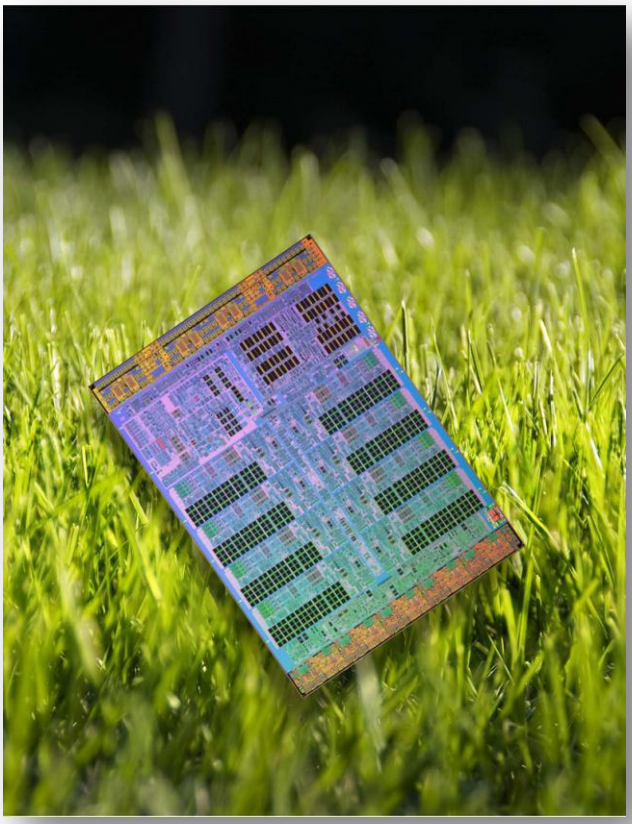
Technical University of Crete

Department of Electronic & Computer Engineering

Parallelization of the Gait Baseline Algorithm for the Cell Broadband Engine



Sinapidis Evangelos

Diploma Thesis

Selection Committee

Prof. Dionisios Pnevmatikatos (Supervisor)

Prof. Apostolos Dollas

Prof. Ioannis Papaefstathiou

25/02/2010

Abstract

Nowadays, computer science evolves in high rate. Within short time, new technologies are invented to respond our ever increasing demands. In order to fulfill these needs, the traditional processor architecture has begun to change and new remarkable and alternative methods have developed.

Especially, single core processor architecture is at its end while multi core processor architecture is evolving. Multi core processors have the ability to execute multiple threads at the same time. Due to this capability and in proportion with the parallelization that can achieve by each architecture, the performance is multiple. Cell Broadband Engine (Cell BE) is one of the new technologies processor and it is produced by IBM, Toshiba and Sony.

In this thesis, we endeavor to use the parallelism that is provided by Cell BE so as to accelerate the human gait recognition. The algorithm which was studied is Gait Baseline Algorithm. This algorithm faces the difficulties in human gait recognition and establishes an infrastructure for the experiment methods, the data amount and the recognition algorithm with the aim of further evolution of algorithm akin to Gait Baseline Algorithm.

The procedure was complicated enough and there were technical difficulties in hardware level. Almost all of them were overcome successfully by using alternative methods of classic programming or by the specialized capabilities of Cell BE. Particularly, we parallelized the algorithm using the function offload method, splitting computing intensive functions in multiple Synergistic Processor Elements (SPEs). Simultaneously we optimized the code using three optimization techniques SIMD, function inline and loop unrolling.

The results were satisfactory as we achieved a considerable improvement of performance. The speedup was 5.51 times faster compared to Power Processor Element, 2.18 times faster than an Intel E 6400 2.13GHz processor and 3.00 times faster compared to an Intel T 2400 1.83GHz or 4.13 times faster at 987MHz. The knowledge that was obtained is essential in order to study similar, complex computing applications of new methods and architectures

This thesis is based on The HumanID Gait Challenge Problem which is the base of the future in gait recognition. Programming modern multi-core processor and gait recognition knowledge were demanded. The improved modified code of Baseline Algorithm introduces a significant evolution and could be matter for further research.

Acknowledgements

Many people's assistance was important in order to accomplish to complete this thesis. The work had been demanding enough and I could not complete it without the people I had around me.

I would like to thank my supervisor professor D. Pnevmatikatos for the help he offered to me all this time. His proposals and solutions on the problems, that I encountered, accelerate the completion of this work and helped to reach the desirable result.

I also like to thank professors I. Papaefstathiou and A. Dollas. All these years of study, all three professors led me through teaching to obtain a correct way of thinking about confrontation and resolution of any problem.

I should also acknowledge the culture and knowledge gained through the Technical University of Crete. After completing my studies and leaving this department, I feel that I am actually an engineer.

Finally I am grateful for the support that my family was gave to me all these years of study and I owe to them the biggest part of my success.

Table of Contents

List of Figures

List of Tables

Introduction

The goal of this thesis is Gait Baseline Algorithm acceleration through proper understanding and use of Cell Broadband Engine (Cell BE) processor capabilities. For this reason, the algorithm did not develop from the beginning. We focused on the most computing intensive points, in order to parallelize them and insomuch that improve the algorithm performance. This thesis is structured by the method we follow to reach the problem solution:

- Chapter 1, Cell BE and Playstation 3 capabilities explanation, on which the search was based.
- Chapter 2, The HumanID Gait Challenge Problem (Data Sets, Experiments, Baseline Algorithm) description.
- Chapter 3, analytical description of the implementation
- Chapter4, reference of search results and the conclusion to which we were led.
- Chapter 5, conclusions and proposals for future implementations.

Contribution

The human gait recognition could be evolved if it is combined with edge technology (Cell BE). We face speedup of gait recognition algorithms, aiming to implement the algorithm to different processor architectures than those already have been used. The HumanID Gait Challenge Problem established new parameters to gait recognition and corrected common mistakes of the studies that already have been published. These characteristics of the project make the study and its improvement important as in fact they underpin the future of gait recognition. To face this issue we had to study both programming modern multi-core processor and gait recognition. In this thesis it is raised the opportunity to face correctly the gait recognition in the future from the aspect of new multi-core processors architectures and alternative solutions, such as GPUs and FPGAs.

1 Cell Broadband Engine

1.1 Introduction to Cell Broadband Engine

1.1.1 History [2]

The Cell BE [3] is a product of multi-core processors technology. The Cell BE processor is the first implementation of a new multiprocessor family conforming to the Cell Broadband Engine Architecture (CBEA). [4] The CBEA and the Cell BE processor are the result of collaboration between Sony, Toshiba, and IBM known as STI [18], formally begun in early 2001. The STI Design Center opened in March 2001. [2] The Cell was designed over a period of four years, using enhanced versions of the design tools for the Power4 processor. Over 400 engineers from the three companies worked together in Austin, with critical support from eleven of IBM's design centers.

In March 2007 IBM announced that the 65 nm version of Cell BE is in production at its plant in East Fishkill, New York. Again in February 2008, IBM announced that it will begin to fabricate Cell BE processors with the 45 nm process. In May 2008, IBM introduced the high-performance double-precision floating-point version of the Cell BE processor, the PowerXCell 8i, at the 65 nm feature size. In May 2008, an Opteron- and PowerXCell 8i-based supercomputer, the IBM Roadrunner system, became the world's first system to achieve one petaFLOPS, and was the fastest computer in the world until fall 2009. The world's three most energy efficient supercomputers, as represented by the Green500 list, are similarly based on the PowerXCell 8i.The 45 nm Cell BE processor was introduced in concert with Sony's PlayStation 3 Slim in August 2009. In November 2009, an IBM representative said that it has discontinued the development of a Cell BE processor with 32 SPUs but they have not halted development of other future products in the Cell BE family. [2]

1.1.2 Motivation [1]

Although the Cell BE processor is initially intended for applications in media-rich consumer-electronics devices such as game consoles and high-definition televisions, the architecture has been designed to enable fundamental advances in processor performance. These advances are expected to support a broad range of applications in both commercial and scientific fields.

The CBEA [4] has been designed to support a very broad range of applications. The first implementation is a single-chip multiprocessor with nine processor elements operating on a shared memory model, as shown in Figure 1.1. In this respect, the Cell BE processor extends current trends in PC and server processors. The most distinguishing feature of the Cell BE processor is that, although all processor elements can share or access all available memory, their function is specialized into two types: the PowerPC Processor Element (PPE) [19] and the Synergistic Processor Element (SPE) [20]. The Cell BE processor has one PPE and eight SPEs.

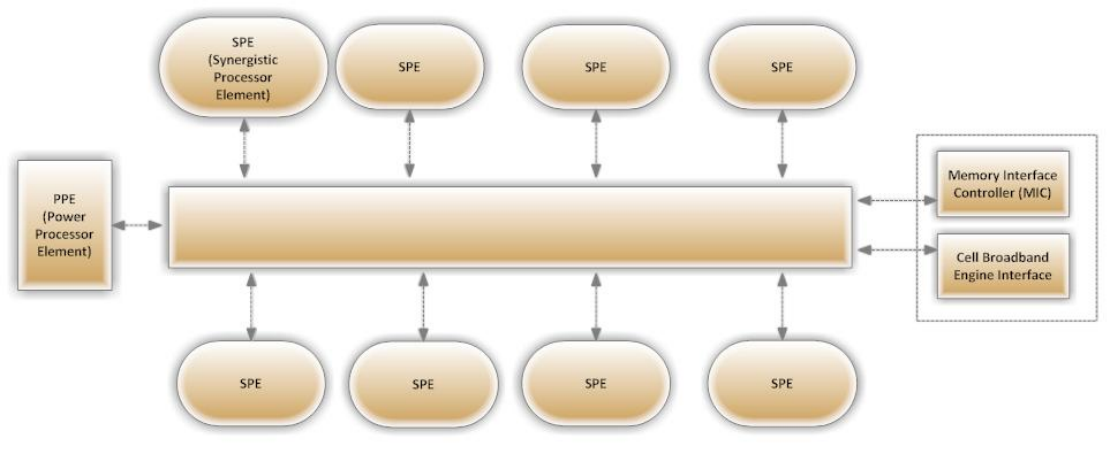


Figure 1-1 Cell BE Block Diagram

The first type of processor element, the PPE, contains a 64-bit PowerPC Architecture core. It complies with the 64-bit PowerPC Architecture [19] and can run 32-bit and 64-bit operating systems and applications. The second type of processor element, the SPE, is optimized for running compute-intensive SIMD applications; it is not optimized for running an operating system. The SPEs are independent processor elements, each running their own individual application programs or threads. Each SPE has full access to shared memory, including the memory-mapped I/O space implemented by multiple DMA units. There is a mutual dependence between the PPE and the SPEs. The SPEs depend on the PPE to run the operating system, and, in many cases, the top-level thread control for an application. The PPE depends on the SPEs to provide the bulk of the application performance.

The SPEs are designed to be programmed in high-level languages, such as (but certainly not limited to) C/C++. They support a rich instruction set that includes extensive SIMD functionality. However, like conventional processors with SIMD extensions, use of SIMD data types is preferred, not mandatory. For programming convenience, the PPE also supports the standard PowerPC Architecture instructions and the vector/SIMD multimedia extensions.

To an application programmer, the Cell BE processor looks like a single core, dual threaded processor with 8 additional cores each having their own local store. The PPE is more adept than the SPEs at control-intensive tasks and quicker at task switching. The SPEs are more adept at compute-intensive tasks and slower than the PPE at task switching. However, either processor element is capable of both types of functions. This specialization is a significant factor accounting for the order-of-magnitude improvement in peak computational performance and chip-area-and-power efficiency that the Cell BE processor achieves over conventional PC processors.

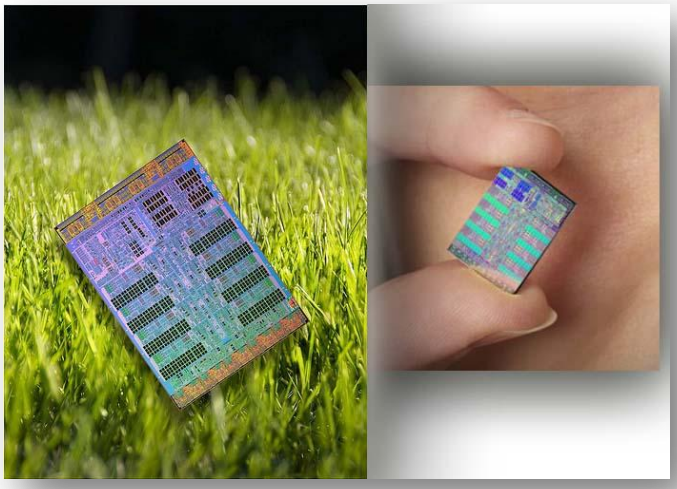


Figure 1-2 Cell Broadband Engine

The more significant difference between the SPE and PPE lies in how they access memory. The PPE accesses main storage (the effective-address space) with load and store instructions that move data between main storage and a private register file, the contents of which may be cached. PPE memory access is like that of a conventional processor technology, which is found on conventional machines. The SPEs, in contrast, access main storage with direct memory access (DMA) commands that move data and instructions between main storage and a private local memory, called a local store or local storage (LS). An SPEs instruction-fetches and load and store instructions access its private LS rather than shared main storage, and the LS has no associated cache. This 3-level organization of storage (register file, LS, main storage), with asynchronous DMA transfers between LS and main storage, is a radical break from conventional architecture and programming models, because it explicitly parallelizes computation with the transfers of data and instructions that feed computation and store the results of computation in main storage.

One of the motivations for this radical change is that memory latency, measured in processor cycles, and has gone up several hundredfold from about the years 1980 to 2000. The result is that application performance is, in most cases, limited by memory latency rather than peak compute capability or peak bandwidth. When a sequential program on a conventional architecture performs a load instruction that misses in the caches, program execution can come to a halt for several hundred cycles (techniques such as hardware threading can attempt to hide these stalls, but it does not help single threaded applications). Compared to this penalty, the few cycles it takes to set up a DMA transfer for an SPE are a much better trade-off, especially considering the fact that each of the eight SPEs DMA controller can have up to 16 DMA transfer in flight simultaneously. Anticipating DMA needs efficiently can provide “just in time delivery” of data which many reduce this stall or eliminate them entirely. Conventional processors, even with deep and costly speculation, manage to get, at best, a handful of independent memory accesses in flight.

One of the SPEs DMA transfer methods supports a list (such as a scatter-gather list) of DMA transfers that is constructed in an SPEs local store, so that the SPEs DMA controller can process the list asynchronously while the SPE operates on previously transferred data. In several cases, this approach to accessing memory has led to application performance exceeding that of conventional processors by almost two orders of magnitude significantly more than one would expect from the peak performance ratio (approximately 10x) between the Cell BE processor and conventional PC processors. The DMA transfers can be set up and controlled by the SPE that is sourcing or receiving the data or in some circumstances by the PPE or another SPE.

1.2 Hardware Overview

1.2.1 PowerPC Processor Element [3]

The PowerPC Processor Element (PPE) is a general-purpose, dual-threaded, 64-bit RISC processor that conforms to the PowerPC Architecture, version 2.02, with the vector/SIMD multimedia extensions. Programs written for the PowerPC 970 processor, for example, should run without modification on the CBEA processors.

The PPE is responsible for overall control of a system. It runs the operating systems for all applications running on the PPE and SPEs. The PPE consists of two main units, the PowerPC processor unit (PPU) and the PowerPC processor storage subsystem (PPSS), shown in Figure 1.3.

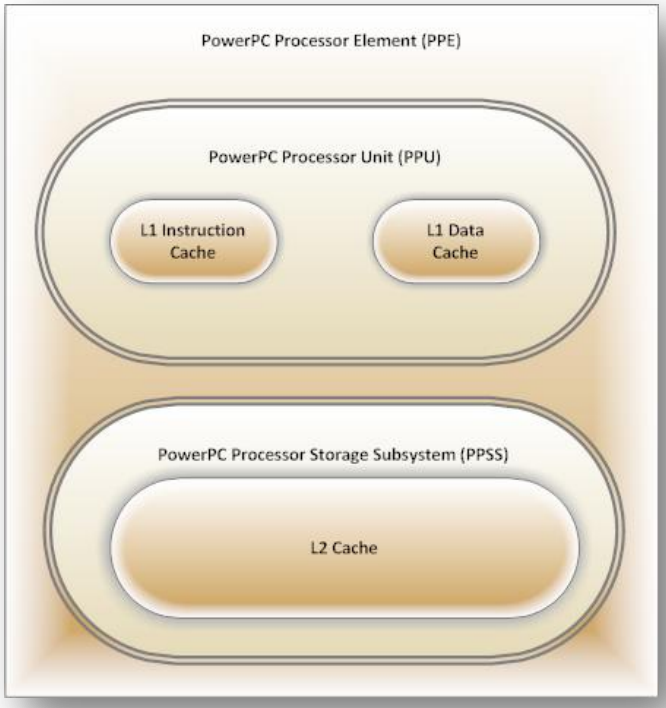


Figure 1-3 PPE Bloch Diagram

The PPU performs instruction execution. It has a level-1 (L1) instruction cache and data cache and six execution units. It can load 32 bytes and store 16 bytes, independently and memory coherently, per processor cycle. The PPSS handles memory requests from the PPU and external requests to the PPE from SPEs or I/O devices. It has a unified level-2 (L2) instruction and data cache. The PPU and the PPSS and their functional units are shown in Figure 1.4.

