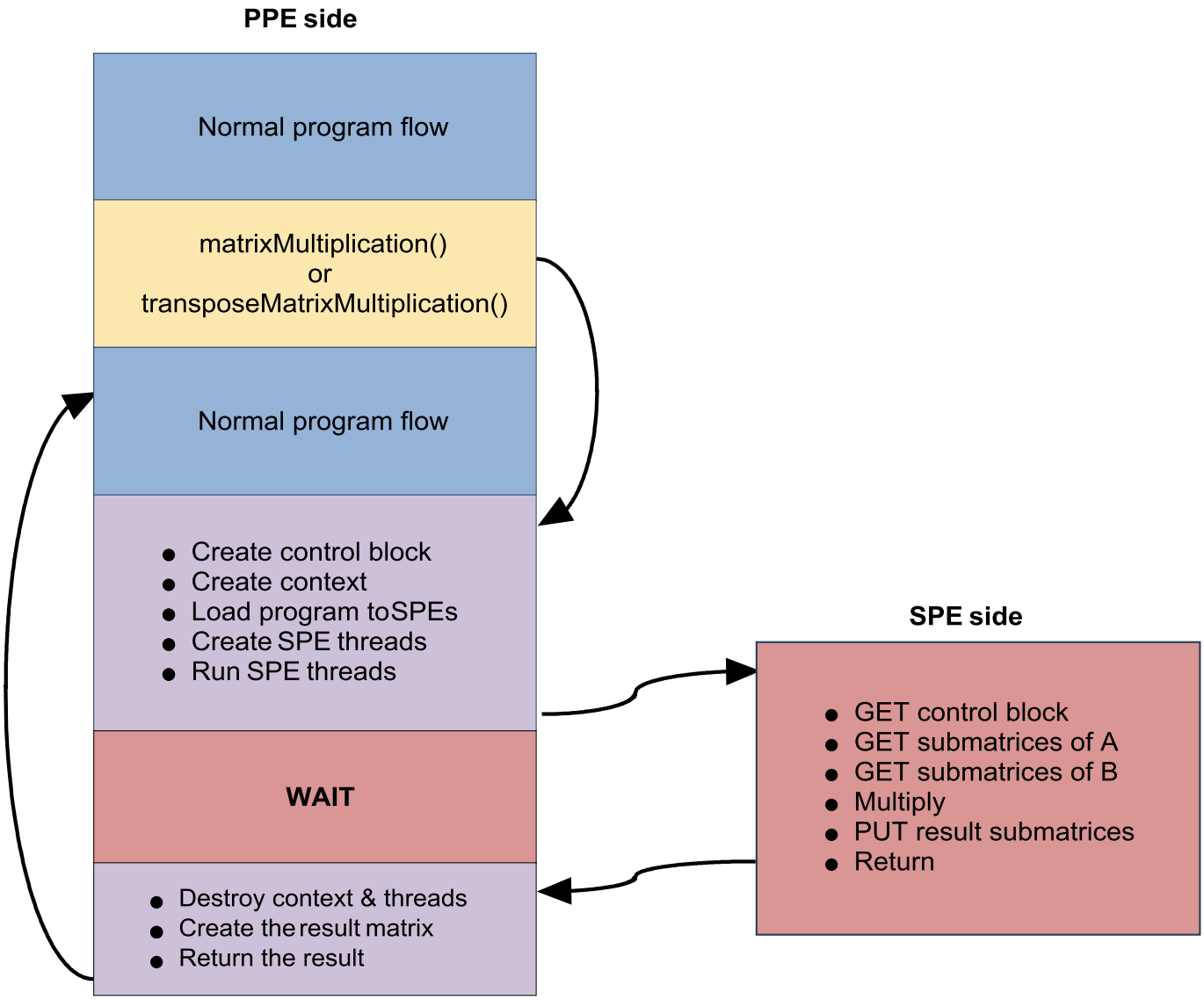

Figure 20: Overall scheduling process for CSU Bayesian Train

As shown in the next figure there are two function-calls in a loop, the creation of the context and the program loading was done outside the loop to avoid their unnecessary repeat. When there is a function-call the PPE side of the code creates the appropriate control block, which is different for every function call and initiates the execution at the SPE side.

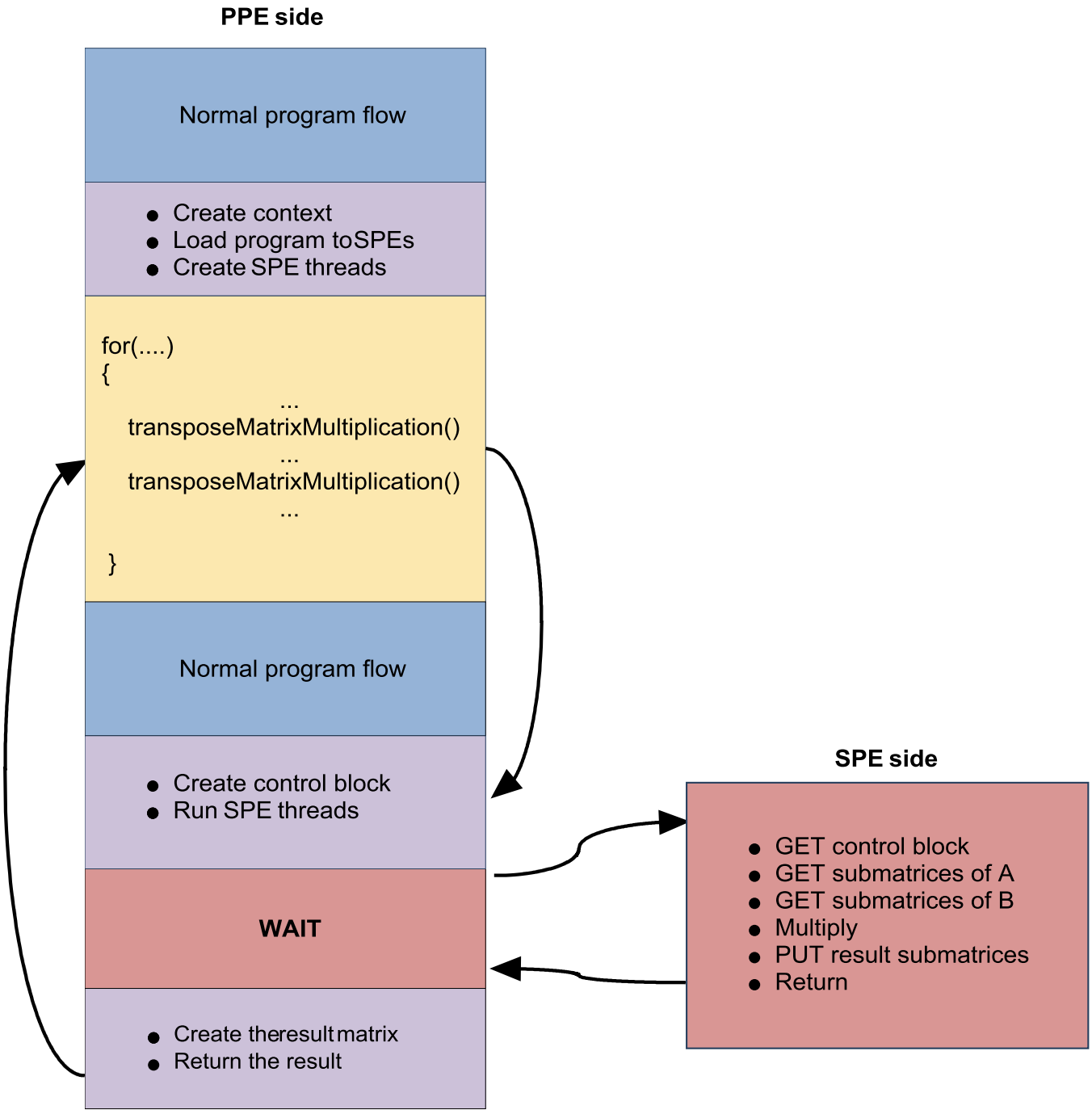

Figure 21: Overall scheduling process for CSU Bayesian Project

Unfortunately not even with this alternative scheduling procedure managed to reduce the impact of scheduling overhead, due to the large number of function-calls and the very small execution time spend at the SPEs. The last optimization that was implemented to increase the performance was the execution of two function-calls together at the SPEs. The number of two function-calls was not selected arbitrarily it was selected because in each iteration of the loop we have two function-calls. A larger number would be preferred but because the iterations of the loop were varying it wasn't able to unroll the loop and offload more function-calls together. With this last optimization the overhead from the SPE initiation procedure was reduced to the half as the initiation was executing once every two multiplications.

4.7 Software Tools Problems

During the development process, apart from the design problems that were described in the previous paragraph, many other problems came up, mostly related with the software tools. In this section a list of these problems is presented, as well as how each problem solved or avoided.

The last version of the *IBM SDK for Multicore Acceleration Version 3.0* (Development Tools + Full-System Simulator), was unable to be installed on one of the available servers. The version of the kernel that runs on the servers was not compatible with the SDK, a newer version was required. It was not currently able to update the servers, so the previous version of SDK was used, version 2.1.

The source code compilation and execution with the 2.1 version SDK was working fine until the *cycle-mode* was enabled. The cycle-mode gives accurate performance results of the application, but as it was discovered, a bug in the 2.1 version was preventing the execution of code with DMA transfers in cycle-mode. This was a major problem because it was unable to measure the performance. The solution to this bug from IBM was the version 3.0 of the SDK, so the version 3.0 was installed on a P4 host machine (not on any server) with OS *Ubuntu 7.10*.

The *Full-System Simulator* application that is included in the SDK is a very demanding application and especially the cycle-mode was extremely slow on the host machine. This was delaying the development process and it was necessary to avoid it, the solution was to execute the applications directly on hardware and measure the performance in different way. The only available hardware was a PS3 and to be able executing applications on the PS3 the installation of an OS was required.

The Yellow Dog Linux 6.0 (YDL) was installed on the PS3, the installation procedure was done by following the detailed guide for YDL installation [34]. The only conflict during the installation was the monitor configuration, because PS3 normally is connected on HDMI monitor, the installation was not working until the proper settings for the monitor were chosen.

Normally the development with the use of IBM's SDK includes a graphical user interface (GUI) environment through Eclipse and a graphical performance analyzer called *Visual Performance Analyzer*. Due to the low memory of the PS3 the GUI was avoided and the development was done with a simple text editor and command line compiler.

The use of the PS3 for development demands a careful memory management from the programmer, during the development a memory leak in a program was causing continues memory swaps. The hard disk of the PS3 is very slow and the swap procedure weighs down the entire OS as well as the running application. Avoiding the development of applications with high memory usage on the PS3 is recommended. Another limitation on the PS3 development is the availability of only six SPEs out of eight, as it was mentioned in paragraph *2.2*. Besides these limitations development on the PS3 is much faster than the use of the simulator, the simulator is better to be used only for specialized measures of performance that cannot be done on hardware.

Most of the solutions on the above problems came up from the Cell development community; the main source for information was the IBM's *developerWorks* web site and forum [35], [36].

CHAPTER 5

Evaluation & Verification

This chapter presents the performance of the design and compares it with the performance of other processors for the specific algorithm. The measuring procedure is also described here as well as the precision and the verification of the implementation. The total application was executed and verified with a subset of the FERET database consisted of 120 images.

5.1 Measuring Performance

Measuring the performance of an application is a very important step and provides the programmer with critical information about its design. In our design the performance was measured gradually during the development process and at the end to determine the final performance of the implementation. For the Cell processor there are currently three ways to measure the performance of an application running on Cell. The first two methods are using two software tools that are available in the SDK to assist in measuring the performance, the *spu-timing analyzer* and the *IBM Full System Simulator for Cell B.E* [37], [38].

The spu-timing analyzer performs a static timing analysis of a program by annotating its assembly instructions with the instruction-pipeline state. This analysis is useful for coarsely spotting dual-issue rates (odd and even pipeline use) and assessing what program sections may be experiencing instruction-dependency and data-dependency stalls. However, static analysis outputs typically do not provide numerical performance information about program execution. Thus, it cannot report anything definitive about cycle counts, branches taken or not taken, branches hinted or not hinted, DMA transfers, and so forth.

The IBM Full System Simulator for the Cell B.E. performs a dynamic analysis of program execution. Any part of a program, from a single line to the entire program, can be studied. Performance numbers are provided for:

- Instruction histograms (for example, branch, hint, and prefetch)
- Cycles per instruction (CPI)
- Single-issue and dual-issue rates

- Stall statistics
- Register use

The output of the IBM Full System Simulator for the Cell Broadband Engine can be a text listing or a graphic plot. The disadvantage of this method is speed, because the method depends completely on the simulator is extremely slow. This method of performance measuring was partially used in the implementation to provide us detailed information for the SPE code, in order to make the appropriate optimizations.

The last method for measuring and the one that was followed in the design is the *dynamic profiling using the hardware counters*. The processor includes two software-visible 64-bit *time-base registers* in the PPE one for configuration and one for counting and eleven software-visible 32-bit *decrementers* (down-counters), three in the PPE and one in each of the eight SPEs [30]. The time-base registers and the decrementers are not clocked at the 3.2 GHz as the core clock, they have their one frequency called *time-base frequency*. This frequency is different on the PS3 than on the Cell Blades [39], the PS3 time-base frequency is *79.8 MHz* and this value was used for our measurements.

During the measuring procedure the one 64-bit time-base register in the PPE was used to measure the total execution time and execution time of code segments at the PPE. The SPEs performance was measured with the use of the decrementers of each SPE. Both types of time-base registers were providing us with a number of clockticks which was converted to execution time by dividing with the time-base frequency. In the case of the SPEs, when multiple SPEs were used the greater time was considered as the SPEs execution time.

In order to have a fair comparison the total execution time for P4 was measured in a similar way. The *time.h* library was used for the P4 to measure the real execution time through the OS. The main purpose was to compare the processors and not the systems, so for both measurements the amount of time for loading data to the main memory and for storing data to the hard disk was taken out. Furthermore all the *printf* system-calls were removed from the programs to avoid as much as possible the OS since the two processors are running different OS. Due to the OS measurements of the same code had a small variation, so five measurements were taken for each case and the average is being presented as the final result.

The measuring procedure that was described above was applied for all the measurements and the results that are following in the next paragraph.

5.2 Performance

This section presents all the performance measurements that were done to evaluate the implementation. First the performance of the model application was measured and compared with equivalent application running on *P4 at 2.66 GHz and 1 GB RAM,* this machine was our reference machine. Then the performance of the total code running on SPEs (multiplication function) was measured and compared with the performance of the same function on the reference machine (P4). Finally the total execution time of CSU Bayesian Project and CSU Bayesian Train was evaluated and compared with the P4.

5.2.1 Performance of Model Application

Firstly the performance of the model application was measured for a single multiplication of each type. The model application was used to perform only one multiplication of each type with a variety of code optimizations. The results are only for the execution time of the multiplication and not of the whole application. The summary of the results is shown in the next tables and figures.

The following tables, **Table 7** and **Table 8** shows the measured clockticks and the execution time for the first case of transpose matrix multiplication with dimensions (19500x59)x(19500x1).

Clockticks 79.8 MHz	Original	SIMD	SIMD + 5x Unroll	SIMD + 25x Unroll
1-SPU	1046791	901476	548363	471731
2-SPUs	527084	455771	287213	242302
4-SPUs	276705	240020	147039	127780
6-SPUs	199681	158850	95475	84613

Table 7: Clockticks for transpose matrix multiplication (19500x59)x(19500x1)

Execution Time (sec)	Original	SIMD	SIMD + 5x Unroll	SIMD + 25x Unroll
1-SPU	0,01311	0,01129	0,00687	0,00591
2-SPUs	0,00660	0,00571	0,00359	0,00303
4-SPUs	0,00346	0,00300	0,00184	0,00160
6-SPUs	0,00250	0,00199	0,00119	0,00106

Table 8: Execution time for transpose matrix multiplication (19500x59)x(19500x1)

As shown in **Table 7** and **Table 8** the performance of the code had a significant improvement with the use of multiple SPEs and the optimizations that were mentioned in paragraph *4.6.7*. **Figure 22** shows the gradually improvement of the performance for the specific multiplication.

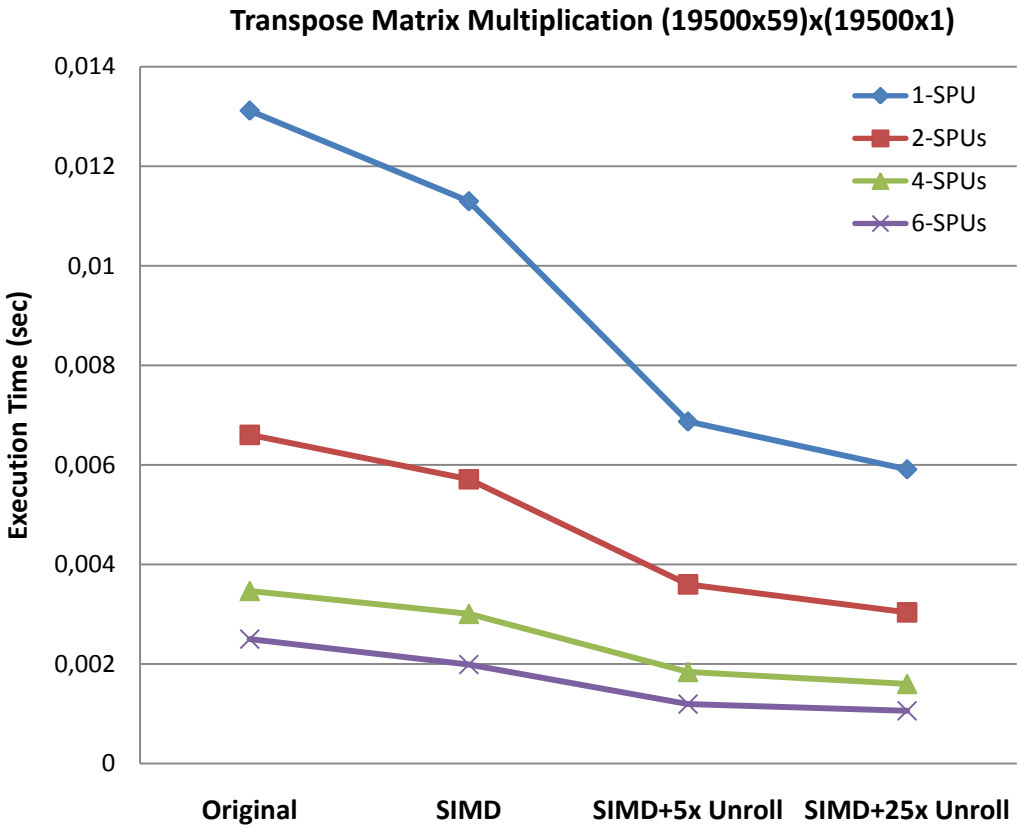

Figure 22: Performance impact of various optimizations (1)

In **Figure 23** the performance results that were taken for the SPEs are being compared with execution time of the same code on the reference machine. Only the results for fully optimized code are being used in the comparison. The CSU implementation of the BIC algorithm is using an optimization level *–O3* during the compilation, with an important improvement in performance, so the performance of the model application was measured on P4 and with an *–O3* option. As it is shown the performance of the P4 *–O3* machine, compared with one SPU, is slightly better. On the other hand comparing the execution time on 6-SPUs with the P4 *–O3* execution time, a speed up of *4.52x* is achieved.