The PPU executes the PowerPC Architecture instruction set and the vector/SIMD multimedia extension instructions [17]. It has duplicate sets of the PowerPC and vector user-state register files (one set for each thread) plus one set of the following functional units:

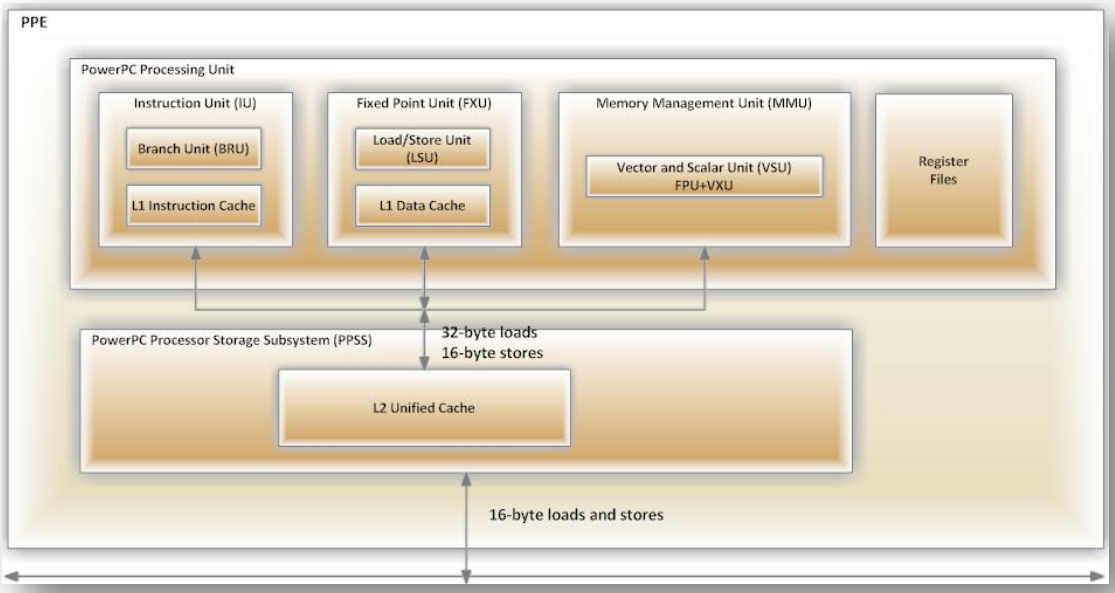


Figure 1-4 PPE Functional Units

- Instruction Unit (IU) The IU performs the instruction-fetch, decode, dispatch, issue, branch, and completion portions of execution. It contains the L1 instruction cache, which is 32 KB, 2-way set-associative, reload-on-error, and parity protected. The cache-line size is 128 bytes.
- Load and Store Unit (LSU) the LSU performs all data accesses, including execution of load and store instructions. It contains the L1 data cache, which is 32 KB, 4-way set-associative, write-through, and parity protected. The cache-line size is 128 bytes.
- Vector/Scalar Unit (VSU) The VSU includes a floating-point unit (FPU) and a 128-bit vector/SIMD multimedia extension unit (VXU), which together executes floating-point and vector/SIMD multimedia extension instructions.
- Fixed-Point Unit (FXU) The FXU executes fixed-point (integer) operations, including add, multiply, divide, compare, shift, rotate, and logical instructions.
- Memory Management Unit (MMU) The MMU manages address translation for all memory accesses. It has a 64-entry segment lookaside buffer (SLB) and 1024-entry, unified, parityprotected translation lookaside buffer (TLB). It supports three simultaneous page sizes 4 KB, plus two sizes selectable from 64 KB, 1 MB, or 16 MB.

The 128-bit VXU operates concurrently with the FXU and the FPU, as shown Figure 1-5. All vector/SIMD multimedia extension instructions are designed to be easily pipelined. Parallel execution with the fixed-point and floating-point instructions is simplified by the fact that vector/SIMD multimedia extension instructions do not generate exceptions (other than data-storage interrupts on loads and stores), do not support complex functions, and share few resources or communication paths with the other PPE execution units.

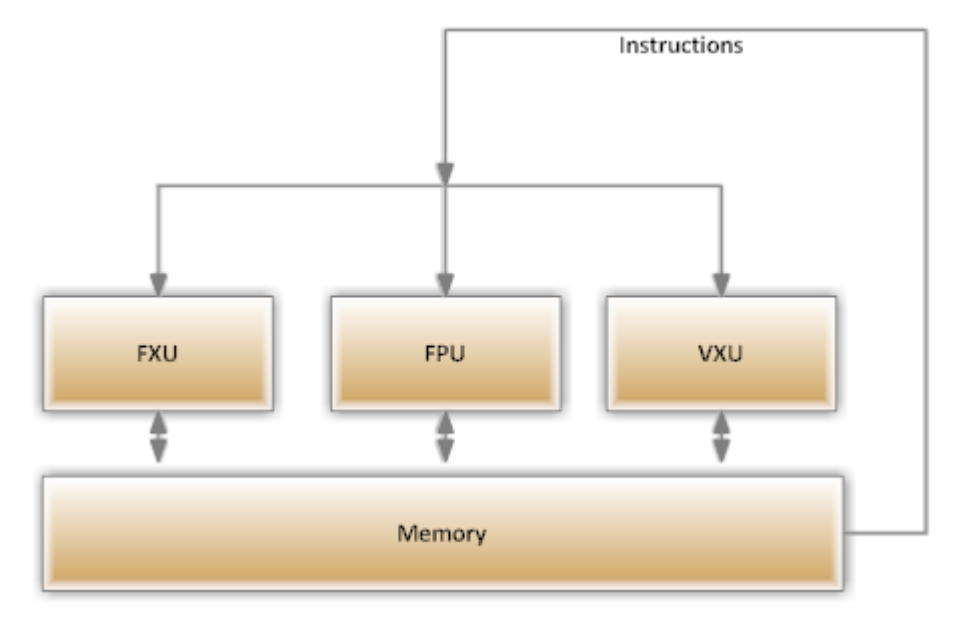


Figure 1-5 Concurrent Execution of Fixed-Point, Floating-Point, and Vector Extension Units

The PPSS handles all memory accesses by the PPU and memory-coherence (snooping) operations from the element interconnect bus (EIB). The PPSS has a unified, 512 KB, 8-way set-associative, write-back L2 cache with error-correction code (ECC). Like the L1 caches, the cache-line size for the L2 is 128 bytes. The cache has a single-port read/write interface to main storage that supports eight software-managed data-prefetch streams. It includes the contents of the L1 data cache but is not guaranteed to contain the contents of the L1 instruction cache, and it provides fully coherent symmetric multiprocessor (SMP) support. The PPSS performs data-prefetch for the PPU and bus arbitration and pacing onto the EIB. Traffic between the PPU and PPSS is supported by a 32-byte load port (shared by MMU, L1 instruction cache, and L1 data cache requests), and a 16-byte store port (shared by MMU and L1 data cache requests). The interface between the PPSS and EIB supports 16-byte load and 16-byte store buses. One storage access occurs at a time, and all accesses appear to occur in program order. The interface supports resource allocation management, which allows privileged software to control the amount of time allocated to various resource groups. The L2 cache and the TLB use replacement management tables, which allow privileged software to control the use of the L2 and TLB. This software control over cache and TLB resources is especially useful for real-time programming.

1.2.2 Synergistic Processor Element [3]

The eight Synergistic Processor Elements (SPEs) execute a new single instruction, multiple data (SIMD) instruction set, the Synergistic Processor Unit Instruction Set Architecture. Each SPE is a 128-bit RISC processor specialized for data-rich, compute-intensive SIMD and scalar applications. It consists of two main units, the synergistic processor unit (SPU) and the memory flow controller (MFC), shown in Figure 1.6.

The SPEs provide a deterministic operating environment. They do not have caches, so cache misses are not a factor in their performance. Pipeline-scheduling rules are simple, so it is easy to statically determine the performance of code and for software to generate high-quality, static schedules.

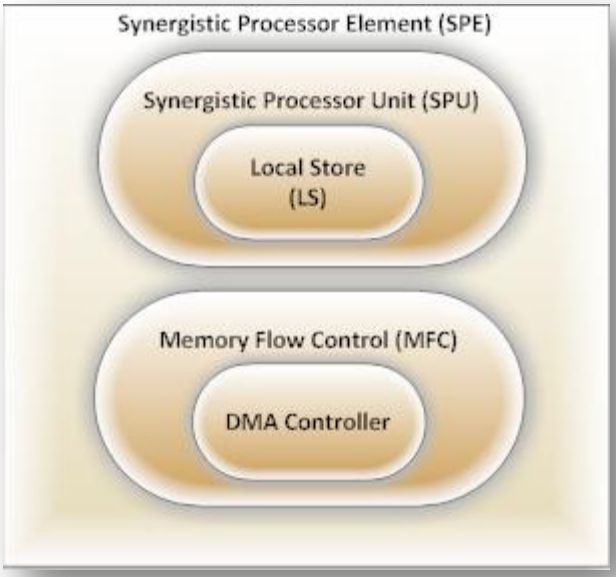


Figure 1-6 SPE Block Diagram

The intent of the synergistic processor unit (SPU) is to fill a void between general-purpose processors and special-purpose hardware. General-purpose processors aim to achieve the best average performance on a broad set of applications. Special-purpose hardware aims to achieve the best performance on a single application. The SPU, however, aims to achieve leadership performance on critical workloads for game, media, and broadband systems. The intent of the SPU and the Cell Broadband Engine Architecture (CBEA) is to provide a high degree of control to expert (real-time) programmers while maintaining ease of programming.

The SPU implements its own instruction set architecture (ISA). The main characteristics of this architecture are:

- Load-store architecture with sequential semantics, using a set of 128 registers, each of which is 128 bits wide.
- Single-instruction, multiple-data (SIMD) capability
  - Sixteen 8-bit integers
  - Eight 16-bit integers
  - Four 32-bit integer or four single-precision floating-point values
  - Two double-precision floating point
- SPU load and store instructions access only the associated local storage register
- Channel input/output for memory flow controller (MFC) control (used for external data access)

The SPU has the following restrictions:

- No direct access to main storage (access to main storage using MFC facilities only)
- No distinction between user mode and privileged state
- No access to critical system control such as page-table entries (PowerPC Processor Element [PPE] privileged software should enforce this restriction)
- No synchronization facilities for shared local storage access

The intent of the SPU is to enable applications that require a high computational unit density and that can effectively use the instruction set provided. A significant numbers of SPU cores in a system, managed by a PPE, allows for cost-effective processing over a wide range of applications. [4]

The SPU fetches instructions from its unified (instructions and data) 256 KB local storage (LS), and it loads and stores data between its LS and its single register file for all data types, which have 128 registers, each 128 bits wide. The SPU has four execution units, a DMA interface, and a channel interface for communicating with its MFC, the PowerPC Processor Element (PPE) and other devices (including other SPEs).

Each SPU is an independent processor element with its own program counter, optimized to run SPU programs. The SPU fills its LS by requesting DMA transfers from its MFC, which implements the DMA transfers using its DMA controller. Then, the SPU fetches and executes instructions from its LS, and it loads and stores data to and from its LS

The main SPU functional units are shown in Figure 1.7. These include the synergistic execution unit (SXU), the LS, and the SPU register file unit (SRF). The SXU contains six execution units. The SPU can issue and complete up to two instructions per cycle, one on each of the two (odd and even) execution pipelines. Whether an instruction goes to the odd or even pipeline depends on the instruction type. The instruction type is also related to the execution unit that performs the function.

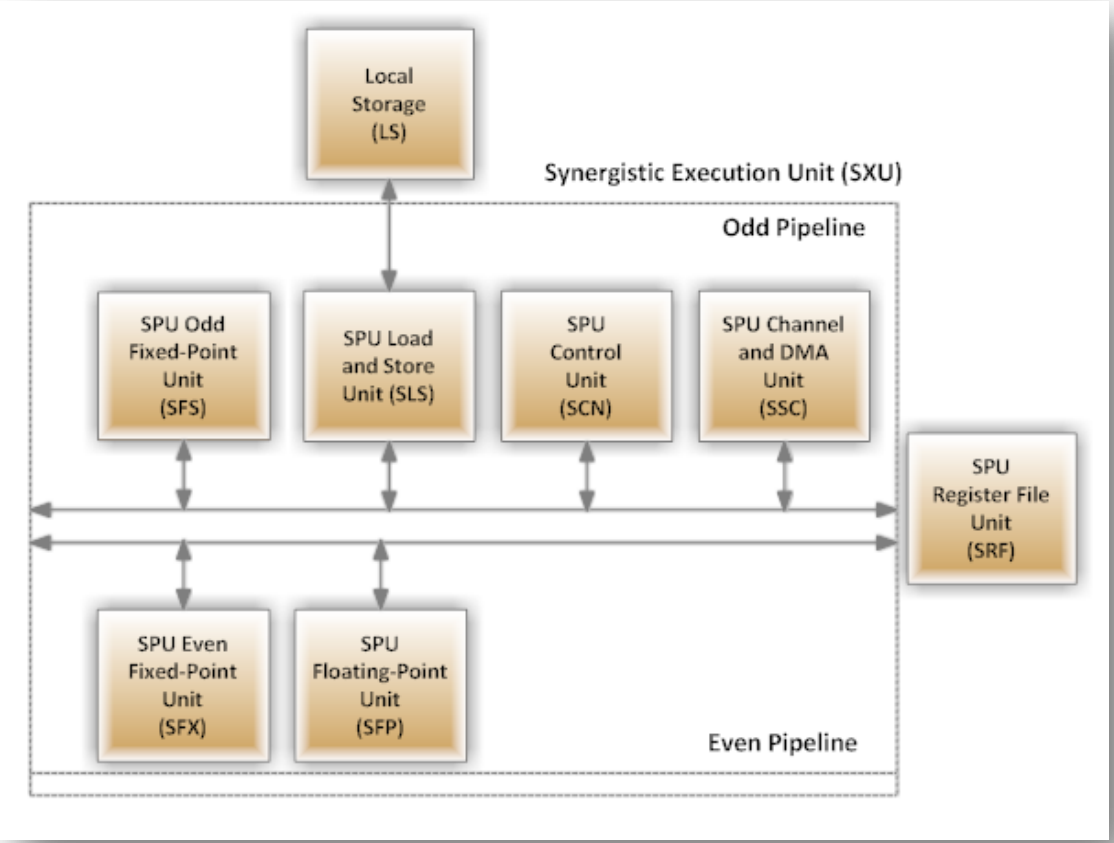


Figure 1-7 SPU Functional Units

The LS is a 256 KB, error-correcting code (ECC)-protected, single-ported, non caching memory. It stores all instructions and data used by the SPU. It supports one access per cycle from either SPE software or DMA transfers. SPU instruction prefetches are 128 bytes per cycle. SPU data access bandwidth is 16 bytes per cycle, quadword aligned. DMA-access bandwidth is 128 bytes per cycle. DMA transfers perform a read-modify-write of LS for writes less than a quadword.

The SPU accesses its LS with load and store instructions, and it performs no address translation for such accesses. Privileged software on the PPE can assign effective-address aliases to LS. This enables the PPE and other SPEs to access the LS in the main-storage domain. The PPE performs such accesses with load and store instructions, without the need for DMA transfers. However, other SPEs must use DMA transfers to access the LS in the main-storage domain. When aliasing is set up by privileged software on the PPE, the SPE that is initiating the request performs address translation.

1.2.3 Element Interconnect Bus [5]

The EIB is the communication path for commands and data between all processor elements on the Cell BE processor and the on-chip controllers for memory and I/O. The EIB supports full memory-coherent and symmetric multiprocessor operations. The EIB manages four 16-byte-wide data rings, which interconnect all units on the chip. Each ring transfers 128 bytes at a time. Two rings run clockwise, and two rings run counterclockwise, Figure 1.8 [6]. Each unit has one on-ramp and one off-ramp. Units attached to the rings can drive and receive simultaneously. Multiple transfers can be in-process concurrently on each ring. The EIB internal maximum bandwidth is 96 bytes per processor cycle, and it can support more than 100 outstanding DMA memory requests between main storage and the SPEs. The EIB does not support any particular quality-of-service (QoS) behavior other than to guarantee forward progress. However, the EIB contains a token manager unit, and software can use it to regulate the rate at which particular devices are allowed to make EIB command requests.