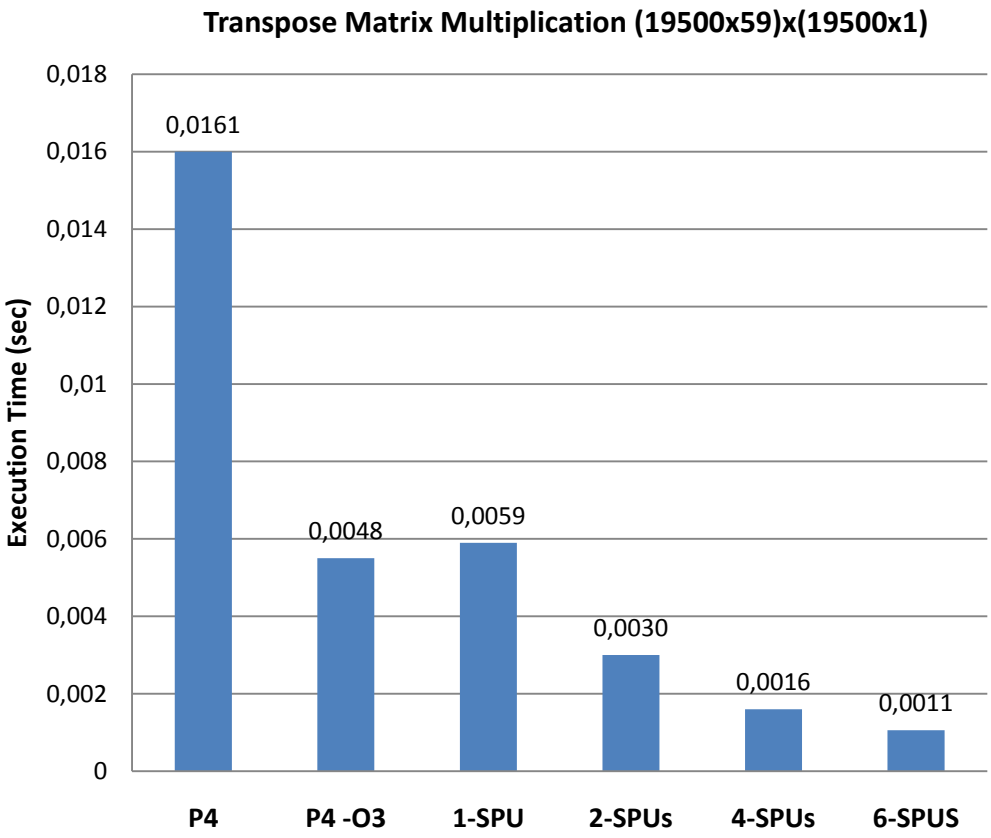

Figure 23: Performance comparison for model application (1)

The next tables, **Table 9** and **Table 10**, are showing the performance results, clockticks and execution time, for the transpose matrix multiplication (19500x100)x(19500x100). As before a figure shows the performance impact of the various optimizations, **Figure 24**, and another one shows the comparison with the reference machine, **Figure 25**.

Clockticks 79.8 MHz	Original	SIMD	SIMD + 5x Unroll	SIMD + 25x Unroll
1-SPU	155876357	130946171	69766634	57622354
2-SPUs	77998093	65276335	34905702	28844555
4-SPUs	39604747	33156633	17738707	14670401
6-SPUs	26031831	21798942	11669627	9701396

Table 9: Clockticks for transpose matrix multiplication (19500x100)x(19500x100)

Execution Time (sec)	Original	SIMD	SIMD + 5x Unroll	SIMD + 25x Unroll
1-SPU	1,95334	1,64093	0,87427	0,72208
2-SPUs	0,97742	0,81800	0,43741	0,36146
4-SPUs	0,49630	0,41550	0,22229	0,18384
6-SPUs	0,32621	0,27317	0,14624	0,12157

Table 10: Execution time for transpose matrix multiplication (19500x100)x(19500x100)

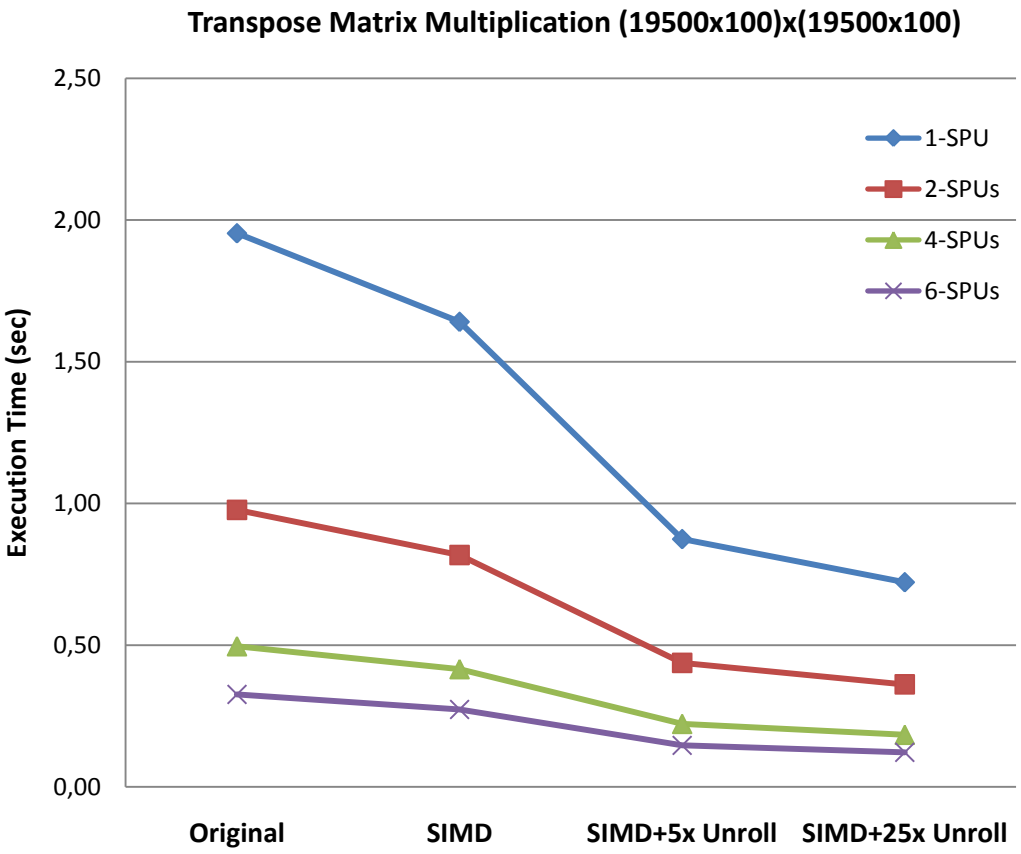

Figure 24: Performance impact of various optimizations (2)

Next figure, **Figure 25** shows the comparison of the SPEs with the reference machine. In contrast with the previous comparison, in this type of multiplication, the 1-SPU implementation achieves better performance than the optimized version on P4. Finally the 6-SPUs implementation has speed-up *6.9x*, compared with the P4, for the specific type of multiplication.

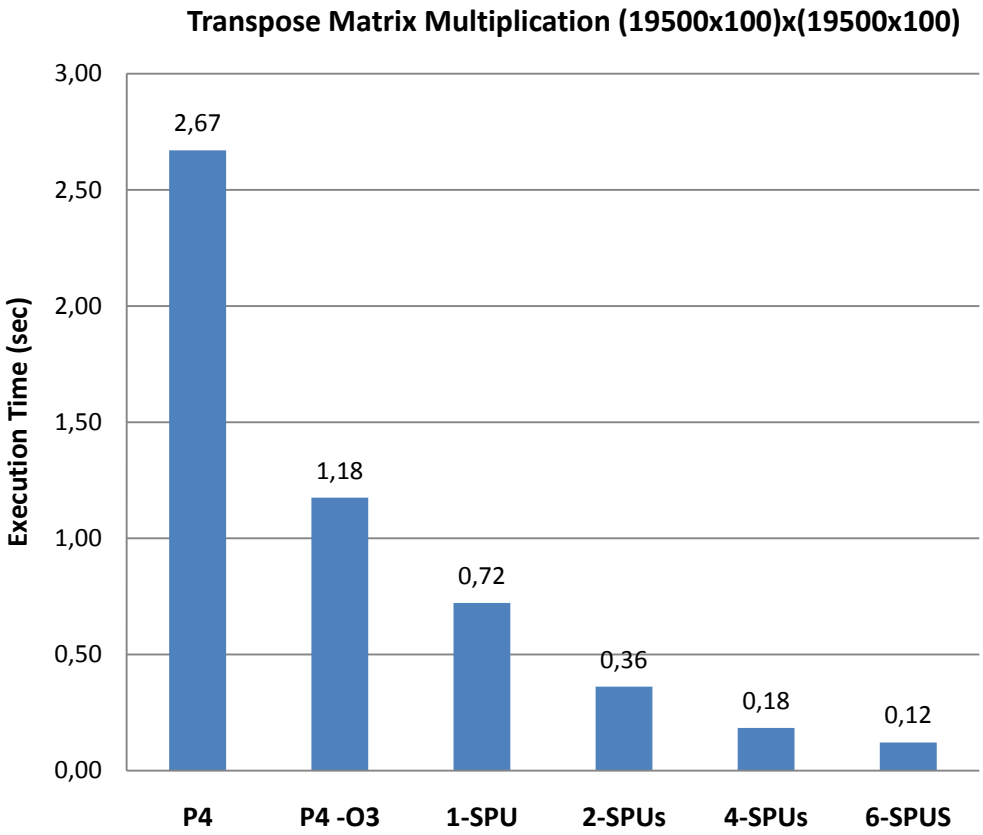

Figure 25: Performance comparison for model application (2)

The last results concern the third type of multiplication, the normal matrix multiplication with dimensions (19500x100)x(100x100). The results are presented with same order as before, **Table 11**, **Table 12** and **Figure 26** are showing the performance measurements and **Figure 27** shows the comparison results with reference machine.

The normal matrix multiplication proved to be much slower than transpose matrix multiplication and this is due to the column major form that was used to store the matrices in the main memory as was mentioned in paragraph *4.4*. The speed-up that is being achieved in this type of multiplication is *13x* and is greater from the previous cases.

Clockticks 79.8 MHz	Original	SIMD	SIMD + 5x Unroll	SIMD + 25x Unroll
1-SPU	198780074	178159632	100880352	86438608
2-SPUs	99394563	89100817	50457176	43210090
4-SPUs	50458809	45233567	25615118	21939141
6-SPUs	33127616	29703043	16821343	14416868

Table 11: Clockticks for matrix multiplication (19500x100)x(100x100)

Execution Time (sec)	Original	SIMD	SIMD + 5x Unroll	SIMD + 25x Unroll
1-SPU	2,49098	2,23258	1,26416	1,08319
2-SPUs	1,24555	1,11655	0,63230	0,54148
4-SPUs	0,63232	0,56684	0,32099	0,27493
6-SPUs	0,41513	0,37222	0,21079	0,18066

Table 12: Execution time for matrix multiplication (19500x100)x(100x100)

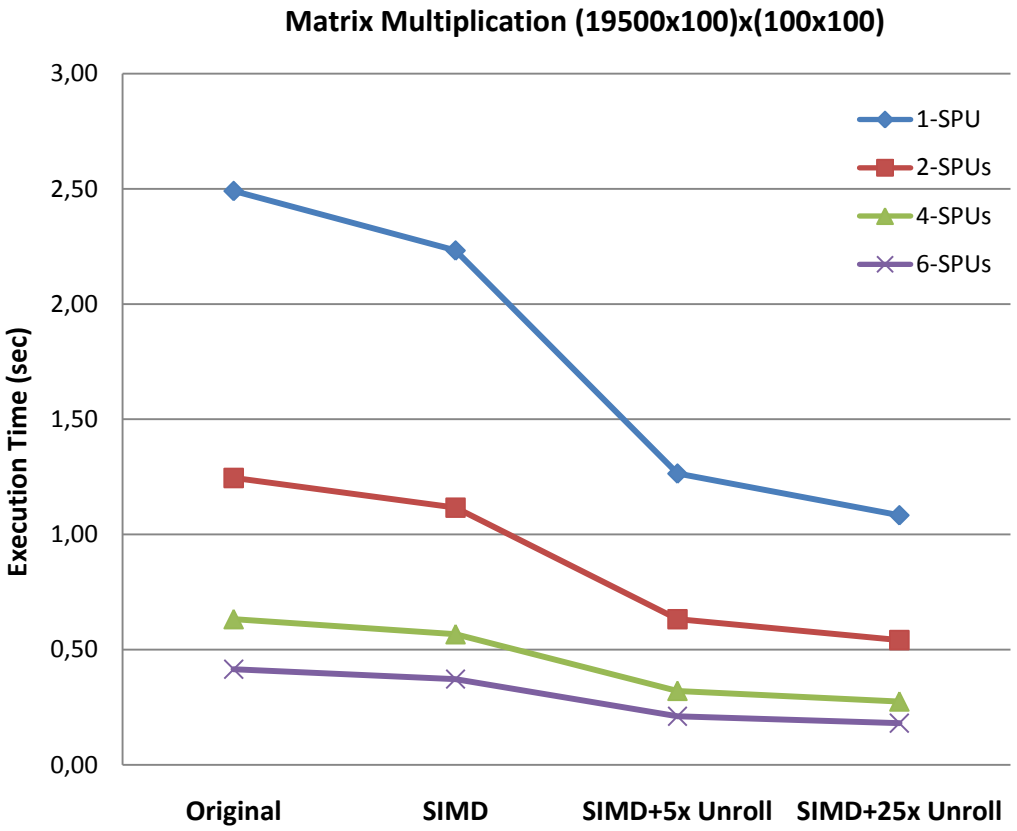

Figure 26: Performance impact of various optimizations (3)

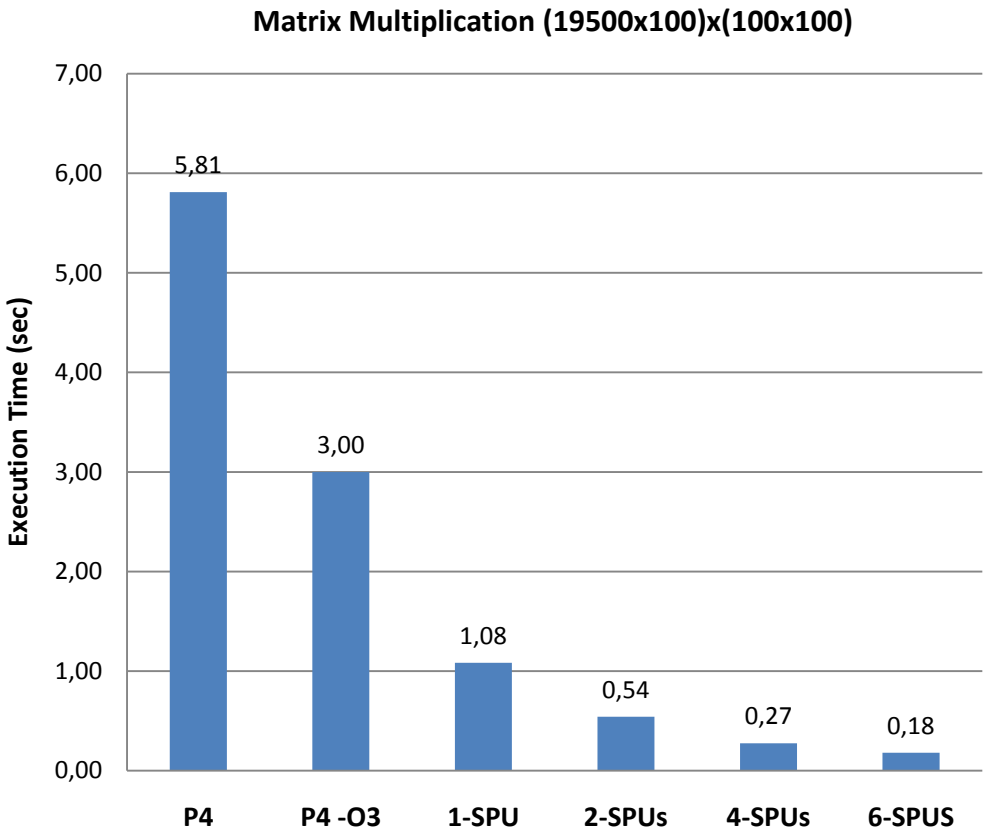

Figure 27: Performance comparison for model application (3)

5.2.2 Performance of SPEs

After the implementation was joined with the rest CSU Bayesian application, the first thing that was measured was the total execution time of the code running on the SPEs. The execution time of the SPEs was measured separately for the Bayesian Train and the Bayesian Project.

The results for the Bayesian Train application are shown in **Table 13** and **Table 14**. **Figure 28** shows as before the impact of various optimizations for the code running on SPEs in Bayesian Train. Finally **Figure 29** shows the comparison between the execution time for the multiplications on the reference machine and the total time consumed by the SPEs on Cell, for the Bayesian Train application. The speed-up that comes up from this comparison is *10.16x*, to make it clear this speed-up concerns only the multiplication time and has nothing to do with the real speed-up. This is the maximum speed-up that can be achieved if overhead is null. Unfortunately the

overhead is not null so the real speed-up is much less than this one, the overhead is being discussed in the next section were the overall speed-up is presented.

Clockticks 79.8 MHz	Original	SIMD	SIMD + 5x Unroll	SIMD + 25x Unroll
1-SPU	1030599761	889726021	493054770	413494050
2-SPUs	515919751	445655372	247038005	208062817
4-SPUs	262608579	233691657	127327553	108099230
6-SPUs	173234840	150420564	85080956	74703660

Table 13: Clockticks at SPEs for CSU Bayesian Train

Execution Time (sec)	Original	SIMD	SIMD + 5x Unroll	SIMD + 25x Unroll
1-SPU	13,08	11,29	6,26	5,25
2-SPUs	6,55	5,66	3,14	2,64
4-SPUs	3,33	2,97	1,62	1,37
6-SPUs	2,20	1,91	1,08	0,95