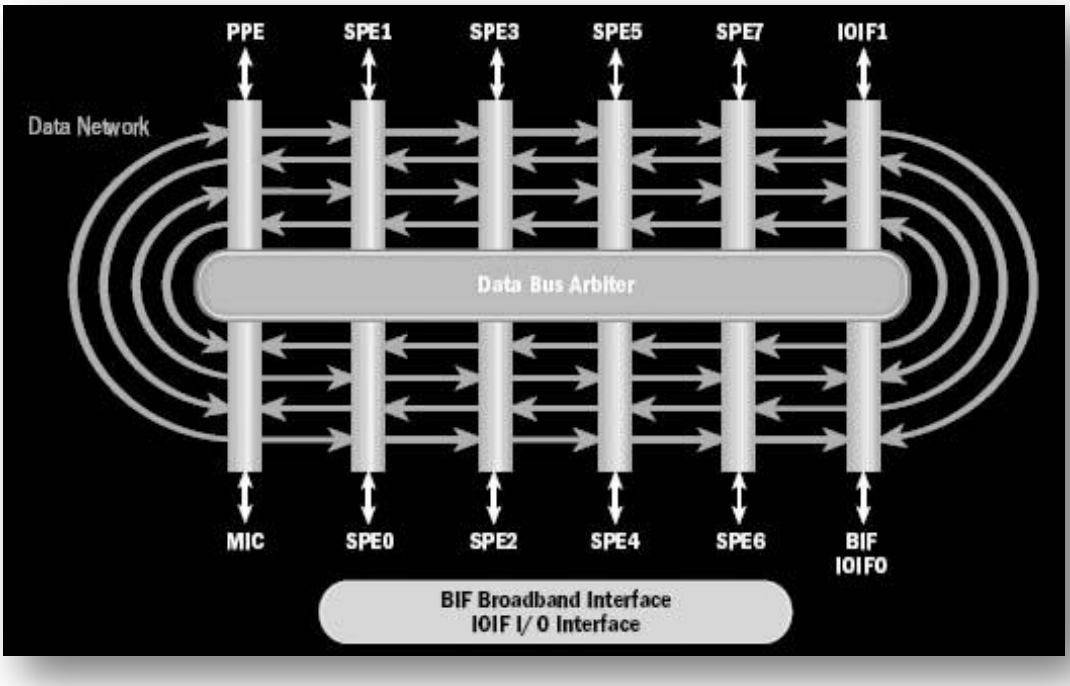


Figure 1-8 Element Interconnect Bus Diagram

1.3 1.3 PlayStation 3

1.3.1 Central Processing Unit

The PS3, Figure 1.9, uses the Cell microprocessor [15], which is made up of one 3.2 GHz PowerPC-based "Power Processing Element" (PPE) and six accessible Synergistic Processing Elements (SPEs). [7] A seventh runs in a special mode and is dedicated to aspects of the OS and security, and an eighth is a spare to improve production yields. PlayStation 3's Cell CPU achieves 204 GFLOPS single precision float and 15 GFLOPS double precision. The PS3 has 256MB of Rambus XDR DRAM, clocked at CPU die speed. As of firmware update 2.01, 32MB of the XDR memory is reserved by the PS3's XrossMediaBar user interface, more XDR memory is required for multiple XMB operations to function at one time.

1.3.2 Graphics Processing Unit

The graphics processing unit [16], according to Nvidia, is based on the NVIDIA G70 (previously known as NV47) architecture. The GPU makes use of 256MB GDDR3 RAM clocked at 700 MHz with an effective transmission rate of 1.4 GHz and up to 224MB of the 3.2 GHz XDR main memory via the CPU (480MB max).

1.3.3 1Operating System

The PlayStation 3 in selected models is capable of running Linux as well as other operating systems if installed on the console's hard drive. Many distributions are compatible with the console.

- Debian [8]
- Fedora 8 [9]
- Gentoo [10]
- OpenSUSE [11]
- Ubuntu [12]
- Yellow Dog Linux [13]

Any Linux operating system has access to 6 of the 7 Synergistic Processing Elements; Sony implements a hypervisor restricting access to the RSX. IBM provides an introduction to programming parallel applications on the PlayStation 3. The PlayStation 3 Slim model removed the possibility to install Linux or any other operating systems using the "Other OS" feature. [7] [15] [16]

1.3.4 Connectivity

The PS3 supports numerous SDTV and HDTV resolutions (from 480i/576i up to 1080p) and connectivity options (such as HDMI 1.3a and component video). In terms of audio, the PS3 supports a number of formats, including 7.1 digital audio, Dolby TrueHD, DTS-HD Master Audio and others; audio output is possible over stereo RCA cables (analog), optical digital cables, or HDMI. For the optical disc drive, a wide variety of DVD and CD formats are supported, as well as Blu-ray Discs. A 20, 40, 60, 80, 120, 160 or 250 GB 2.5 in SATA 150 hard disk is pre-installed. In the 60 GB and 80 GB configurations, flash memory can also be used, either Memory Sticks;

CompactFlash cards; or SD/MMC cards. All models support USB memory devices; flash drives and external hard drives are both automatically recognized. However, they must be formatted with the FAT32 file system—the PS3 does not support the NTFS file system that is the standard in the Windows NT family. For communication, the system sported four USB 2.0 ports at the front on the 20 and 60 GB models as well as the NTSC 80 GB model, but the 40 GB and 80 GB PAL models only have 2 USB ports. All models (80 and 160 GB) released after August 2008 have been reduced to two USB ports, as well as dropping CompactFlash and SD card support. One Gigabit Ethernet port, Bluetooth 2.0 support, and built-in Wi-Fi are available on the 40, 60, 80, 120, and 160 GB versions. [15]

1.3.5 Universal Power Supply

The power supply [15] can operate on both 60 Hz and 50 Hz power grids. It uses a standard C14 IEC connector and a C13 power cord appropriate for the region it is being used in. The power supply on the "fat" model is capable of delivering approximately 380 W, although the PS3 has never been measured using this much power. The power supply was reduced to 250 watts in the 120 GB "Slim" model.

1.3.6 Disc Drive

The PlayStation 3 disc drive [15] is an all-in-one type allowing the use of different formats. The Blu-ray drive is a 2× speed, the DVD drive is an 8× speed and the Compact Disc drive is a 24× speed.

Figure 1-9 Playstation 3

2 The Human Gait Challenge Problem [14]

2.1 Introduction

Gait is the pattern of movement of humans, over a surface (walking). There is a variety of gaits, selecting gait based on speed, terrain, the need to maneuver, and energetic efficiency. Each human has different gait due to differences in anatomy that prevent use of certain gaits, or simply due to evolved innate preferences as a result of habitat differences. The complexity of biological systems and interacting with the environment make gait distinctions confused at best. Gaits are typically classified according to footfall patterns, but recent studies often prefer definitions based on mechanics. [37]

The human gait study has launched several years ago. From the aspect of Psychology, a seldom recent search proved that people have the ability to recognize faster human movement from other kind of movement. Since then several experiments have been done in order to show that human can recognize the gender, the motion direction even if possible object that may be carried. Especially for gait recognition, most of searches have been focused on discriminations of human motion kinds, such as running, walking, jogging or climbing stairs.

It is only recently that human gait recognition (HumanID) has attracted the attention and been investigated as an active sector of computer vision. The majority of papers report results, limited in data set size, less than 30 people, taken indoors or under a limited number of conditions. These papers have developed gait recognition from its begging. In order to mature and estimate the potential of such a paper, larger and more various data sets are required.

HumanID Gait Challenge Problem [14] has been motivated from evolution of gait recognition, while the answers of the questions below were one more incentive:

- Is progress being made in gait recognition of humans?
- To what extent does gait offer potential as an identifying biometric?
- What factors affect gait recognition and to what extent?
- What are the critical vision components affecting gait recognition from video?
- What are the strengths and weaknesses of different gait recognition algorithms?

HumanID Gait Challenge Problem has evolved gait recognition by providing a foundational framework in reference to these issues. It includes a developed data set of 12 experiments and a baseline algorithm. The Baseline Algorithm represents a performance report and an initial characterization of automatic gait recognition.

The data sets of HumanID Gait Challenge Problem have collected outdoors. The outdoor choice is based on observations that

- Several indoor data sets are available,
- Nearly perfect gait recognition performances have been reported on indoor data sets, and
- Gait biometrics is most appropriate in outdoor at-a-distance settings, where other biometric sources are harder to acquire.

Each data set corresponds to HumanID Gait Challenge Problem that includes the gait variations based on five covariates. These covariates were chosen because they affect either human's gait or the extraction of gait features from images. For instance, surface type, shoe-wear type, and weight carried affect human gait in outdoor. Video data are also affected from the point of view. Furthermore, human gait could differ from time to time. It is important to understand the meaning of the factors above. These five covariates were selected from a larger list of intended factors that affected human's gait. There are also possible interesting covariates such as the mood of a person, clothing, speed, and backpack, which were not in view.

Two different conditions were chosen for each of these five covariates:

- two camera angles,
- two shoe types,
- two surfaces (grass and concrete),
- with and without carrying a briefcase, and
- two different dates six months apart.

It was attempted to acquire a person’s gait in all possible combinations of these five factors and so there are up to 32 sequences for some persons. The baseline algorithm is based on spatial-temporal correlation between silhouettes. In order to reduce the effects of clothing texture artifacts, comparisons are made with the silhouettes.

2.2 Data Set [14]

The HumanID gait challenge problem data set was designed to advance the state-of-the-art in automatic gait recognition and to characterize the effects on performance of five conditions. These two goals were achieved by collecting data on a large data set consisted of 122 persons that fulfills the requirements of current standards in gait.

The individuals were submitted to 32 different conditions, which were the result of all possible combinations of the five covariates with two values each. Each of the five covariate has two values, all possible combinations of these values are the 32 different conditions to which the individuals were submitted.

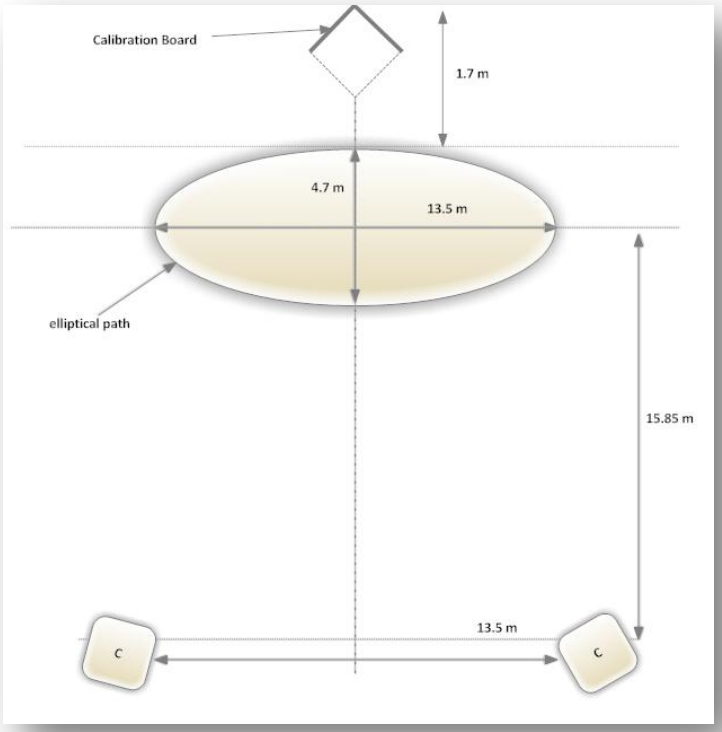


Figure 2-1 Camera setup for the gait data acquisition

During the sampling, each subject walked several times, counterclockwise, around each of two similar sized and shaped elliptical courses. The basic setup is illustrated in Figure 2.1. The elliptical courses were approximately 15 meters on the major axis and five meters on the minor axis. Both courses were outdoors.

One course was laid out on a flat concrete walking surface. The other was laid out on a typical grass lawn surface. Each course was viewed by two cameras, whose lines of sight were not parallel, but verged at approximately 30 degrees, so that the whole ellipse was the same visible from each of the two cameras. When a person walked along the rear portion of the ellipse, their view was approximately fronto-parallel. Figure 2.2 shows one sample frame from each of the four cameras on the two surfaces. The orange traffic cones marked the major axes of the ellipses. The checkered object in the middle is a calibration object that can be used by future algorithms to calibrate the two cameras.

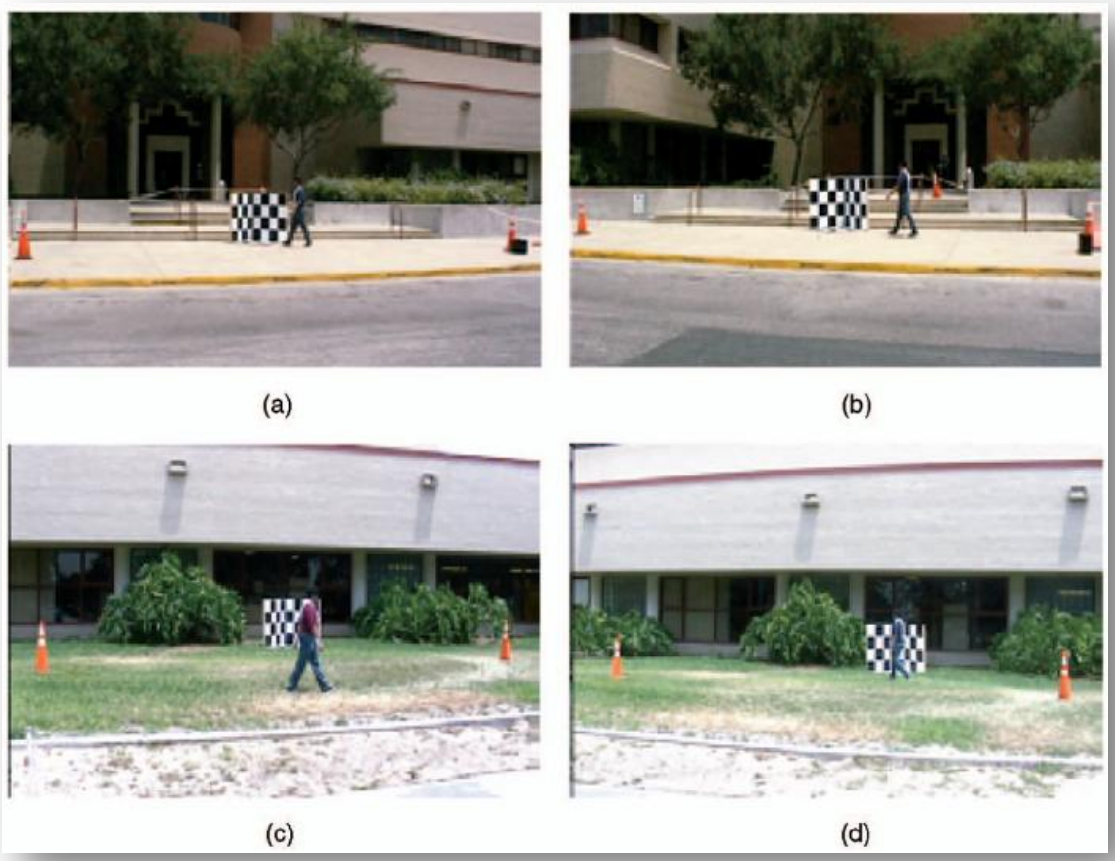


Figure 2-2 Frames from (a) the left camera for concrete surface, (b) the right camera for concrete surface, (c) the left camera for grass surface and (d) the right camera for grass surface

Although data from one full elliptical circuit for each condition is available, the challenge experiments on the data are from the rear portion of the ellipse. The motivations for the elliptical path are

- to challenge the development of algorithms that are robust with respect to variations in the fronto-parallel assumption and
- to provide a data sequence that includes all the views of a person, in order to assist the future development of 3D model-based approaches or 3D visual hull-based approaches.

For such approaches, the calibration object and the two views were imported. Cameras capture 30 frames per second with a shutter speed of 1/250 second and with autofocus left on.
The following metadata was collected on each subject: sex (75 percent male), age (19 to 59 years), height (1.47 m to 1.91 m), weight (43.1 kg to 122.6 kg), foot dominance (mostly right), type of shoes (sneakers, sandals, etc.), and heel height.

Subjects were asked to bring a second pair of shoes so that they could walk the two ellipses a second time in a different pair of shoes. A little over half of the subjects walked in two different shoe types. In addition, subjects were also asked to walk the ellipses carrying a briefcase of known weight (approximately 6 kilograms). Most subjects walked both carrying and not carrying the briefcase. The values of each of the covariates:

- Surface type by G for grass and C for concrete,
- Camera by R for right and L for left,
- Shoe type by A or B,
- NB for not carrying a briefcase and BF for carrying a briefcase, and
- The acquisition time, May and November, simply by M and N.

After processing, video files were stored to computer as one 24-bit, RGB, PPM file per frame of 720x480 resolution. Each subject walking several laps of the course. For the gait data set, frames from the last complete lap were saved, which are from 600 to 700 frames in length. It is important to be marked that although the data set contains frames from one whole lap, the results in this paper are on frames from the back portion of the ellipse. A subject's size in the back portion of the ellipse is on average 100 pixels in height and 25 to 50 pixels in width.

2.3 The Challenge Experiments [14]

The second aspect of the challenge problem is a set of 12 challenge experiments. The 12 experiments are designed to investigate the effect of five factors on performance. The five factors are studied not only individually but also in combinations. The results of the baseline algorithm for the 12 experiments provide a difficulty ordering of the experiments.

In order to allow comparison among a set of experiments and limit the total number of experiments, a gallery was fixed as a control. Then, 12 probe sets were created to examine the effects of different covariates on performance. The gallery consists of sequences with the following covariates: Grass, Shoe Type A, Right Camera, No Briefcase, and collected in May and in November. This set was selected as the gallery because it was one of the largest for a given set of covariates. The last two experiments were focused on studying the impact of time. The time covariate implicitly gave the chance of frames of different shoes and clothes because it was not required from subjects to wear the same clothes or shoes in both data collections.

2.4 The Baseline Algorithm [14]

The third part of HumanID Gait Challenge Problem was the baseline algorithm that concerns future performance improvements. Inspired from the recent success of template-based recognition strategies in computer vision, a four-part algorithm that relies on silhouette template matching was developed. The first part defines semi-automatically bounding boxes around the moving person in each frame of a sequence. The second part extracts silhouettes from the bounding boxes. The third part computes the gait period from the silhouettes. The gait period is used to partition the sequences for spatial-temporal correlation. The fourth part performs spatial-temporal correlation to compute the similarity between two gait sequences. The baseline algorithm does not require the specification of any parameters—it is parameter free.