Table 14: Execution time at SPEs for CSU Bayesian Train

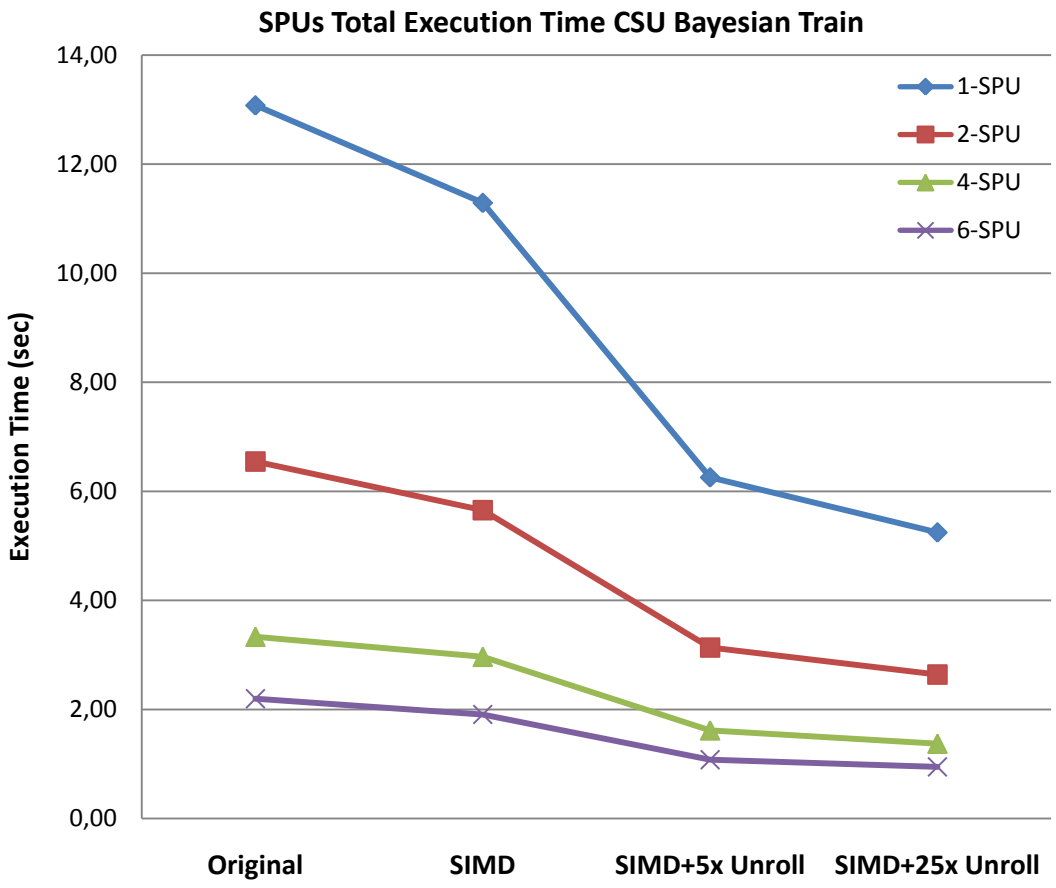

Figure 28: SPEs total execution time for Bayesian Train application

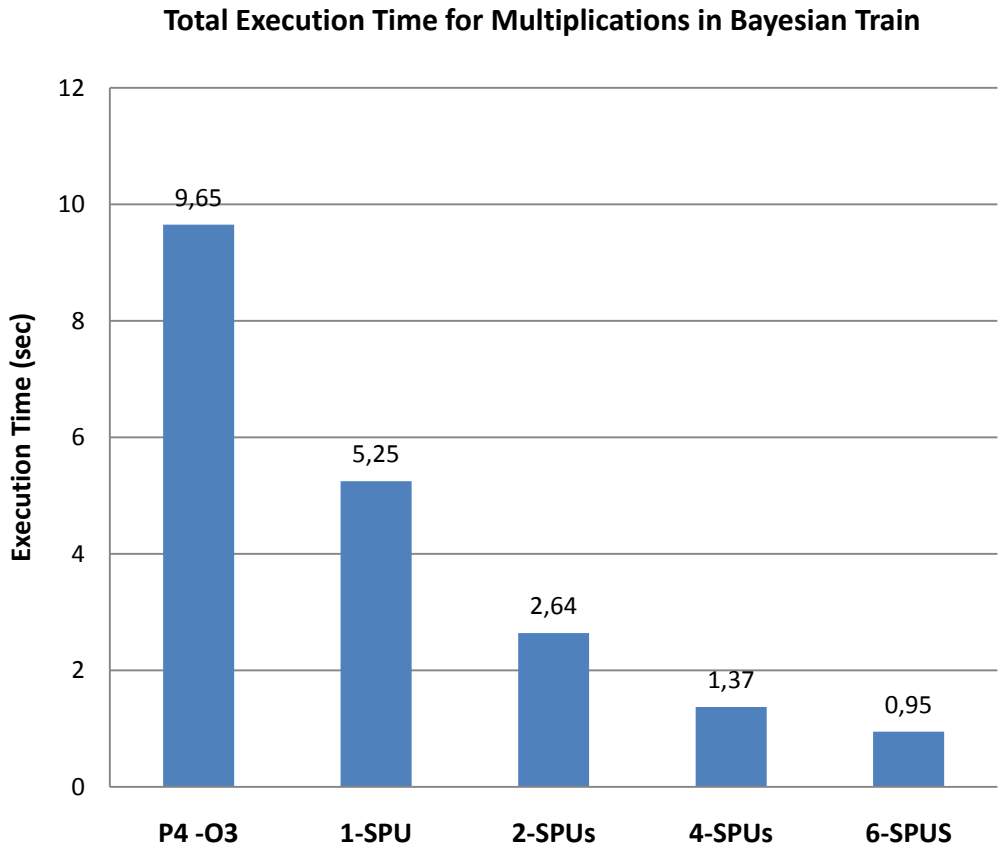

Figure 29: SPEs performance comparison with P4 for Bayesian Train

Next tables and figures present the results and the comparisons for Bayesian Project in the same way they were shown for Bayesian Train.

Clockticks 79.8 MHz	Original	SIMD	SIMD + 5x Unroll	SIMD + 25x Unroll
1-SPU	1,52E+10	1,31E+10	7,90E+09	6,87E+09
2-SPUs	7,67E+09	6,60E+09	4,07E+09	3,51E+09
4-SPUs	4,00E+09	3,47E+09	2,15E+09	1,87E+09
6-SPUs	2,64E+09	1,23E+09	1,42E+09	2,28E+09

Table 15: Clockticks at SPEs for CSU Bayesian Project

Execution Time (sec)	Original	SIMD	SIMD + 5x Unroll	SIMD + 25x Unroll
1-SPU	193,05	166,12	100,27	87,17
2-SPUs	97,27	83,81	51,68	44,59
4-SPUs	50,81	44,01	27,34	23,79
6-SPUs	33,52	28,99	17,97	15,64

Table 16: Execution time at SPEs for CSU Bayesian Project

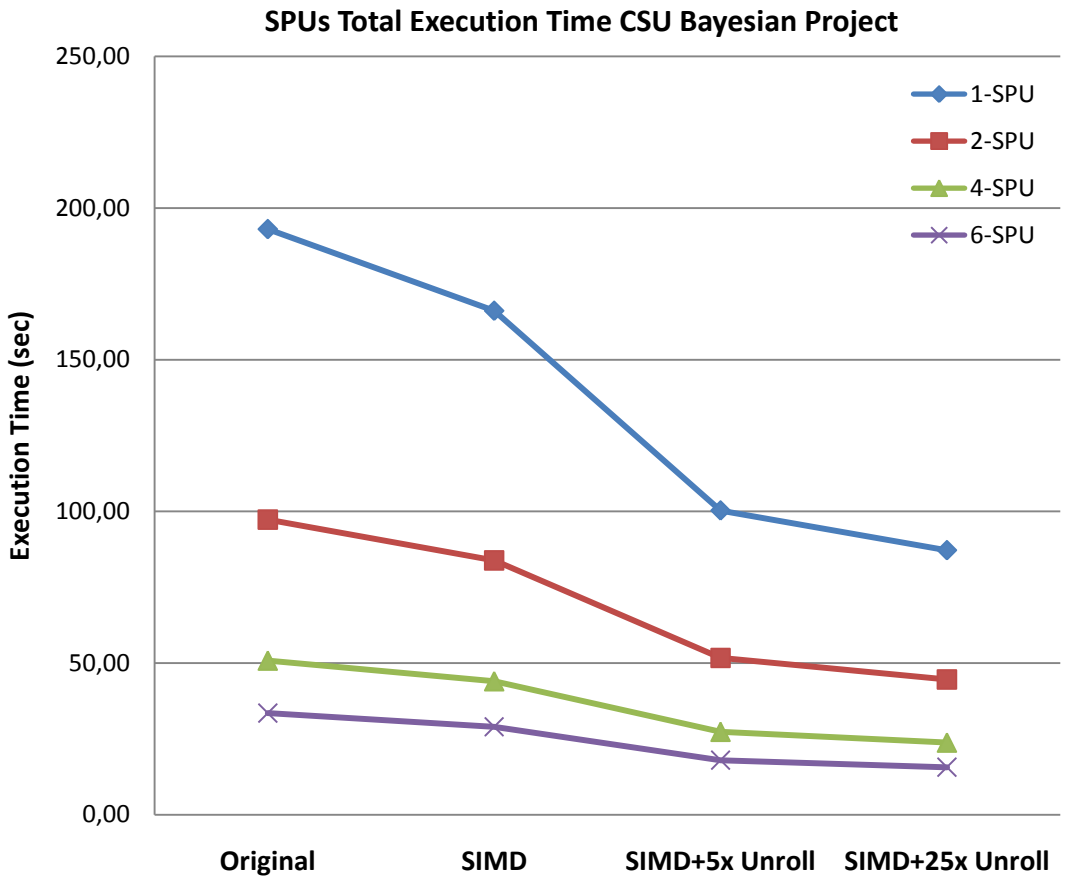

Figure 30: SPEs *total execution time for Bayesian Project application*

The above figure, **Figure 30**, shows the total execution time of the SPEs for the Bayesian Project application. In contrast with Bayesian Train, Bayesian Project becomes faster from the P4 only when two or more SPEs are in use. This difference is due to the size of matrices, smaller matrices are being multiplied in Bayesian Project.