2.4.1 First Part

The BoundingBox (Figure 2.3) definition in each frame is a semiautomatical procedure. User defines coordinates on each frame that include human. The coordinate values are registered manually in an XML file, which is unique for each probe sample. The XML files may have coordinates of frames in which human doesn't move fronto-parallel, these coordinates are signed as null. The ReadXML function is responsible for reading an XML file in order to rebound coordinates values.

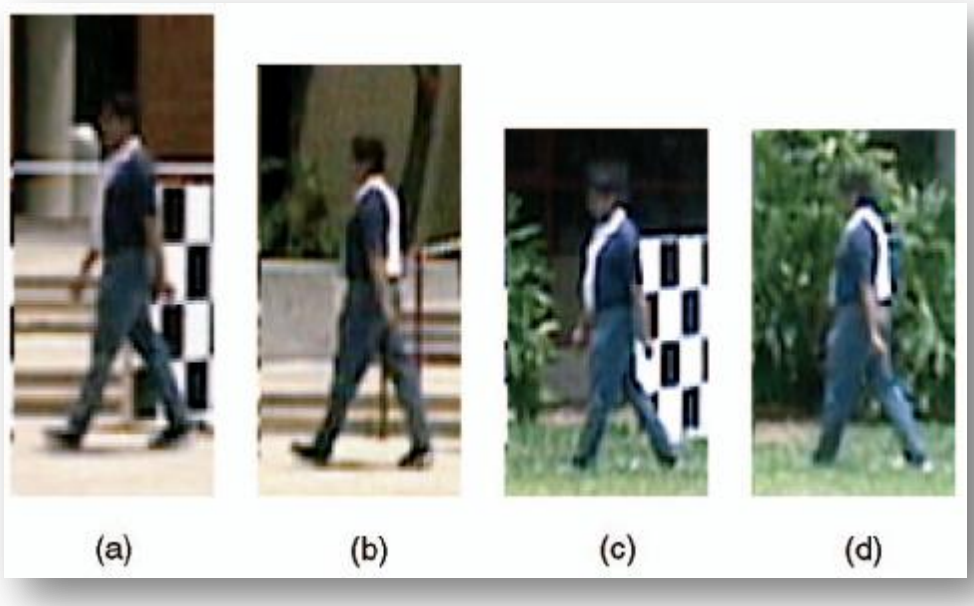


Figure 2-3 Sample Bounding boxed image data as viewed from (a) left camera on concrete, (b) right camera on concrete, (c) left camera on grass and (d) right camera on

2.4.2 Second Part

During the second part, the human silhouette is extracted. As silhouette it is defined the pixels that consist human Figure 2.4. The procedure is presented on the flowchart below and it is explicated further.

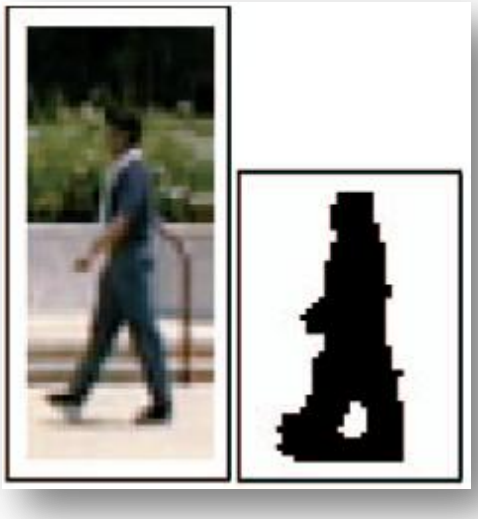


Figure 2-4 Human Silhouette

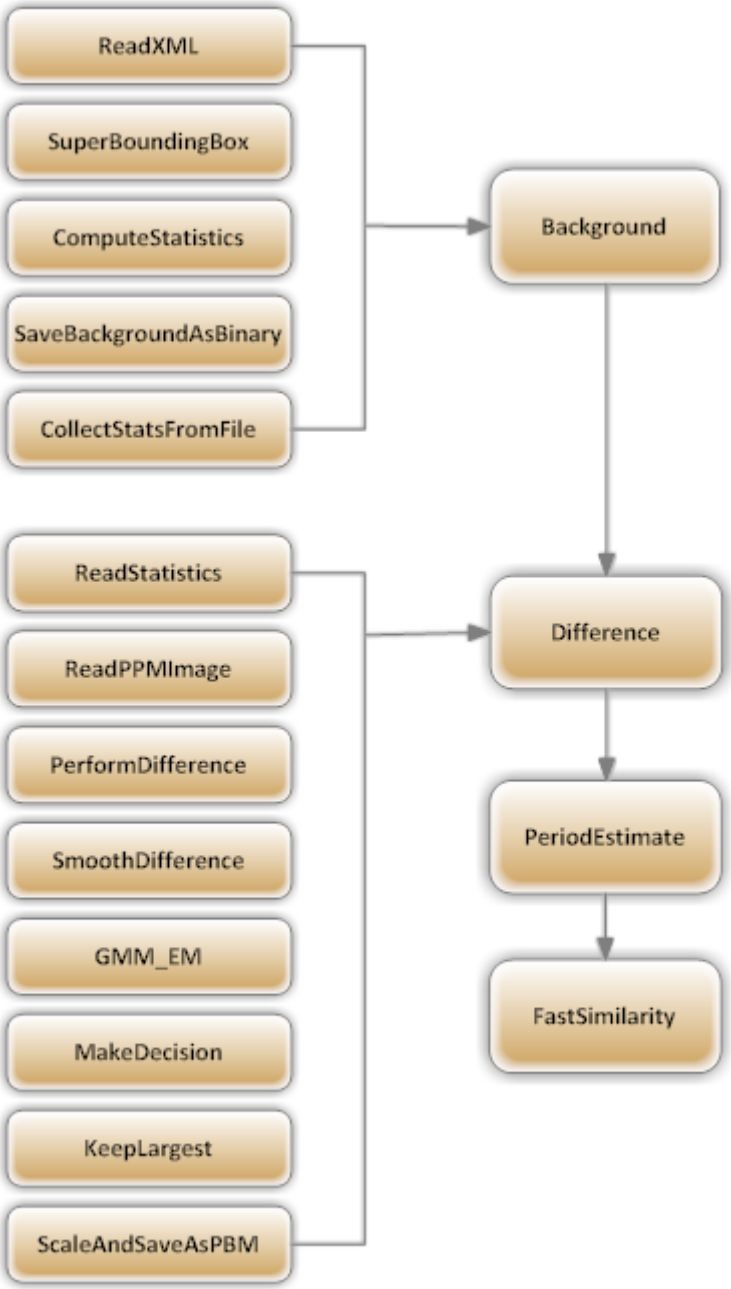


Figure 2-5 Baseline Algorithm Flowchart

As the flowchart presents two executable files are used in order to extract the silhouette. The first executive program (Background) computes the SuperBoundingBox, a pixel area, the coordinates of which are the maximum and the minimum of all not null BoundingBox. Every BoundingBox is included in SuperBoundingBox. The next step is the computation of the background pixels statistics for each frame. As background pixels, for each frame, are defined the pixels that consists the SuperBoundingBox while simultaneously differ from the BoundingBox pixels. This procedure is implemented by the CollectStatsFromFile function which accesses each frame and loads the RGB values to memory. Each pixel

statistics, average and covariance, are computed by calling ComputeStatistics function. After computing, statistics of each pixel are saved to a binary file for future usage, by calling SaveBackgroundAsBinary.

The second executive program (Difference) loads the BoundingBoxes of each frame in memory using ReadXML and the statistics that were computed with Background and were saved as binary file using ReadStatistics. After loading all the previous said data, the next step is the computation oh Silhouette for each frame separately. Firstly, ReadPPMImage function reads the frame and loads the RGB values for each pixel in memory. Then, for each one pixel of BoundingBox, PerformDifference function computes the Mahalanobis Distance for every pixel value R, G, and B using the average which was found with Background. The computation values are saved in a new image in memory. The next stage is smooth of image that was produced previously. It has been found that if the computed Mahalanobis distance array (image) is smoothed by using a 9x9 pyramidal-shaped averaging filter or, equivalently, two passes of a 3x3 averaging filter, the visual quality of the silhouette and the recognition performance improves. Thus, by calling SmoothDifference function the new image is being smooth by a two-pass 3x3 averaging filter. The convergence of the EM process is faster with these smoothed distances than without, possibly due to a reduction in the noise of the computed Mahalanobis distances. The previous procedures have as a result an image, the pixels of which include smooth values of Mahalanobis Distance [22]. According to the pixel value, there are Background or Foreground pixels. The present version decides adaptively on the foreground and background labels for each frame by estimating the foreground and background likelihood distributions using the iterative Expectation Maximization (EM) procedure. At each pixel, indexed by k, we have a two-class problem based on a scalar observation—the Mahalanobis distance, dk. We model the observations as a two-class, {Foreground $= w_1$, Background $= w_2$}, Gaussian Mixture Model (GMM),

$$P(d_k) = \sum_{i=1}^{2} P(w_i) p(d_k | w_i, \mu_i, \sigma_i),$$

Where the class likelihood

$$p(d_k | w_i, \mu_i, \sigma_i) = \frac{1}{\sqrt{2\pi\sigma_i}} e^{-\frac{(d_k - \mu_i)^2}{2\sigma_i^2}}.$$

For each pixel, we would like to estimate the posterior $P(w_1|d_k)$.We iteratively estimate this using the standard EM update equations reproduced below

$$P^{(n+1)}(w_i) = \frac{1}{N} \sum_{k=1}^{N} P^{(n)}(w_i|d_k)$$

$$\mu_i^{(n+1)} = \frac{\left(\sum_{k=1}^{N} P^{(n)}(w_i|d_k) d_k\right)}{\left(\sum_{k=1}^{N} P^{(n)}(w_i|d_k)\right)}$$

$$\sigma_i^{(n+1)} = \frac{\left(\sum_{k=1}^{N} P^{(n)}(w_i|d_k)(d_k - \mu_i)^2\right)}{\left(\sum_{k=1}^{N} P^{(n)}(w_i|d_k)\right)}$$

$$P^{(n+1)}(w_i|d_k) = \frac{\left(p\left(d_k|w_i, \mu_i^{(n)}, \sigma_i^{(n)}\right) P^{(n)}(w_i)\right)}{\left(\sum_{i=1}^{2} p\left(d_k|w_i, \mu_i^{(n)}, \sigma_i^{(n)}\right) P^{(n)}(w_i)\right)}$$

The EM process is initialized by choosing

$$P^{(0)}(w_1|d_k) = \min\left(1.0, d_k/255\right)$$

$$P^{(0)}(w_2|d_k) = 1 - P^{(0)}(w_1|d_k)$$

With this initialization strategy, the process stabilizes fairly quickly, within 15 or so iterations. It is worth mentioning a few words about pre and post processing steps that impact overall performance. Computing each pixel possibility, it is decided using MakeDecision where each pixel belong to, by using a simple comparison between $P(w_1|d_k)$ and $P(w_2|d_k)$. First, isolated, small, noisy regions were eliminated by keeping only the foreground region with the largest area. Second, this foreground region is scaled so that its height is 128 pixels and occupies the whole height of the 128x88 pixel-sized output silhouette frames. The scaling of the silhouette offers some amount of scale invariance and makes the fast computation of a similarity measure easier. The silhouette was also centered along the horizontal direction to compensate for errors in the placement of the bounding boxes. The silhouette is shifted in the horizontal direction so that the center column of the top portion of the silhouette is at column 44. After each frame scaling and shifting, all Silhouette frames are saved in a PBM file.

2.4.3 Third Part

The next step in the baseline algorithm is gait period detection. Gait period,N_{gait} is estimated by a simple strategy. The number of foreground pixels in the silhouette is counted in each frame over time, $Nf(t)$. This number will reach a maximum when the two legs are farthest apart (full stride stance) and drop to a minimum when the legs overlap (heels together stance). To increase the sensitivity, the number of foreground pixels was considered mostly from the legs, which are selected simply by considering only the bottom half of the silhouette. Figure 2.5 shows the number of pixels during gait. The two consecutive strides constitute a gait cycle. The median of the distances is computed between minima, skipping every other minimum. Using this strategy, two estimates of the gait cycle are gotten, depending on whether the first minimum is skipped or not. The gait period is estimated by the average of these two medians. It is important to note that this strategy works for near fronto-parallel views, which is the view of choice for gait recognition and would not work for frontal view
s.

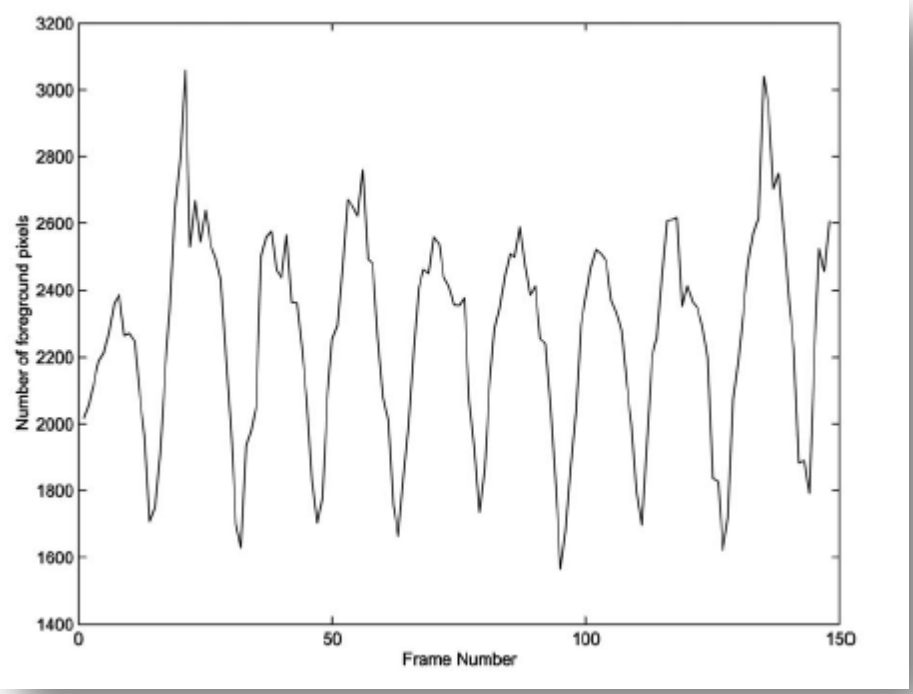


Figure 2-6 the number of foreground pixels during gait

2.4.4 Forth Part

The output from the gait recognition algorithm is a complete set of similarity scores between all gallery and probe gait sequences. Similarity scores are computed by spatial-temporal correlation. As a probe sequence of M frames can be denoted by $S_P = \{S_P(1), \dots, S_P(M)\}$ and as a gallery sequence of N frames can be denoted by $S_G = \{S_G(1), \dots, S_G(M)\}$. The final similarity score is constructed out of matches of disjoint portions of the probe with the gallery sequence. Specifically, the probe sequence is partitioned into disjoint subsequences of N_{gait} contiguous frames, where N_{gait} is the estimated period of the probe sequence from the previous step. It is remarkable the fact that the starting frame of each partition is not constrained to be at a particular stance. As a k^{th} probe subsequence can be denoted by $S_{Pk} = \{S_P(kN_{gait}), \dots, S_P((k+1)N_{gait})\}$. The gallery gait sequence consists of all silhouettes extracted in the gallery sequence from the back portion of the elliptical path. It is significant to note that this gallery sequence is not partitioned.

Each of the subsequences S_{Pk} is correlated with the entire gallery sequence S_G. There are three ingredients to the correlation computations: frame correlation, correlation between S_{Pk} and S_G, and similarity between a probe sequence and a gallery sequence, comparing S_P and S_G.

At the core of the above computation is the need to compute the similarity between two silhouette frames, $FramcSim(S_P(i), S_G(j))$, which is defined as the ratio of the number of pixels in their intersection to their union. This measure is also called the Tanimoto similarity measure [21], defined between two binary feature vectors. Thus, if the number of foreground pixels can be denoted in silhouette by Num(S), then we have,

$$FramcSim(S_P(i), S_G(j)) = \frac{\mathrm{Num}(\mathrm{S_P(i)} \cap \mathrm{S_G(j)})}{\mathrm{Num}(\mathrm{S_P(i)} \cup \mathrm{S_G(j)})}.$$

It is important to be marked the fact that since the silhouettes have been prescaled and centered, it is not necessary to consider all possible translations and scales during the computing of frame-to-frame similarity. The next step is to use frame similarities to compute the correlation between S_{Pk} and S_G

$$Corr(S_{Pk}, S_G)(l) = \sum_{j=0}^{N_{gait}-1} FramcSim(S_P(k+j), S_G(l+j))$$

The median value of the maximum correlation of the gallery sequence with each of these probe subsequences is chosen as more suitable measure of similarity. Other choices such as the average, minimum, or maximum did not result in better performance.

$$Sim(S_P, S_G) = Median_k(max_l Corr(S_{Pk}, S_G)(l))$$

3 Implementation

3.1 Code Profile

Understanding of the computing intensive functions is the most important part of this thesis. In order to achieve our goal, we used Intel Vtune Performance Analyzer 9.0 [23] and the clock() C++ function. The results of Vtune Analyzer led us to the conclusion that the most computing intensive executable files are Fast Similarity and Difference, especially its GMM_EM function. Clock function measures simultaneously the total execution time and the clock cycles approximately. However, the Vtune Analyzer results of clock cycles were far different from the approximate measures. After search, we found out that the hard disk read and write calls were responsible for this difference, as Vtune Analyzer does not measure the clock cycles which are necessary for the hard disk access.

In order to understand and utilize all Cell BE capabilities, it was decided to be made some code modifications. The main code modification was the elimination of hard disk access. To achieve this, we had to join the five executable files to a unified project. This code development had as a result, a single executable file that implements the Baseline Algorithm. We actually keep the results of each part of the algorithm into memory instead of using hard disk. The rest code modifications concern the elimination of some useless loop branches and more efficient memory management.

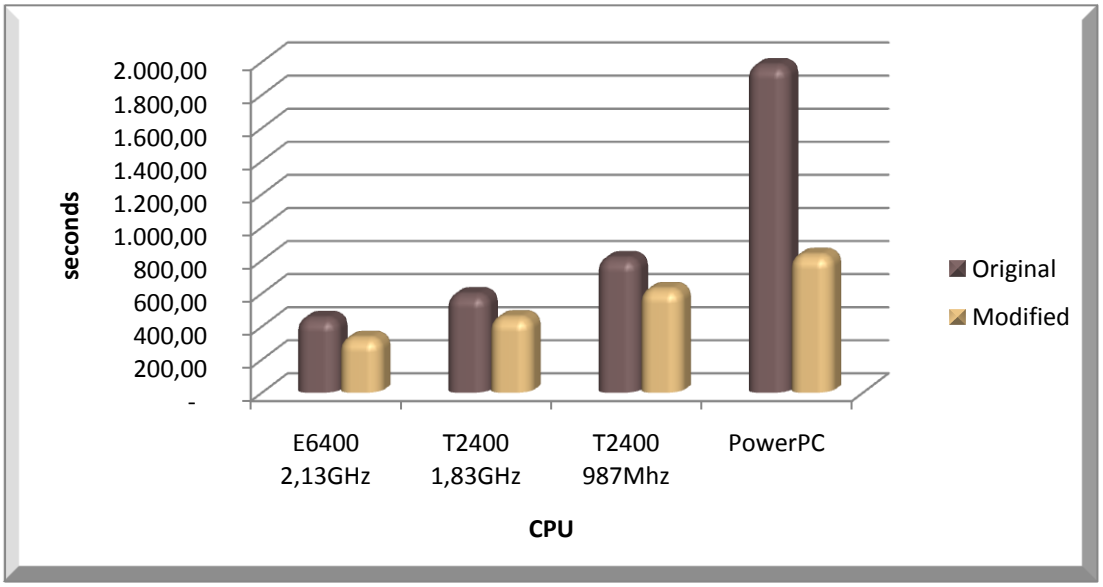


Figure 3-1 Code improvement

seconds	E6400 2.13 GHz	T2400 1.83 GHz	T2400 987 MHz	PowerPC
Original	448.57	592.46	810.34	1977.19
Modified	328.19	452.74	623.29	830.01

Table 3-1 Code improvement

After code improvement and the extract of the hard disk read write calls the new executable file demands less execution time. Insomuch that, we studied Cell BE capabilities, using the modified code. The diagram below presents a percentage depiction of each function execution time. The algorithm executed in three x86 processors and in Cell BE using only the PowerPC Processing Element (PPE).

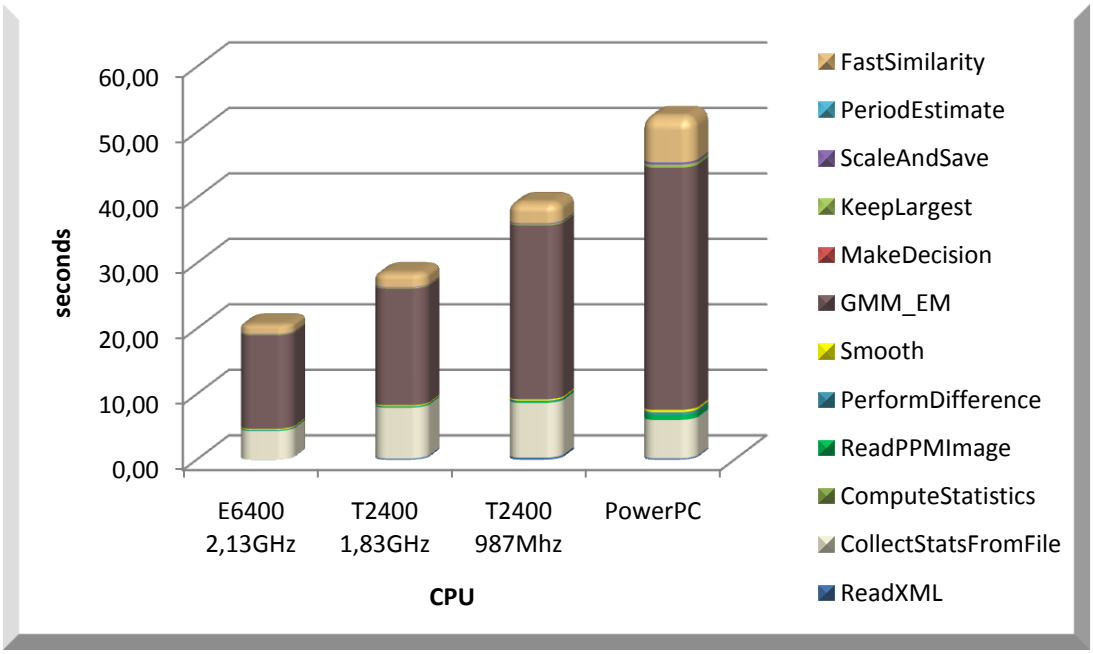


Figure 3-2 The proportion of the functions in the overall execution time of the code.

seconds	CPU			
	E6400 2,13GHz	T2400 1,83 GHz	T2400 987 MHz	PowerPC
ReadXML	0.016250	0.180000	0.298375	0.191826
CollectStatsFromFile	4.366375	7.759250	8.370750	5.871366
ComputeStatistics	0.012375	0.018750	0.031625	0.019416
ReadPPMImage	0.096250	0.115500	0.165875	0.756059
PerformDifference	0.091250	0.100190	0.149625	0.419161
Smooth	0.123250	0.137875	0.197000	0.323276
GMM_EM	14.307750	17.817125	26.561000	36.990616
MakeDecision	0.009000	0.008500	0.011625	0.034304
KeepLargest	0.093125	0.130375	0.171875	0.453052
ScaleAndSave	0.089500	0.103125	0.159125	0.339422
PeriodEstimate	0.003375	0.011750	0.016875	0.014444
FastSimilarity	1.425000	2.108571	3.098714	7.020544

Table 3-2 The execution time of the functions

From the diagram above, we conclude that three most computing intensive functions are GMM_EM, FastSimilarity and CollectStatsFromFile. The improvement of the first two will be presented in next sections. While, the third function, CollectStatsFromFile is not easily improvable. This function read the frames and its performance is based on the hard disk capabilities. The first program version used to read the whole frame. We improved this by constrain reading to the necessary frame section only.

3.2 Programming Models [26]

There are several types of programming models that use the Cell BE Synergistic Processor Elements (SPEs). The most important types of these are:

- Function-Offload Model,
- Device-Extension Model,
- Computation-Acceleration Model,
- Streaming Model,
- Shared-Memory Multiprocessor Model,
- Asymmetric-Thread Runtime Model,
- User-Mode Thread Model.

Function-Offload Model

In the Function-Offload Model, the SPEs are used as accelerators for performance-critical procedures. This model is the quickest way to effectively use the Cell BE 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. The Function-Offload Model is sometimes called the Remote Procedure Call (RPC) Model. The model allows a PPE program to call a procedure located on an SPE as if it were calling a local procedure on the PPE. This provides an easy way for programmers to use the asynchronous parallelism of the SPEs without having to understand the low-level workings of the MFC DMA. In this model, we identify which procedures should execute on the PPE and which should execute on the SPEs. The PPE and SPE source modules must be compiled separately, by different compilers.

Device-Extension Model

The Device Extension Model is a special case of the Function-Offload Model in which the SPEs act like I/O devices. SPEs can also act as intelligent front ends to an I/O device. Mailboxes can be used as command and response FIFOs between the PPE and SPEs. The SPEs can interact with I/O devices because:

- all I/O devices are memory-mapped, and
- The SPEs DMA transfers support transfer sizes of a single byte.

I/O devices can use an SPEs signal-notification facility to tell the SPE when commands complete. When SPEs are used in the Device-Extension Model, they usually run privileged software that is part of the operating system. As such, this code is trusted and may be given access to privileged registers for a physical device. For example, a secure file system may be treated as a device. The operating system's device driver can be written to use the SPE for encryption and decryption and for responding to disk-controller requests on all file reads and writes to this virtual device.

Computation-Acceleration Model

The Computation-Acceleration Model is an SPE-centric model that provides a smaller-grained and more integrated use of SPEs. The model speeds up applications that use computation-intensive mathematical functions without requiring significant rewrite of the applications. Most computation intensive sections of the application run on SPEs. The PPE acts as a control and system-service facility. Multiple SPEs work

in parallel. The work is partitioned manually by the programmer, or automatically by the compilers. The SPEs must efficiently schedule MFC DMA commands that move instructions and data. This model either uses shared memory to communicate among SPEs, or it uses a message-passing model.

Streaming model

In the Streaming Model, each SPE, in either a serial or parallel pipeline, computes data that streams through. The PPE acts as a stream controller, and the SPEs act as stream-data processors. For the SPEs, on-chip load and store bandwidth exceeds off-chip DMA-transfer bandwidth by an order of magnitude. If each SPE has an equivalent amount of work, this model can be an efficient way to use the Cell Broadband Engine because data remains inside the Cell Broadband Engine as long as possible.

Shared-Memory Multiprocessor Model

The Cell BE can be programmed as a shared-memory multiprocessor, using two different instruction sets. The SPEs and the PPE fully interoperate in a cache-coherent Shared-Memory Multiprocessor Model. All DMA operations in the SPEs are cache-coherent. Shared-memory load instructions are replaced by DMA operations from shared memory to local store (LS), followed by a load from LS to the register file. The DMA operations use an effective address that is common to the PPE and all the SPEs. Shared-memory store instructions are replaced by a store from the register file to the LS, followed by a DMA operation from LS to shared memory. The SPEs DMA lock-line commands provide the equivalent of the PowerPC Architecture atomic-update primitives (load with reservation and store conditional). A compiler or interpreter could manage part of the LS as a local cache for instructions and data obtained from shared memory.

Asymmetric-Thread Runtime Model

Threads can be scheduled to run on either the PPE or on the SPEs, and threads interact with one another in the same way they do in a conventional symmetric multiprocessor. The Asymmetric-Thread Runtime Model extends thread task models and lightweight task models to include the different instruction sets supported by the PPE and SPE. Scheduling policies are applied to the PPE and SPE threads to optimize performance. Although preemptive task-switching is supported on SPEs for debugging purposes, there is a runtime performance and resource-allocation cost. FIFO run-to-completion models, or lightweight cooperatively-yielding models, can be used for efficient task-scheduling. A single SPE can run only one thread at a time; it cannot support multiple simultaneous threads. The Asymmetric-Thread Runtime Model is flexible and supports all of the other programming models described in this chapter. Any program that explicitly calls spe_context_create and spe_context_run is an example of the Asymmetric-Thread Runtime Model.

User-mode thread model

The User-Mode Thread Model refers to one SPE thread managing a set of user-level functions running in parallel. The user-level functions are called microthreads (and also user threads and user-level tasks) . The SPE thread is supported by the operating system. The microthreads are created and supported by user software; the operating system is not involved. However, the set of microthreads can run across a set of SPUs. The SPU application schedules tasks in shared memory, and the tasks are processed by available SPUs. For example, in game programming, the tasks can refer to scene objects that need updating. Microthreads can complete at any time, and new microthreads can be spawned at any time. One advantage of this programming model is that the microthreads, running on a set of SPUs under the control of an SPE thread, have predictable overhead. A single SPE cannot save and restore the MFC commands queues without assistance from the PPE.

In this thesis, for the Baseline Algorithm we choose the Function-Offload Model. This choice was selected because there is already an implementation. With this model we simply replace the computing intensive functions with SPEs calls. This model does not change the sense of algorithm, it just improve the performance with parallelism. We developed the GMM_EM and FastSimilarity functions by this model. It was the best and the fastest way to improve the performance.

3.3 Development Model

This section presents the development model which was followed in order to develop the SPEs code so as to be joined to PPE program.

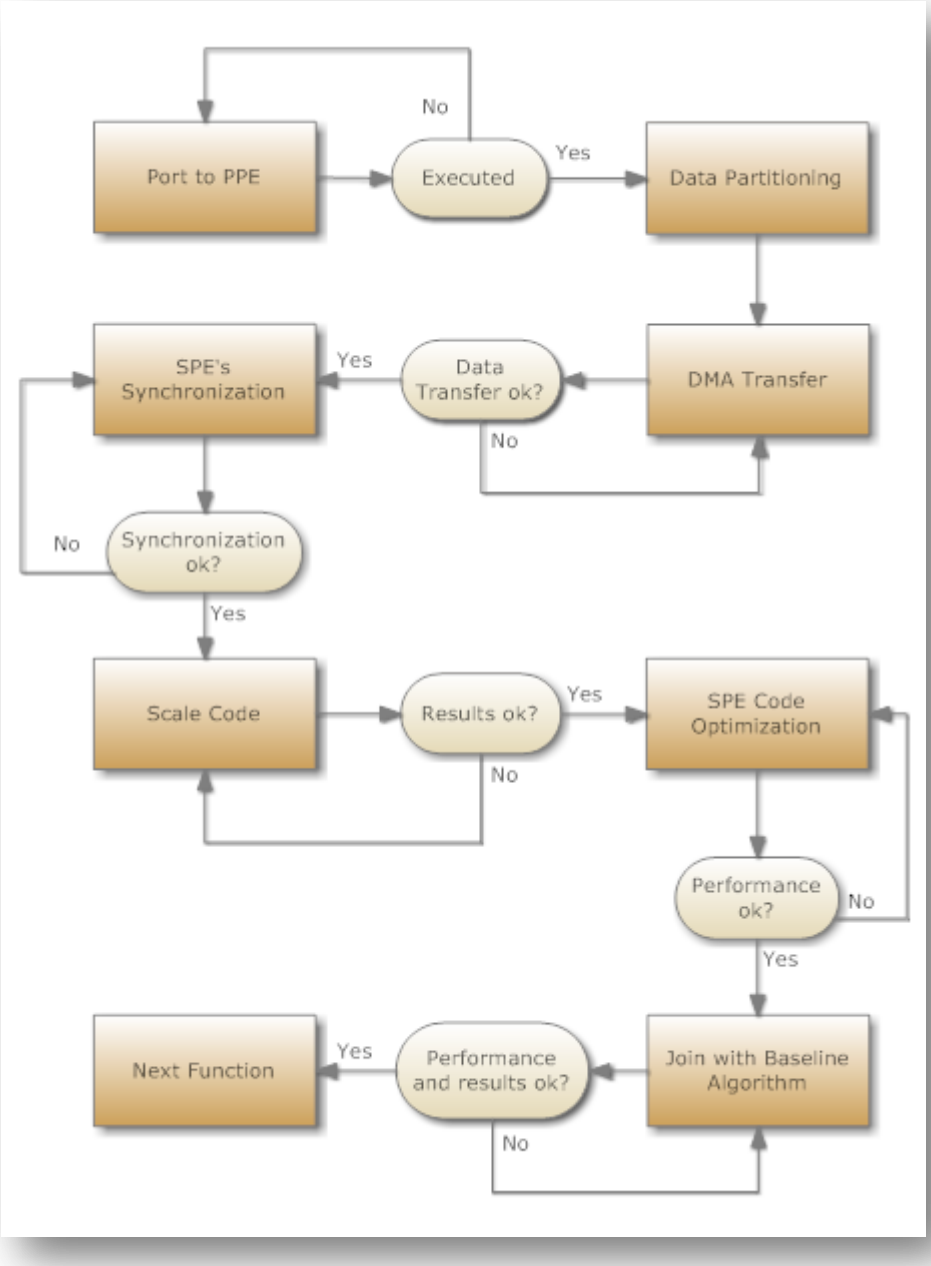


Figure 3-3 Development Model Flow Chart

Port to PPE

A C++ program is easily executable by the PPE. The only requirement is a compiler change into the makefile and the addition of some necessary libraries.

Data Partitioning

The second part which was important to be studied by using Function-Offload model was the appropriate data partitioning. The code conversion to SIMD demands careful data partitioning. As we know the number of the available SPEs and the data amount that is going to be transferred, the simplest way of data partitioning is the division of data amount with the number of SPEs. It is also essential, the data amount that would be transferred to each SPE must be multiple of the data that could exist in a SIMD instruction. Insomuch that, the data amount is usually a little bit larger than the quotient of the total amount divide by the number of SPEs. Thus, we transferred to the last SPE some useless data that are skipped during program execution.

DMA Transfer

The transfer way was also studied as there are constrains. All data was memory 16-byte aligned as it is required by MFC DMA transfers [27]. The data were transferred directly, if they were small enough, and they were used repeatedly. Or they were transferred by a double buffer, if they were numerous.