The same result is shown and in **Figure 31**, for two SPEs and up Cell is faster than P4 in this function-level comparison. The maximum speed-up for Bayesian Project as it derives from the results is *4.56x,* approximately two times less than the maximum speed-up that was achieved in the case of Bayesian Train. As mentioned before this not the real speed-up but the maximum speed-up for the multiplications and not for the total application

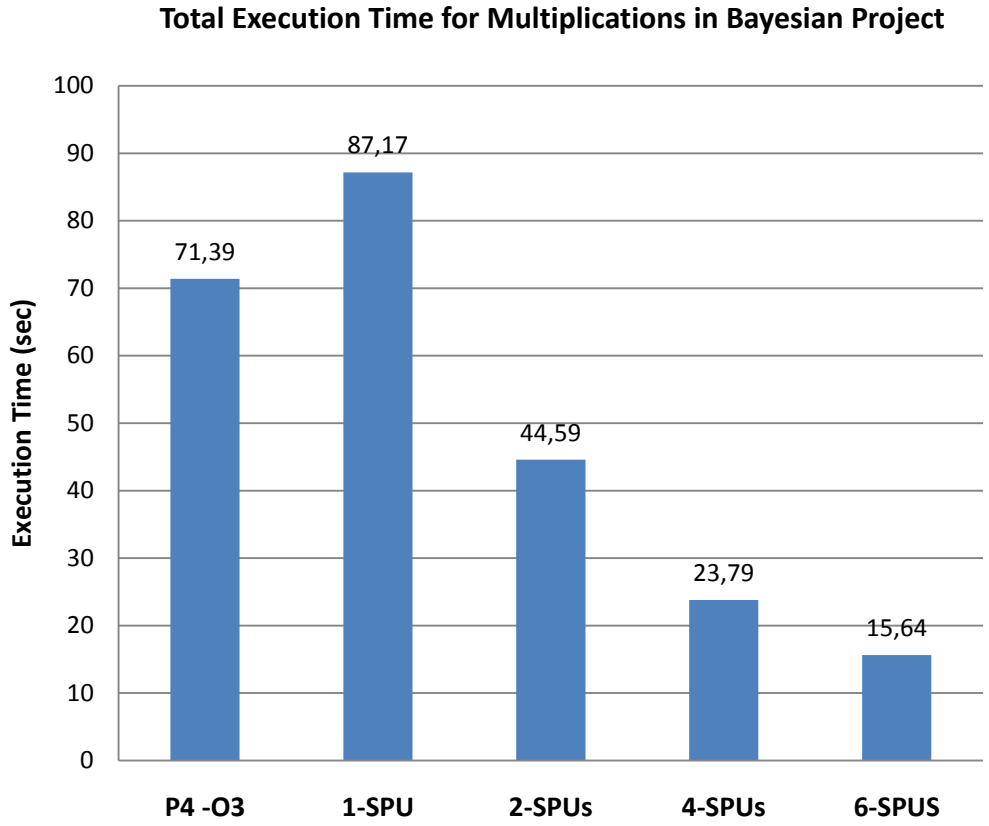

Figure 31: SPEs performance comparison with P4 for Bayesian Project

5.2.3 Total Performance

Finally the total performance for both applications is being measured, total execution time on the reference machine and total execution time on Cell. First the results for the Bayesian Train application are shown in the next tables and figures. **Table 17** and **Table 18** show the measured clockticks and the total execution time for the Bayesian Train in same form with all the previous results.

Clockticks 79.8 MHz	Original	SIMD	SIMD + 5x Unroll	SIMD + 25x Unroll
1-SPU	1,15E+09	1,00E+09	5,95E+08	5,16E+08
2-SPUs	6,22E+08	5,52E+08	3,51E+08	3,10E+08
4-SPUs	3,67E+08	3,26E+08	2,25E+08	2,06E+08
6-SPUs	2,79E+08	2,55E+08	1,95E+08	1,79E+08

Table 17: Total clockticks for CSU Bayesian Train

Execution Time (sec)	Original	SIMD	SIMD + 5x Unroll	SIMD + 25x Unroll
1-SPU	14,41	12,56	7,46	6,47
2-SPUs	7,80	6,92	4,40	3,88
4-SPUs	4,60	4,09	2,82	2,58
6-SPUs	3,50	3,19	2,44	2,24

Table 18: Total execution time for CSU Bayesian Train

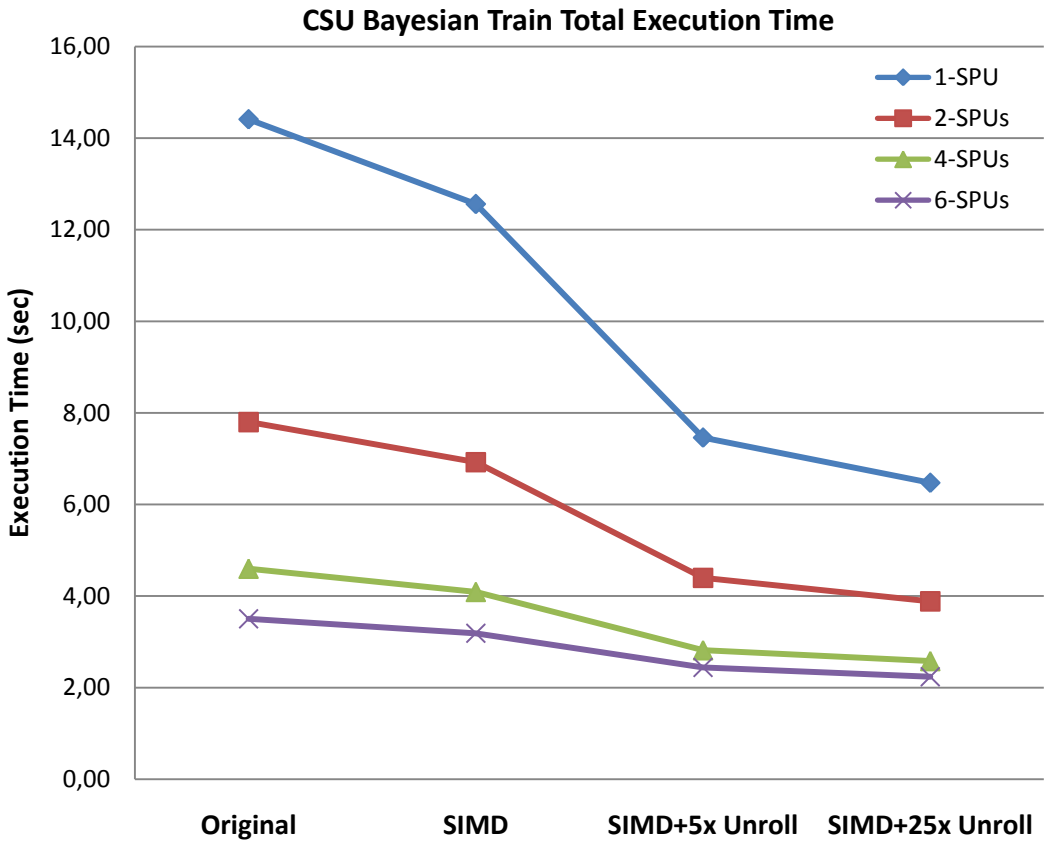

Figure 32: Total execution time of the Bayesian Train application

Figure 32 shows the gradual improvement in performance for the overall application of Bayesian Train. The impact of the optimizations is gradually decreased as the number of SPEs increases. This is caused because the percentage of the execution time that accepts the optimizations is reduced every time, on the other hand the added overhead is increasing.

Next figure, **Figure 33**, shows the performance of CSU Bayesian Train application compared with the reference machine and with the execution of the application on the PPE only.

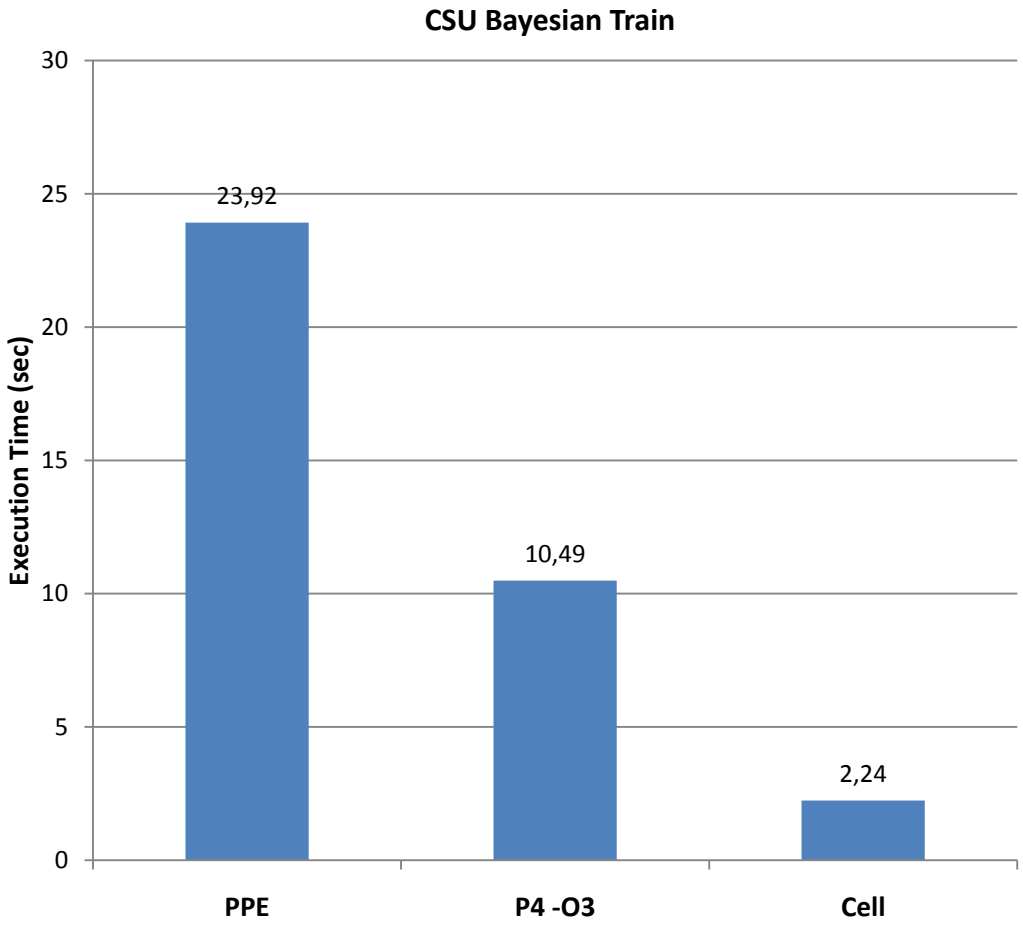

Figure 33: Performance comparisons for Bayesian Train

Obviously the speed-up that was finally achieved is much less than the function-level speed-up of the previous paragraph. The final speed-up is *4.68x* (in relation with P4), this decrement from the expected speed-up is caused by a number of factors. The main overhead in this case is the extremely slow execution of the rest code running on PPE, especially the part of this code which has many accesses to the main memory. There are two more factors which are increasing the overhead, the one is the scheduling process and the other is the additions and the merging to construct the final matrix from the submatrices. The scheduling overhead is affecting us but not as much as in the case of Bayesian Project; on the other hand addition and

merging are affecting us more in this case because of the larger matrices. Next table, **Table 19** shows a timing analysis of the Bayesian Train application.