DMA transfers are limited to 16 KB data amount per instruction and the size can only be 1, 2, 4, 8, 16 or multiple of 16 bytes. With the double buffer method we achieve the better performance for the data transfer, as it can be executed simultaneously with the rest code execution. By this way, we transfer the first data and during their process we transfer the next. The same way is used for their store back in shared memory. There are two ways of MFC DMA transfers. Either the PPE initiates transfers data or the SPEs initiate. The second way was selected as it is faster and leaves the PPE available for other functions.

SPEs Synchronization

Sometimes, the results of each SPE must be used from other SPEs. For this reason, we store the SPEs results in the main memory, whereas each SPE notifies the others, with signals [27], that processing and storing have been completed. In order to establish a way of communication among SPEs, the PPE has to map SPEs signals area (effective address) to main storage and then send each SPE the effective address of the others ESP's signals area. These signals are actually a variation of DMA transfers.

Scale Code

After we completed all the previous mentioned stages, the simplest part of implementation is the code of the process that we have to develop to the SPEs. This code is the same with the one that was existed to PPE. Sometimes, some changes must be done if there are PPE process instructions with different name from the SPE one. Finally, we check if the results and the performance improvement are prospective. The ideal improvement is achieved when execution time is as many times faster as the SPEs number.

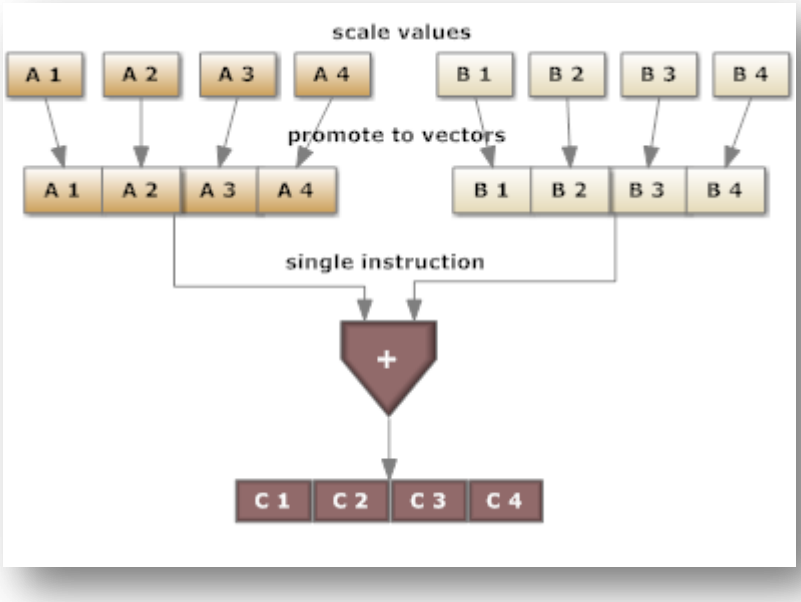
SPE Code optimization

For the improvement of the code which was developed till now for the SPEs, we use 3 methods: Function Inline, SIMD, Loop Unrolling.

An inline function is one for which the compiler copies the code from the function definition directly into the code of the calling function rather than creating a separate set of instructions in memory. Instead of transferring control to and from the function code segment, a modified copy of the function body may be substituted directly for the function call. These include the branch and set link for function-call entry, and the branch indirect for function-call return.

Generally, the branch avoidance at SPEs is an appropriate method, as the branch prediction is computing intensive for the SPEs. In order to achieve this, we use the flag-Winline into makefile so as to make the compiler to develop code with function inlining.

We can execute the same instruction for more data, using the Single Instruction Multiple Data (SIMD) [24] [25]. A general example is shown in the figure below.



If during the data partitioning the amount of data that is transferred is multiple of the data that are used in SIMD instruction, the code conversion is typical. The conversion from scale to SIMD code requires attention especially as far as it concerns loop branches decrement. If the code results are the desirable, the conversion was succeeded. Then, we studied if the required execution time is the

ideal. In this case, the ideal time is submultiple of the initial time as many times as amount data of the SIMD instruction multiplied with the SPEs number.

We can reduce the number of loop branches by using the Loop Unrolling function. As SPEs are computing intensive as far as it concerns branch prediction, we can discriminate execution time significantly.

Join with Baseline Algorithm

We adjust SPEs code to the initial one. As the program is executed by the SPEs, the PPE is inactive and waits the execution to be completed, during this time, we can make PPE to execute another code part until SPEs complete their execution. By this way we achieve even wider parallelization. Finally, we check the time and total program results. If they are not the desirable we move on probable improvements.

3.4 GMM_EM Development

3.4.1 Port to PPE and Analysis

The GMM_EM function requires a large enough data amount. Also according to Vtune Analyzer, there are eight commands that require several clock cycles to execute. As first thought we had considered to transfer data for such instructions and their execution. But to complete the process, those instructions should be executed thousands of times as they are in loop iteration. But the call of the SPEs in each iteration is not beneficial. Calling the SPEs requires a time which is quite large compared to the time of execution. Also it is large enough and the time of data transfer. To begin with, we implemented a model function in PPE and tried to transfer it in SPE.

3.4.2 Data Flow Analysis and Data Partitioning

We are going to estimate the data amount that is required by GMM_EM so as to be executed. Then, we are going to search how we can decrease the data amount. The function has some thousands double precision numbers, as entry, which consist the R level of each pixel. The number of pixel per frame is different and proportional with the BoundingBox of each frame. As this number is not constant, we compute the minimum and maximum number of pixels that are included in the data frames, which we are going to process and the average as well.

double precision	Data Flow		
	Minimum	Maximum	Average
Pixel/Frame	5.390	11.470	8.063
Data Size (KB)	42,10	89,61	62,99
Pixel/Sample	1.078.000	2.294.000	1.612.600
Data Size (MB)/Sample	8,22	17,50	12,30

Table 3-3 GMM_EM Data Flow Analysis for double precision

Having the R value of each pixel we have to compute the observed distance and the initialization of posterior probability. Then, we compute the average, mean and covariance of posterior probabilities. Once we have all the necessary variables, we continue to the calculation of posterior probabilities. If we want to achieve only the calculation of posterior probabilities in the SPEs, the amount of data that must be transferred is at least three times the amount of R values of pixels. Since the SPEs are quite fast in their calculations, it is preferable to load the R of each pixel and continue from there the calculations at the SPEs. But data partitioning is quite

complicated. Insomuch that, it is preferable to calculate the observed distance values for each pixel in the PPE and then transfer to the SPEs to continue his calculation.

The amount of data is such that allows us to keep all values for each frame in the local memory (LS). In order to have enough local memory (LS) for each frame, it must be available at least three SPEs. Having at our disposal at least three SPEs the Data Partitioning is quite easy and fast. Thus each SPE will load the data, the amount of which will be submultiple of the SPEs number on a SIMD command data. The data that may have an instruction SIMD double precision is two. However, due to the fact that data implementation is possible, even if just for experimentation, in the form of single precision, we consider that data in a SIMD is four.

single precision	Data Flow		
	Minimum	Maximum	Average
Pixel/Frame	5.390	11.470	8.063
Data Size (KB)	21,05	44,81	31,50
Pixel/Sample	1.078.000	2.294.000	1.612.600
Data Size (MB)/Sample	4,11	8,75	6,15

Table 3-4 GMM_EM Data Flow Analysis for single precision

3.4.3 DMA Transfer

Having calculated the Data Partitioning, we must calculate the way of data transferring. We will call the DMA transfer command from the SPEs, not only because the Data Partitioning allows us to, but also because they are faster. Also, as it was decided earlier, all calculations of the function will take place in the SPEs. In this way we face a major problem. The problem is how the SPEs will have all the addresses that are located the data for each Frame.

The problem was solved by giving the data in a fixed amount array. Because we know the maximum amount of data we can use this array. Of course it requires time for the initialization and zeroing the unused elements of array. This term is quite small compared to the gain time. So we load to the SPEs the address of the fixed array.

The next step is to choose whether to load all the data directly or if we use the method of double buffer. Because we need all the data to calculate the average at the beginning and it is not very computing intensive, necessarily we load all data directly and not with the double buffer method.

3.4.4 Synchronization

The way with which we decided to implement the process creates a problem. How the SPEs would know that data, which they have to load, have been initialized and that they must start their execution. To solve this problem we use mailboxes. Each SPE can receive and send mail of 32bit size to and from the PPE. Once we have calculated the required data by the SPEs with the above array, the PPE sends a mail to each SPE. The SPEs in order to start the process they wait until the mail received. Once they complete the process, they send mail to PPE for updating and continuing the program.

SPEs should have all the data to calculate the mean, average and covariance. But due to the Data Partitioning the data are divided in SPEs. Thus, we calculate the sum of the data in each SPE and the result was stored in the main memory. One of the SPEs loads the results and calculates the final values. Finally, they are stored back into main memory from where they can be uploaded by the SPEs.

Upon completion of the addition, the SPEs send a signal to original SPE that they have completed this process and saved the value in main memory. The first SPE loads these values and calculates the average and covariance. The data amount is very low, 96 - 256 byte. At the end of the calculation the first SPE communicate with other SPEs and in the same way as before the other SPEs take the values. It was preferred the values to be stored in main memory, because the transfer between local memory was not very stable and requires a complex synchronization. The time difference between the two methods is negligible compared to the total optimization.

3.4.5 Scale Code

The GMM_EM consists of three parts. The first two compute the average and the covariance of data. The third part of GMM_EM calculates new values for our data without any dependence. So there is no communication between the SPEs. Theoretically the results should be the same and we would continue the optimization of SPEs. But the results were not the same, they were varied after the tenth digit of decimal point. During the SPEs using, we had to change the order of addition. Due to the fact that the floating point numbers addition is not associative, it was displayed a round-off error from the tenth digit after the decimal point.

At this point we wanted to check if there is deterioration in the performance of the algorithm due to this difference. For each pixel, we count two possibilities, whether it is foreground or background. In following function, the two probabilities were compared, and whichever is longer and takes the final price. So compare the pixels values of the new and the old implementation. The comparison was carried out at double precision and single precision and if precision was not important, we could accelerate even more the function by using single precision numbers.

In our Samples (3204 Frames)	Different Pixels		
	Minimum	Maximum	Average
Pixel/Frame	0	69	0,81

Table 3-5 Different pixels due to order of addition

In our Samples (3204 Frames)	Modified		
	No	Yes	Modified percent
Frames	3082	122	3,96%

Table 3-6 Different frames due to order of addition

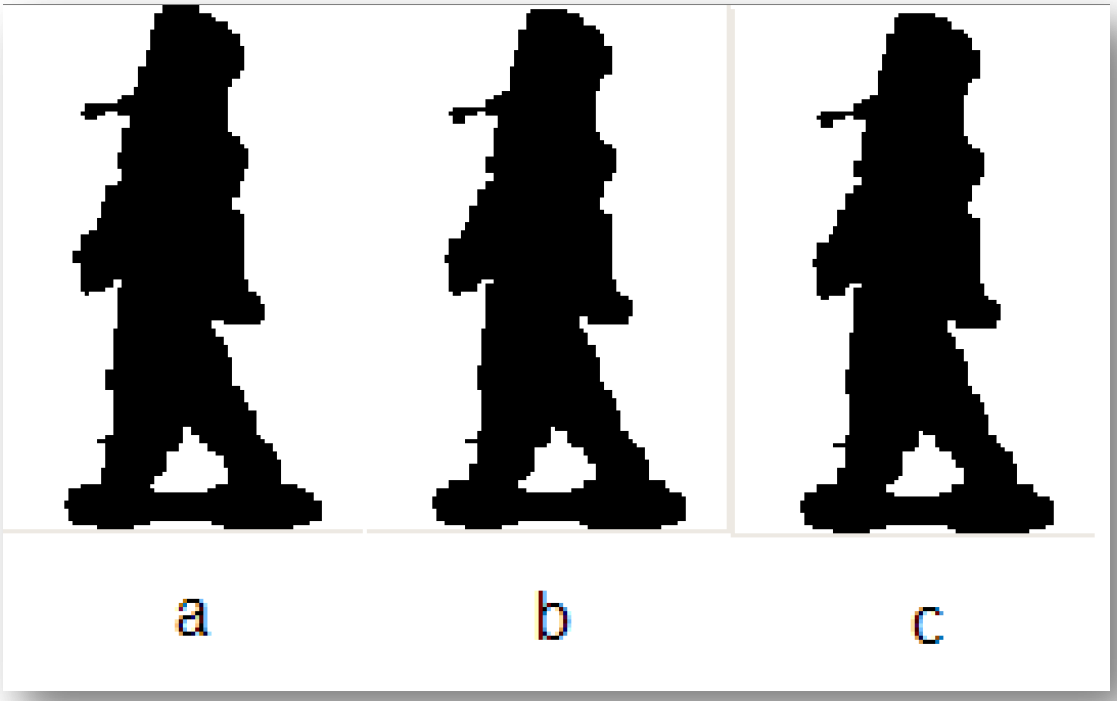


Figure 3-5 a) Original silhouette frame b) silhouette frame with double precision and addition error c) silhouette frame with single precision and addition error

From the results, we observe that the differences are few and most pixels have no difference either in implementation of double precision or in single precision. Observing visually the differences, we can see that the varying pixels are located borderline of the silhouettes which means that probably is not part of it. The results of the algorithm change in such level that do not impact its performance.

3.4.6 SPE Optimization

In order to improve the speed of SPEs we used the function inline method. The difference was not evident as there were not many improvements that could be made.

Also we used SIMD instructions. The data transferred to the SPEs, they are placed in vectors and the amount is multiple of the data vectors either they are double precision or they are single precision. In the original process data was in double precision form. At this point we had a problem with the implementation speed of double precision vector SIMD instructions. Double precision operations have latency 13 clock cycles. The first six cycles are actually stalls in which no other instruction can be issued. The SPU is not only able to process multiple values at once through its pipeline, it can also dual-issue instructions through different pipelines. The SPU has two pipelines, even (sometimes called pipeline 0 or the execute pipeline) and odd (sometimes called pipeline 1 or the load pipeline). While SPU execute a double precision operation, Dual-issue is not allowed with these instructions either. [32]

That is why we tested the choice of the data being in single precision. As we saw earlier the change in precision is not important. By using single precision form instead of double precision, we dramatically increase the speed and the results were consistent with the main theory of the processor. Also, the processor is made so as to perform known mathematical operations SIMD, such as square root, even faster.

Finally, we did not see any particular increase in the implementation speed of the program by the method of Loop Unrolling, as there were not many miss predicted branches and the loop branches were few. Particularly, we develop some loops 25 times and some other completely.

3.4.7 Join Code

Although the final result is much faster than the original, the time of the SPEs data process is not negligible. Once the SPEs process data, the PPE remains inactive. But, there is the possibility to continue with the program execution. In order to improve even further the performance speed we used the inactive period to run another process. Specifically, using the buffer logic, we execute a process that initializes data for the SPEs.

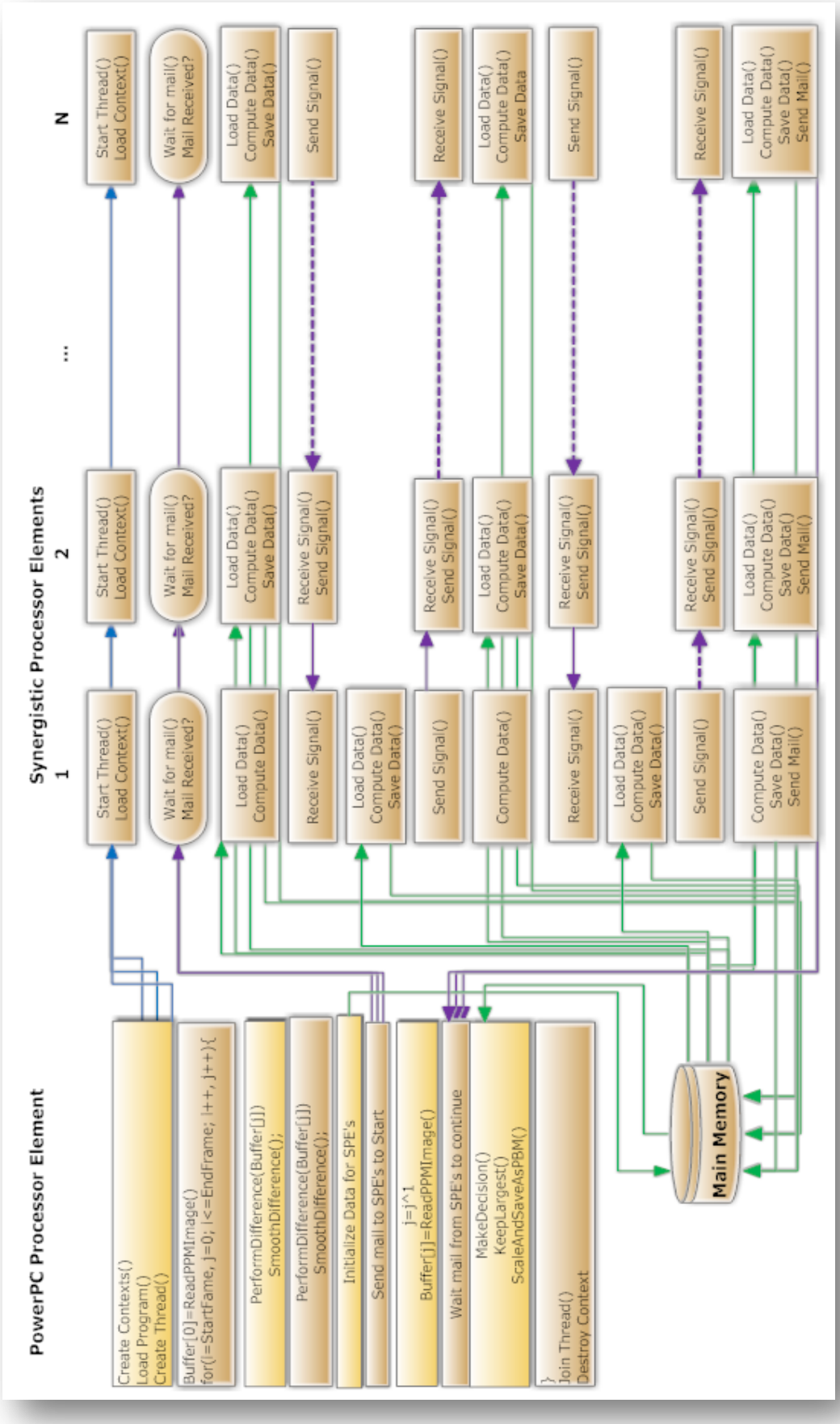


Figure 3-6 GMM_EM Flow Chart

3.5 Fast Similarity Development

3.5.1 Port to PPE and Analysis

The second and quite computing intensive process is the comparison of the probe samples experiments with the samples of the Gallery. And more specifically, during the finding of correlations between the sequences of frames sample and the Gallery sample. In comparison there is no dependence between the data and for this reason it could easily be parallelized. This part was from the beginning, a separate executable file. So we worked on it as a model to convert it from the PPE code to SPE.

3.5.2 Data Flow Analysis and Data Partitioning

The frames of the sample are divided into sequences and each sequence frame is compared with the entire Gallery sample frames. Separation of sequences in Frames is based on the period of human gait. The sequences of these may be from 4 to 7. So each core loads one sequence. In case that the sequences are more than the available cores, the latter is loaded on the first SPE. Even if this option is not the most appropriate in terms of complexity, it was accepted because the difference in the duration of this mode and the same data amount sharing is not important. Also in this way we can have an easily executable algorithm.

204 frames/sample approximately	Data Flow			
	Probe Sample	Gallery Sample	Sum	Per SPE
Frames	204	204	408	238
KB	281	281	562	327

Table 3-7 Fast Similarity Data Flow Analysis

Each sample has not a fixed number of frames. The samples have an average of 204 frames. Each frame size is 1,408 Byte, forming an image of 11,264 bits or 128x88 pixels. So any comparison of samples, it has to be loaded about 562 KByte total. Each SPE loads approximately about 34 frames that represent a sequence of 204 probe sample frames and 204 frames of a Gallery sample, a total of about 327 KByte.

3.5.3 DMA Transfers

According to the Data Partitioning, we notice that all data that must be loaded on a SPE are not fit to be stored in the Local Store. To reduce the transfer time we use the method of buffer transfer and we transfer the Gallery of Frames per pieces. Since the sequences are fragmented into period frames, we load the frames and a period from samples of the Gallery. During the correlation, we transfer the next package. Thus, we have every time approximately 68 frames loaded in the Local Store, about 93,5 KByte.

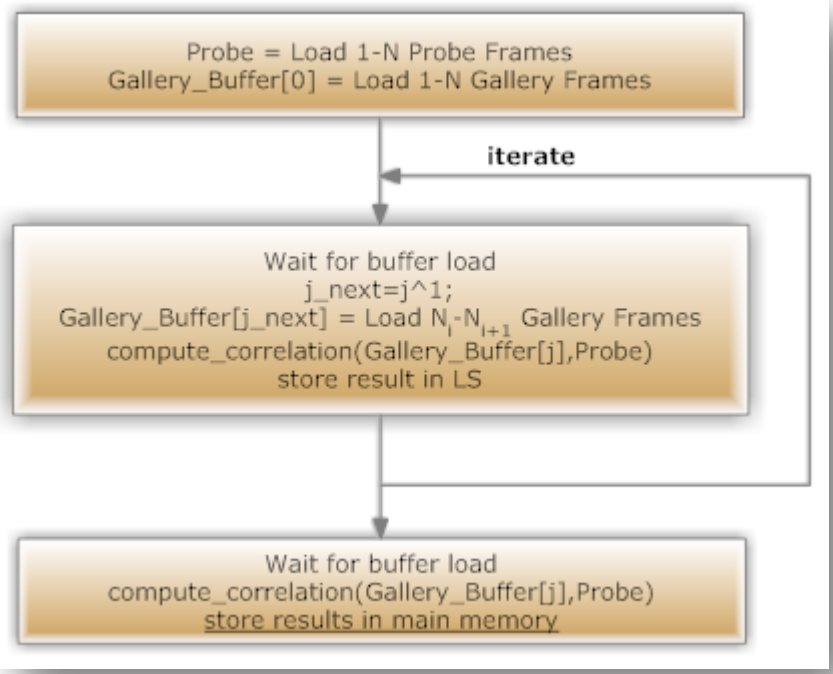


Figure 3-7 DMA Transfer with double buffer method

3.5.4 Scale Code

The SPEs running a process with data that are not dependent on anything else. So no need for synchronization between the SPEs. By moving the code to scale form and by running we marked that we get the desired results, so we move to optimization.

3.5.5 SPE Code Optimization

We used through the makefile the method Function Inline, but due to the limited code we did not expect significant improvements. It was also developed a full loop unrolling in an important section of code and in another 22 times, but we had no significant improvement.

Finally we used SIMD instructions. The data was already correctly distributed and the 1,408 byte is a multiple of a 16 byte character SIMD instruction. Because the data is in the form of character (1 byte) we can achieve 16 times faster results. The SIMD instructions are limited in terms with character data, we encountered a problem and some instructions do not exist. Especially, after the comparison of the entire vector elements with zero we could not have resulted how many of them were zero and how many they are not. So we used a different instruction that if an element is other than zero then it converts all bits to one (255 in decimal). After adding all the elements and dividing by 255, we could see how many of them are not zero.

3.5.6 Join Code

The association with the PPE code was very simple and had no specificity. As this is the last piece of code was not something that can be executed in the PPE during the performance of SPEs.

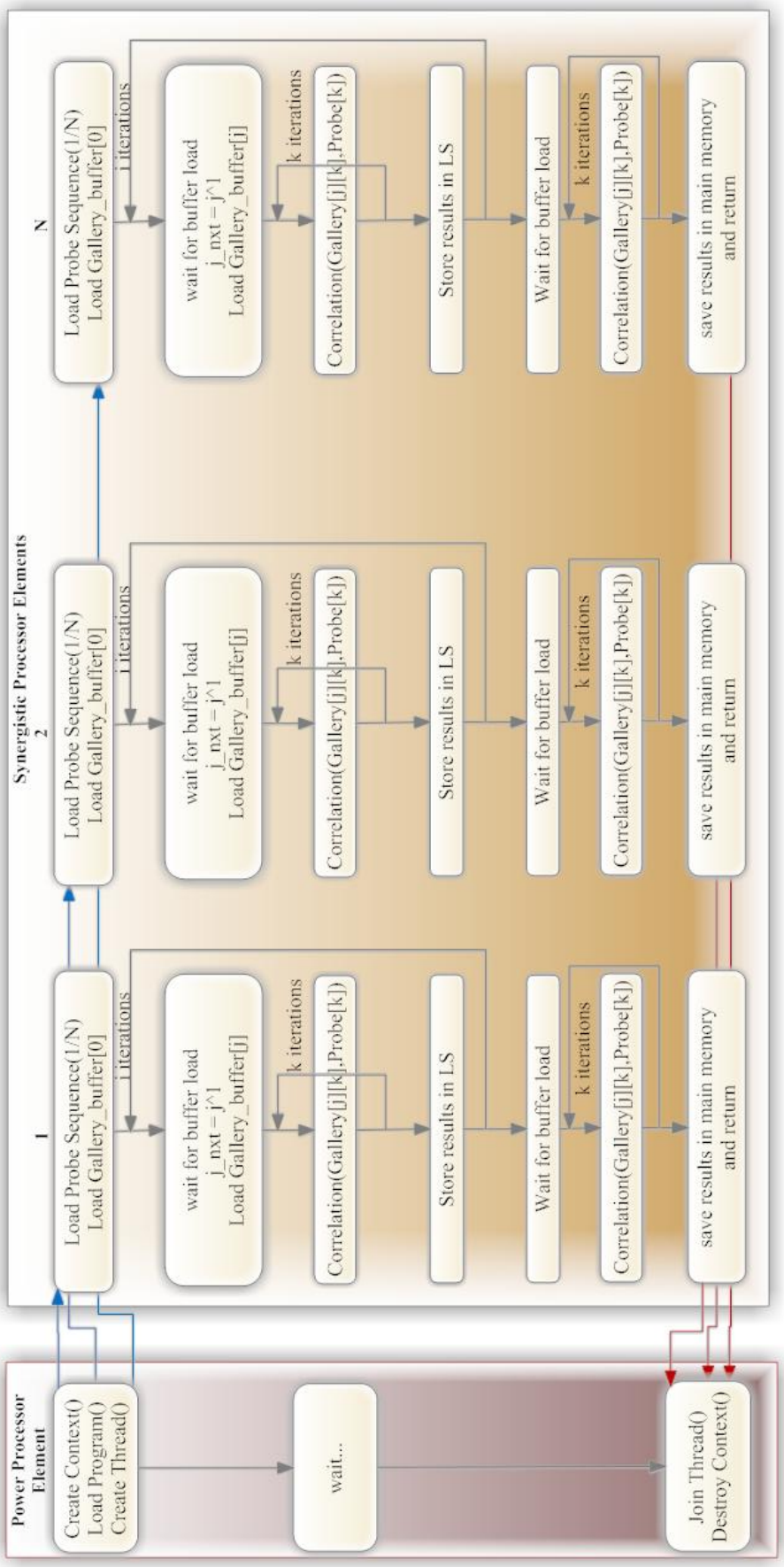


Figure 3-8 Fast Similarity Flow Chart

4 Performance

4.1 Measurement

To measure performance and run-time of parallelized Baseline Algorithm, we used both the Playstation 3 and the IBM Full System Simulator for Cell BE [30] [31]. We used the dynamic profiling method using the hardware counters in order to measure on Hardware. The Cell BE has two software-visible 64-bit time-base registers in PPE:
Time Base Register (TB). This SPR register contains the time-base count value
Time Base Register (TBR). This memory-mapped I/O (MMIO) register specifies the timebase sync mode and the internal reference clock divider setting
and eleven software-visible 32-bit decrementers (down-counters), three in the PPE and one in each SPE. We used one of two 64-bit time-base register to count the PPE and the SPEs a decrementer for each one. The frequency is not the same with the processor. It is equal to 79.8 MHz In time computing, we simply divide the result of registers, clockticks, with their frequency.

The IBM Full System Simulator for Cell B.E. provide to us all the information of the program implementation. But due to the slow performance was only used for small pieces of code and to perform specific functions modeled on SPEs. The SPEs do not complete their process simultaneously. For this reason we chose the largest of the times.

For a general view of the Cell BE performance compared to current technologies, we measured also the modified code on two processors from Intel. In this part, the performance was measured using the function clock () of library time.h.

During the program execution, it had been removed all calls of printf () to avoid possible interferences. All experiments were carried out five times in order to avoid extreme values and we took their average.

4.2 GMM_EM model Performance

The first measurement that we made was the function GMM_EM one. Firstly, we wanted to see the performance difference if we would change the data from double precision to single precision to x86 processors.

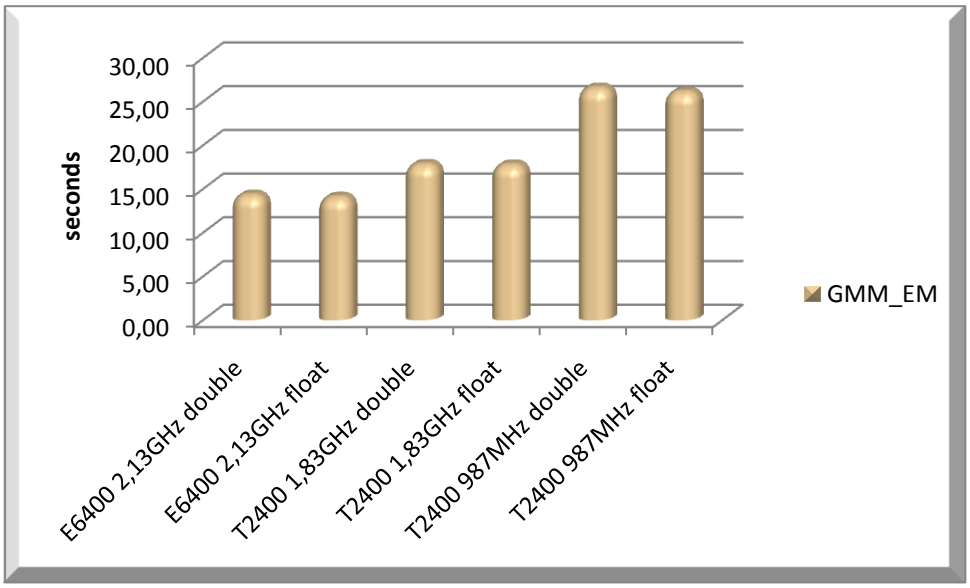


Figure 4-1 Intel CPUs double/float

seconds	Intel CPUs double/float					
	E6400 2,13 GHz double	E6400 2,13 GHz float	T2400 1,83 GHz double	T2400 1,83 GHz float	T2400 987 MHz double	T2400 987 MHz float
GMM_EM	14.31	14.07	17.82	17.76	26.56	26.12

Table 4-1 Intel CPUs double/float

The results showed that there is no difference in the performance of the algorithm when the data is single precision on Intel processors. For this reason we will keep the data in double precision for measurements below.

Then, we count the execution time of GMM_EM adding also our implementations by using 6 SPEs, SIMD instructions, and loop unrolling.

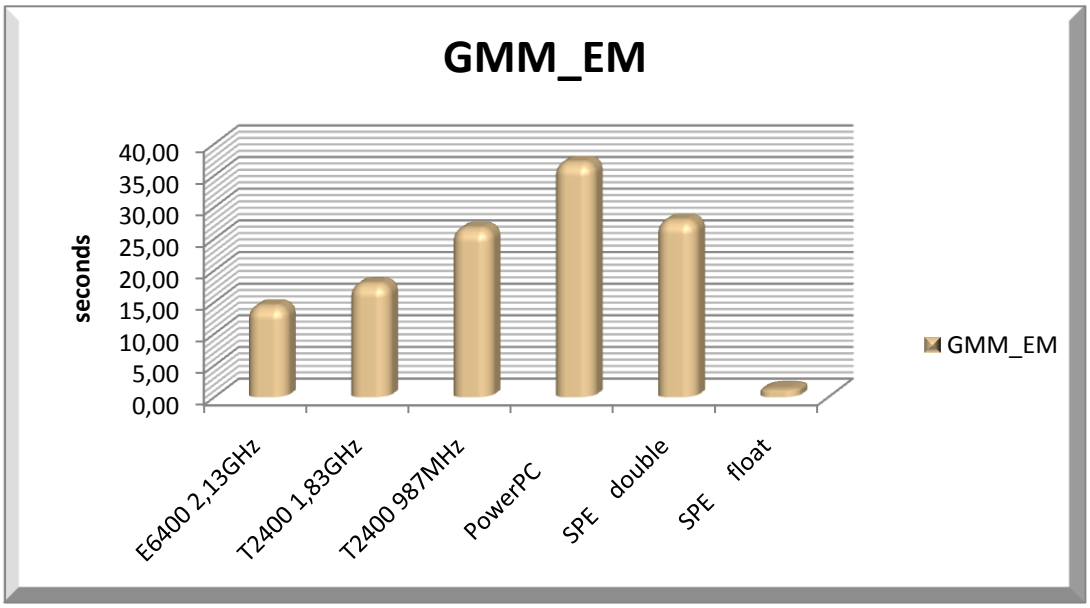


Figure 4-2 GMM_EM model execution time

seconds	GMM_EM					
	E6400 2,13 GHz	T2400 1,83 GHz	T2400 987 MHz	PowerPC	SPE double	SPE float
GMM_EM	14.31	17.82	26.56	36.99	27.84	1.25

Table 4-2 GMM_EM model execution time

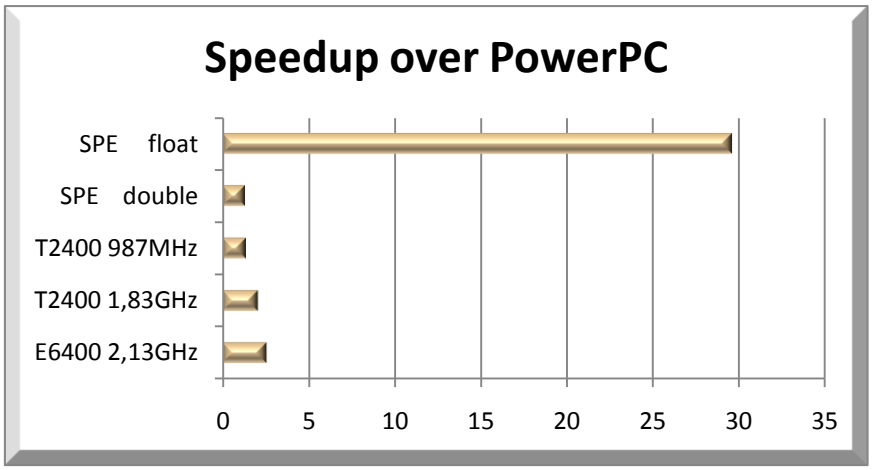


Figure 4-3 Speedup over PowerPC

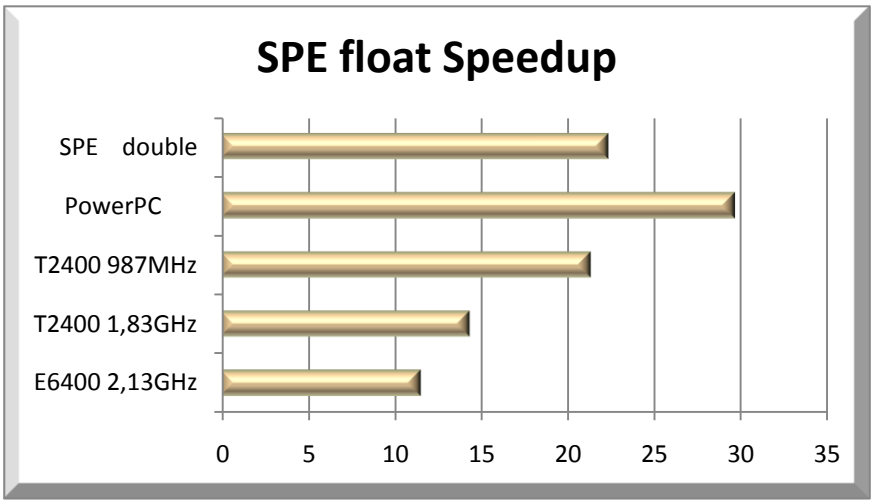


Figure 4-4 SPE float Speedup

From the above results, we noticed that the PPE execution is too slow, even though its frequency is by far the greatest (3.2 GHz). Also, we marked that the optimization of double precision using SIMD is very small and worst than this of Intel processors. On the other hand, we see that by using single precision, the speedup became too large, around 30 times faster than the PPE. Also, the performance is quite good over the E6400, approximately at 11.44 times better. This large performance increase is explained by the fact that the SPEs are made to perform common mathematical operations faster than other processors.