	Execution Time (sec)	Percentage %
PPE code	0,89	40,09
SPE code	0,95	42,79
Addition	0,24	10,81
Merging	0,10	4,50
Scheduling	0,04	1,80

Table 19: Execution time analysis for CSU Bayesian Train

The last results are showing the overall performance for the CSU Bayesian Project application. As previously, **Table 20** and **Table 21** shows the clockticks and the execution time for the application. **Figure 34** shows the gradually improvement of the performance and the last figure, **Figure 35** shows the results of the comparison between Cell, PPE and P4.

Clockticks 79.8 MHz	Original	SIMD	SIMD + 5x Unroll	SIMD + 25x Unroll
1-SPU	1,69E+10	1,48E+10	9,68E+09	8,56E+09
2-SPUs	9,49E+09	8,36E+09	5,85E+09	5,28E+09
4-SPUs	5,81E+09	5,33E+09	4,01E+09	3,76E+09
6-SPUs	5,02E+09	4,72E+09	3,90E+09	3,76E+09

Table 20: Total clockticks for CSU Bayesian Project

Execution Time (sec)	Original	SIMD	SIMD + 5x Unroll	SIMD + 25x Unroll
1-SPU	212,01	185,95	121,27	107,31
2-SPUs	118,89	104,73	73,25	66,22
4-SPUs	72,77	66,85	50,30	47,08
6-SPUs	62,95	59,13	48,81	47,11

Table 21: Total execution time for CSU Bayesian Project

As shown in the next figure, **Figure 34**, the performance has a measurable improvement for two and four SPUs, but for six SPUs the execution time converges to the value that was achieved with 4-SPUs. This means that the percentage of the improvement between four and six SPUs is very small and is being overlapped by the scheduling overhead which is increasing according to the number of SPUs.

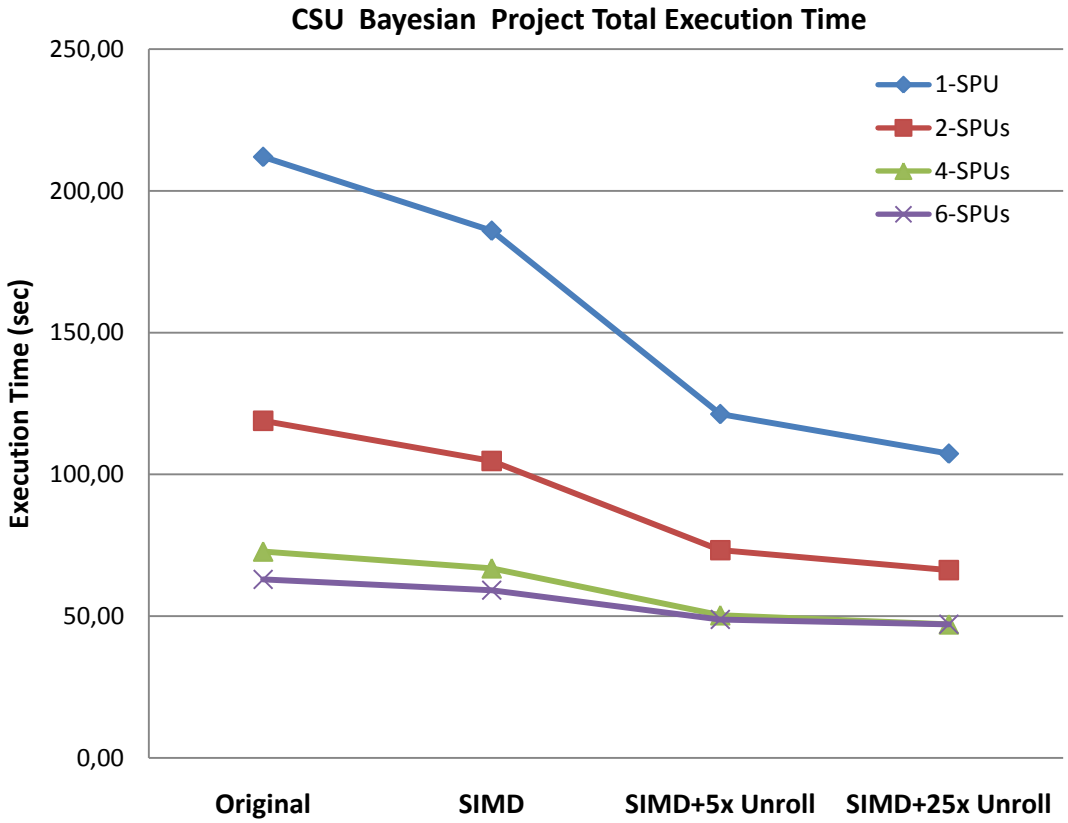

Figure 34: Total execution time of the Bayesian Project application

In the next figure, **Figure 35**, the main observation is the same as before, the half of the function-level speed-up is achieved. The speed-up of the CSU Bayesian Project is only *1.77x* and the main reasons for that is the scheduling overhead and the rest code running on PPE. As it was mentioned in paragraph *4.6.8*, the structure of the application does not allow us to "stay" at the SPEs; instead, the execution is passing from the PPE to the SPEs and back multiple times. This control trade-off from PPE to SPEs and round is increasing dramatically the scheduling overhead. Besides the scheduling and PPE overhead a small amount of the overall overhead is caused from the final addition and merging process.

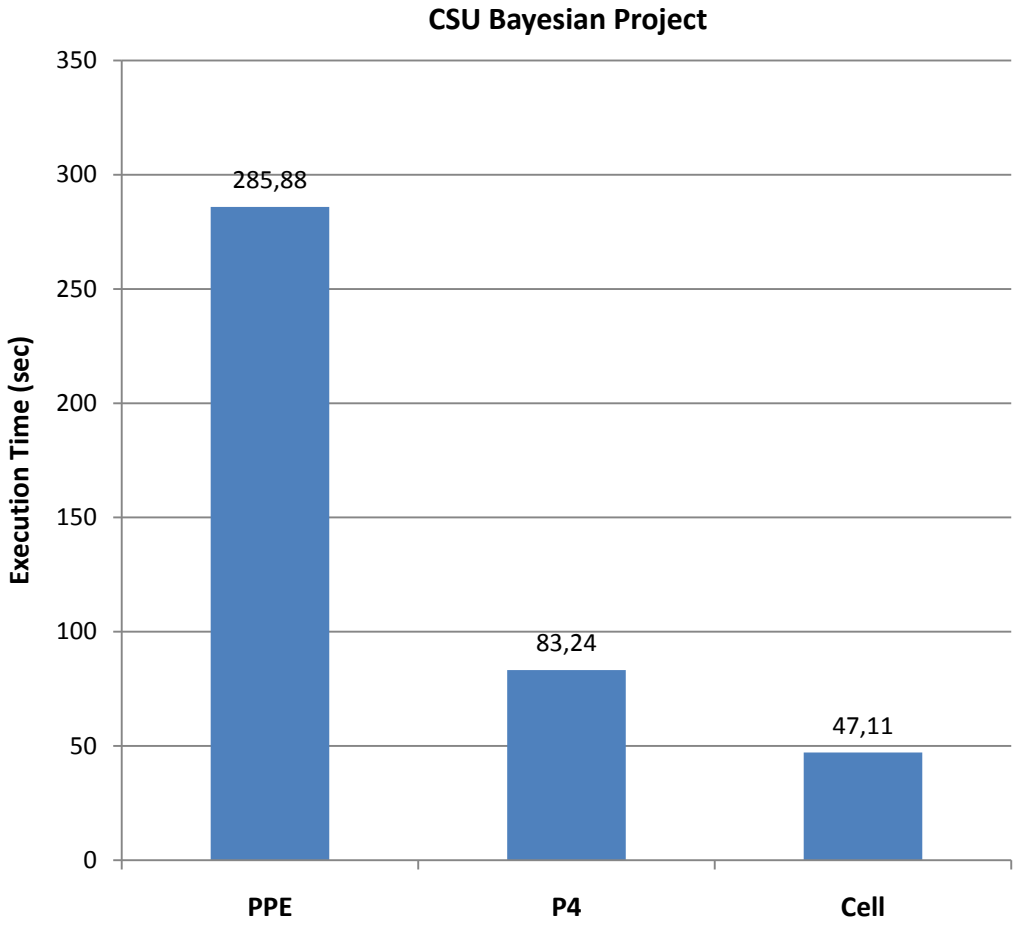

Figure 35: Performance comparisons for Bayesian Project

Table 22 shows the execution times for the different parts of the CSU Bayesian Project application. As the table shows a large amount of time is consumed for scheduling, contrary to the CSU Bayesian Train application.

	Execution Time (sec)	Percentage %
PPE code	16,33	35,07
SPE code	15,97	34,31
Addition	2,18	4,68
Merging	0,02	0,05
Scheduling	12,05	25,89

Table 22: Execution time analysis for CSU Bayesian Project

5.3 Precision

As it was mentioned in paragraph *4.6.2* the normal method of matrix multiplication compared with the submatrices multiplication method has a precision difference located after the tenth decimal digit. It is well known that floating-point addition is not associative so that the calculation sequence and the order in which the addends enter the floating-point calculation of the sum greatly influence the size of the accumulated *round-off error* of the result. In the case of normal matrix multiplication for the calculation of each element a sequence of additions and multiplication is being executed. When submatrices are being used this sequence is difference since the partial products are first evaluated for each pair of submatrices and then the final addition takes place.

The round-off error depends on the number of the final additions, meaning that it depends on the number of submatrices. In our case the maximum round-off error affects the result up to the tenth decimal digit. This does not affect the decision of the classifier; it has been observed that the image with the smaller distance from the probe has a significant difference from the other images. The only way this error can lead the classifier to the wrong decision is when two images had a difference between them less than 10^{-10} and the wrong image has the smaller distance from the probe. During all the tests that were done, never the round-off error caused a wrong decision.

5.4 Verification

The last but not least step of the design was the overall verification of the application. Until now the correctness of the multiplications was verified many times during the development process, now the final results of the application have to be verified. As it was mentioned in paragraph *3.5* the output of the application is a set of files, so a comparison of these files was made to verify the results. A result set produced by the execution of the original code was compared with a result set produced from the execution on PS3. Due to the large amount of data that is contained in these files, a script written on *Matlab* was used for faster verification.

The verification process was successful and all the results were the same, meaning that in both executions the algorithm was taking the same decision for each image. The difference on the decimal digits that was mentioned in the previous paragraph was not affecting the final results.

CHAPTER 6

Conclusions & Future Work

This chapter summarizes the contribution and the results of this work. Some final conclusions are being presented as well as and some thoughts for future work.

6.1 Conclusions

The main contribution of this work was the parallelization of the BIC algorithm and its execution on the Cell processor. Moreover the use of an existing application and its partial redesign for the Cell processor were also parts of this work contribution. The implementation of this design proved to be hard enough due to the inexperience on the development tools as well as and on parallel programming techniques. Multicore processors increase the performance but they also increase the complexity of the software development process, so matters like scheduling and synchronization must now be considered. Several problems came up during the development; most of them were related with software tools and design aspects, a close study of these problems let us to their resolve.

In the final results a considerable improvement of performance was achieved, but with a significant overhead too, which prevents us from reaching the desirable results. The main reason for this is the selection of a fast programming model, such as the function-offload model instead of a hard model like streaming model. The selection of a different model could possibly bring us better results with a significant cost on time and effort. This extra cost derives from the fact that these models demand the complete redesign of the application and the use of more complex techniques. However the selection of this model cannot be considered as wrong because it balances the main targets, performance, fast development and conservation of the original application logic. Another important factor, apart from the model that affects the final results is the structure of the algorithm and the application. For example the double precision data reduces the second level of parallelism (SIMD) to 2-way only, the variable number of iterations in some loops prevents the use of loop unrolling technique etc.

Concluding, Cell processor, although initially designed mainly for games and multimedia, is a very promising architecture for scientific computations, as well. But due to the absence of specialized software tools and the simple architectural design of the SPUs, the user is forced to consider a number of low-level details, which increase the complexity of the development. Below is shown a summary of the major design and tools problems that were encountered during this work:

- Data partitioning to fit in LS, paragraph *4.5*.
- Partitioning on PPE side was ineffective, paragraph *4.6.1*.
- DMA transfers limitations, paragraph *4.6.5*.
- Scheduling overhead due to multiple function calls in Bayesian Project, paragraphs *4.6.4* and *4.6.8*.
- Small computation time on SPEs in Bayesian Project, paragraph *4.6.8*.
- Unable to install SDK version 3.0 on server, paragraph 4.7.
- Bug at the execution of the simulator in cycle–mode, for version 2.1, paragraph 4.7.
- Simulator was extremely slow, paragraph 4.7.
- Development on PS3 has limited memory and SPEs, paragraph 4.7.

6.2 Future Work

This project is a complete work which has exhausted most of the possible improvements that could be done with the use of the function offload model. Any future work on the specific algorithm must focus on the complete redesign of the application and the use of a different model. Perhaps an implementation with the streaming model or any other SPE-centric model would result a better performance. Whatever programming model will be used the design must use as much as possible the SPEs and avoid the re-scheduling; by doing this the code running on the PPE will be reduced as well as the scheduling overhead. Below, some more specific ideas for future work are being proposed:

- For the CSU Bayesian Project application would be better to project more images at once instead of projecting one by one. This will increase the dimensions of the matrices for the Bayesian Project and will reduce the iterations.
- A vectorization of the code running on the PPE would probably increase its performance.

- If the use of single precision floating point is not dramatically affecting the results, then this will increase the second level parallelism from 2-way to 4-way.

References

[1] J. A. Kahle, et al., "Introduction to the Cell Multiprocessor," *IBM Systems Journal*, vol. 49, no. 4/5, pp. 589-605, 2005.

[2] Sony. Playstation. [Online]. http://gr.playstation.com/ps3/index.html

[3] A. Chow, G. Fossum, and D. Brokenshire, "A Programming Example: Large FFT on the Cell Broadband Engine," in *GSPx*, Santa Clara, 2005.

[4] D. Bader, V. Agarwal, K. Madduri, and S. Kang, "High Performance Combinatorial Algorithm Design on the Cell Broadband Engine," *Parallel Computing*, vol. 33, no. 10-11, pp. 720-740, 2007.

[5] F. Petrini, et al., "Multicore Suprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine," in *IEEE International Parallel and Distributed Processing Symposium*, Long Beach, 2007, p. 62.

[6] V. Sachdeva, M. Kistler, E. Speight, and T. K. Tzeng, "Exploring the Viability of the Cell Broadband Engine for Bioinformatics Applications," in *IEEE International Parallel and Distributed Processing Symposium*, Long Beach, 2007, p. 259.

[7] S. Williams, et al., "The potential of the Cell processor for scientific computing," in *Third Conference on Computing Frontiers CF'06*, New York, 2006, pp. 9-20.

[8] D. Bolme, R. Beveridge, M. Teixeira, and B. Draper, "The CSU Face Identification Evaluation System: Its Purpose, Features and Structure," in *International Conference on Vision Systems*, Graz, Austria, 2003, pp. 304-311.

[9] R. Beveridge. Evaluation of Face Recognition Algorithms. [Online]. http://www.cs.colostate.edu/evalfacerec

[10] B. Moghaddam and A. Pentland, "Probabilistic Visual Learning for Object Detection," in *IEEE International Conference on Computer Vision*, Cambridge,MA, 1995, pp. 786-793.

[11] B. Moghaddam, C. Nastar, and A. Pentland, "A Bayesian Similarity Measure for Direct Image Matching," in *International Conference on Pattern Recognition*, Vienna, Austria, 1996, pp. 350-358.

[12] M. L. Teixeira, "The Bayesian Intrapersonal/Extrapersonal Classifier," *Master's Thesis, CSU Computer Science Department*, Jul. 2003.

[13] D. Pham, et al., "The Design and Implementation of a First Generation Cell Processor," in *International Solid State Circuits Conference*, San Fransisco, 2005, pp. 184-185.

[14] T. Chen, R. Raghavan, J. Dale, and E. Iwata, "Cell Broadband Engine Architecture and its First Implementation," *IBM developerWorks techical article*, 2005.

[15] IBM, "Cell Broadband Engine Architecture," Oct. 2007, Version 1.02.

[16] IBM, "VectorSIMD Multimedia Extension Technology," Oct. 2006, Version 2.07c.

[17] IBM, "PowerPC 970FX RISC Microprocessor User's Manual," Dec. 2005, Version 1.6.

[18] M. Gschwind, et al., "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro*, vol. 26, no. 2, pp. 10-24, 2006.

[19] B. Flachs, et al., "The Microarchitecture of the Synergistic Processor for a Cell Processor," *IEEE Solid State Circuits*, vol. 41, no. 1, 2006.

[20] IBM, "SPU Instruction Set Architecture," Jan. 2007, Version 1.2.

[21] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro*, vol. 26, no. 3, pp. 10-23, 2006.

[22] Fedora. Fedora Project. [Online]. http://fedoraproject.org/

[23] TerraSoft. Yellow Dog Linux. [Online]. http://www.terrasoftsolutions.com/products/ydl/

[24] Gentoo. Gentoo Linux. [Online]. http://www.gentoo.org/

[25] Debian. Debian -- The Universal Operating System. [Online]. http://www.debian.org/

[26] NIST. (2003) FERET Database. [Online]. http://www.itl.nist.gov/iad/humanid/feret/

[27] G. Shakhnarovich and B. Moghaddam, "Face Recognition in Subspaces," in *Handbook of Face Recognition*. Springer-Verlag, 2004.

[28] IBM, "Cell Broadband Engine Programming Tutorial," Oct. 2007, Version 3.0.

[29] A. Arevalo, et al., *Programming the Cell Broadband Engine: Examples and Best Practices*, 1st ed. IBM, 2007.

[30] IBM, "Cell Broadband Engine Programming Handbook," Apr. 2007, Version 1.1.

[31] IBM, "SPE Runtime Management Library," Sep. 2007, Version 2.2.

[32] IBM, "SPU C/C++ Language Extensions," Mar. 2006, Version 2.1.

[33] D. A. Brokenshire, "Maximizing the Power of the Cell Broadband Engine Processor: 25 Tips to Optimal Application Performance," *IBM developerWorks technical article*, 2006.

[34] TerraSoft. Yellow Dog Linux-Installation Support. [Online]. http://www.terrasoftsolutions.com/support/installation/

[35] IBM. Cell Broadband Engine Resource Center. [Online]. http://www.ibm.com/developerworks/power/cell/documents.html

[36] IBM. Cell Broadband Engine Architecture Forum. [Online]. http://www.ibm.com/developerworks/forums/forum.jspa?forumID=739&cat=46

[37] IBM, "Full-System Simulator for the Cell Broadband Engine Processor," Oct. 2007, Version 3.0.

[38] IBM, "Performance Analysis with the Full-System Simulator," Oct. 2007, Version 3.0.

[39] IBM. IBM BladeCenter QS20. [Online]. http://www-03.ibm.com/technology/splash/qs20/