To this point, we present measurements of modeled function GMM_EM to note the improvement in the method of Loop Unrolling.

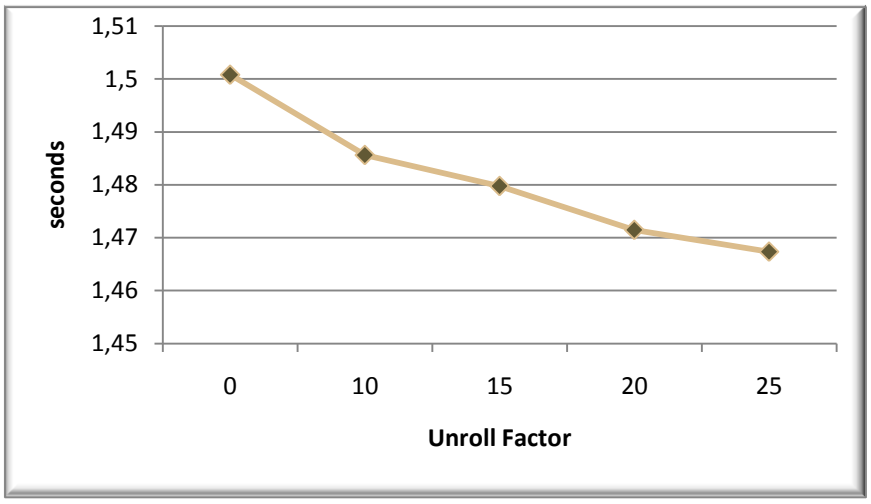


Figure 4-5 Loop Unrolling performance in seconds

As we predicted in theory, the improvement by the loop unrolling method has no significant impact, as loop branches are not so time consuming as the other computing intensive pieces of code.

Finally, the measurements of Cell BE processors with more than 6 cores present some interest. The measurements that have been taken using the IBM Full System Simulator, as Playstation provides to developers only 6 cores.

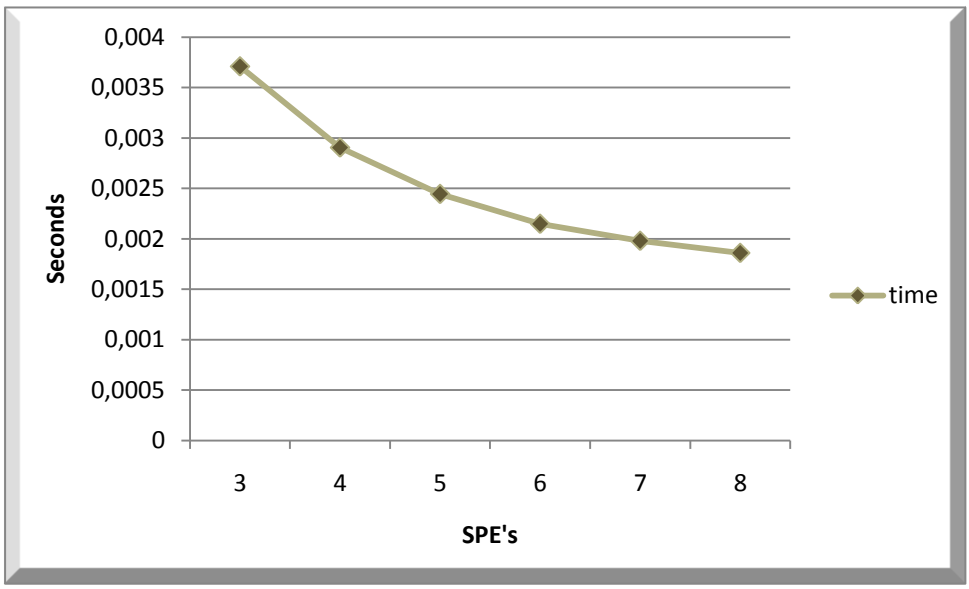


Figure 4-6 Execution time with multiple SPEs

With these results, we can see that there is a significant increase in performance using 4 SPEs instead of 3. The performance is improved by using more SPEs, but as we can see from figure 4.5, after the 8-core, theoretical, performance does not increase significantly.

4.3 Fast Similarity Performance

Fast Similarity is the second and the last function that was optimized. In this case, it did not exist a separate model function as originally it was constituted as a separate single executable file. Also, it includes optimization of character SIMD instructions which is not common enough.

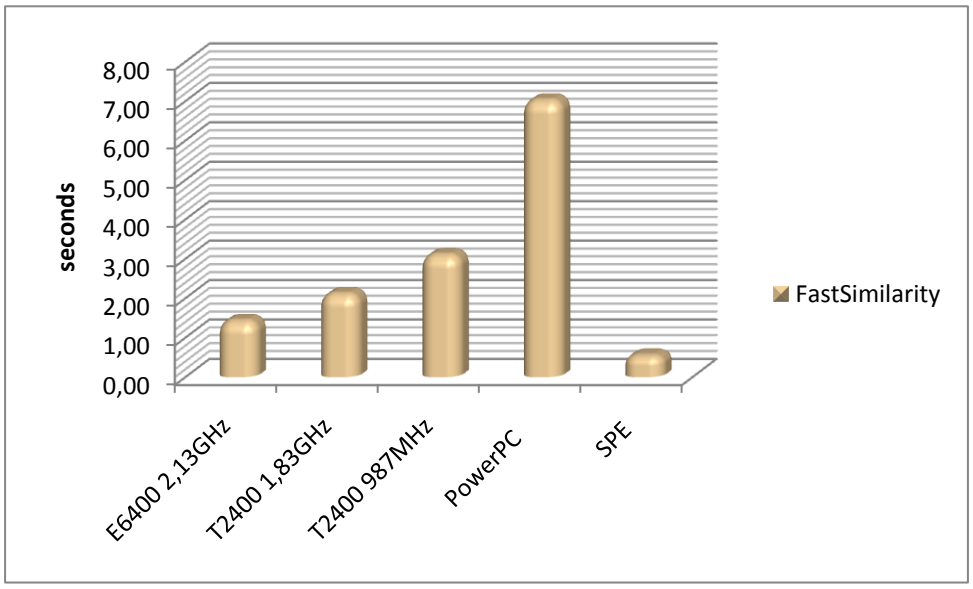


Figure 4-7 Fast Similarity Execution time

	Fast Similarity				
	E6400 2,13GHz	T2400 1,83 GHz	T2400 987 MHz	PowerPC	SPE
seconds	1.425	2.109	3.099	7.021	0.545

Table 4-3 Fast Similarity Execution time

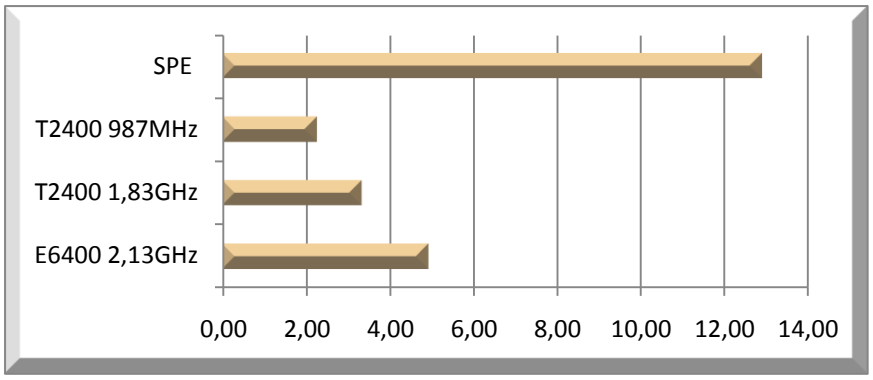


Figure 4-8 Speedup over PowerPC

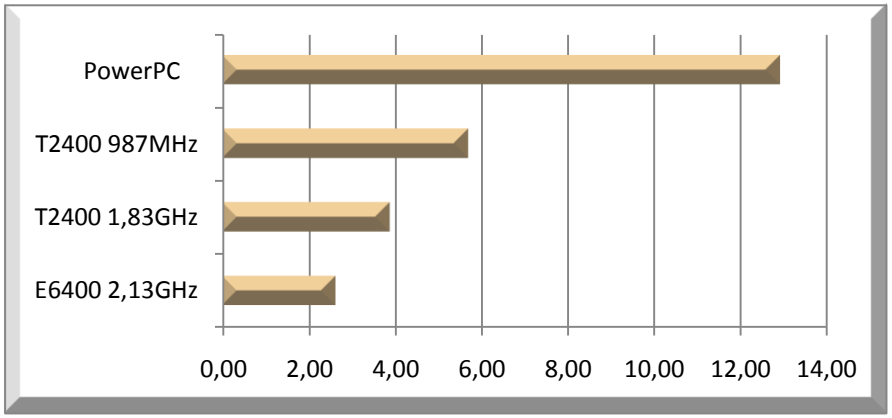


Figure 4-9 SPE Speedup

The results show that there is a significant improvement in the performance of Fast Similarity. Particularly, the 12.89 times improvement over the PPE is quite important and close to the 16 times theoretical one. If we consider also the time needed to transfer data with some calculated data that can be vectorized, then the result is entirely satisfactory. Also quite good is the performance of SPEs compared to the E6400. The 2.62-time faster execution is a good result.

4.4 Total Performance

After the completion of the two computing intensive functions of the Baseline Algorithm and having the desired results, we pass the overall efficiency of the algorithm calculation. Although the measurements that were taken from the two functions were much better than the rest, the overall plan we will see whether the functions that were not achieved at SPEs affect. This is because until now we have noticed that the PPE on his own has the worst performance.

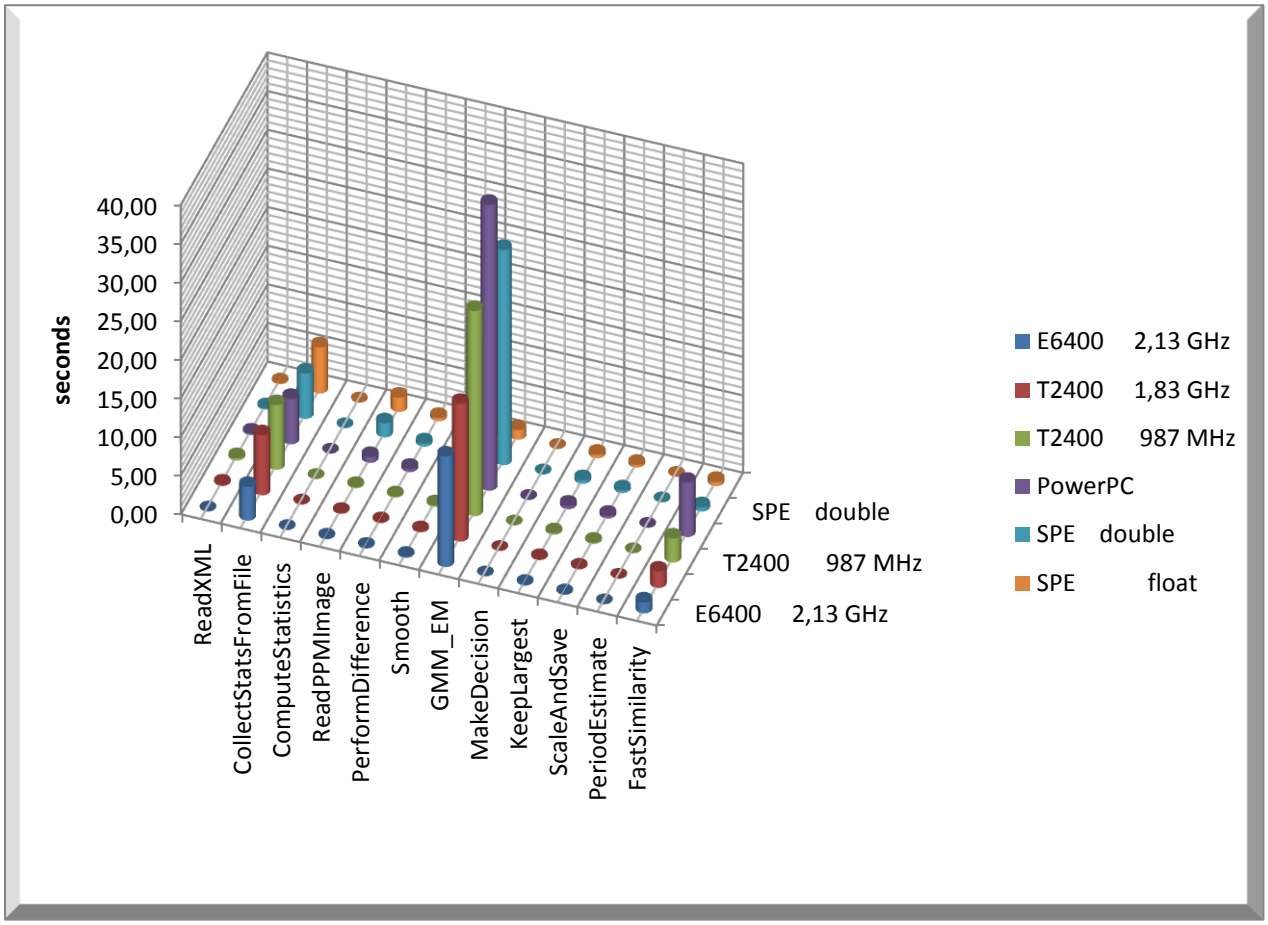


Figure 4-10 Functions Execution time

Seconds	Baseline Algorithm					
	E6400 2,13GHz	T2400 1,83 GHz	T2400 987 MHz	PowerPC	SPE double	SPE float
ReadXML	0.0163	0.1800	0.2984	0.1918	0.1823	0.1092
CollectStatsFromFile	4.3664	7.7593	8.3708	5.8714	5.8566	5.9657
ComputeStatistics	0.0124	0.0188	0.0316	0.0194	0.0195	0.0547
ReadPPMImage	0.0963	0.1155	0.1659	0.7561	0.0466	0.0499
PerformDifference	0.0913	0.1002	0.1496	0.4192	0.4228	0.4237
Smooth	0.1233	0.1379	0.1970	0.3233	0.2188	0.2412
GMM_EM	14.3078	17.8171	26.5610	36.9906	27.8414	1.2507
MakeDecision	0.0090	0.0085	0.0116	0.0343	0.0325	0.0312
KeepLargest	0.0931	0.1304	0.1719	0.4531	0.4510	0.4568
ScaleAndSave	0.0895	0.1031	0.1591	0.3394	0.3415	0.3423
PeriodEstimate	0.0034	0.0118	0.0169	0.0144	0.0148	0.0149
FastSimilarity	1.4250	2.1086	3.0987	7.0205	0.5522	0.5445

Table 4-4 Functions Execution time

From the above results, we noticed that the functions, which were not implemented in SPEs, are slower. In order to reduce the execution time of the previous mentioned functions we did another parallelization. While SPEs are executing the code, PPE stays inactive. Instead of leaving PPE inactive, we use it to execute another function. Particularly, the function ReadPPMImage is executed by the PPE while the SPEs are calculating GMM_EM. This parallelization technique almost eliminated the ReadPPMImage execution time. Otherwise, this function will last more than in other processors. Finally, we note that the functions implemented in SPEs have been joined well and the results are as well as the results we got from the models.

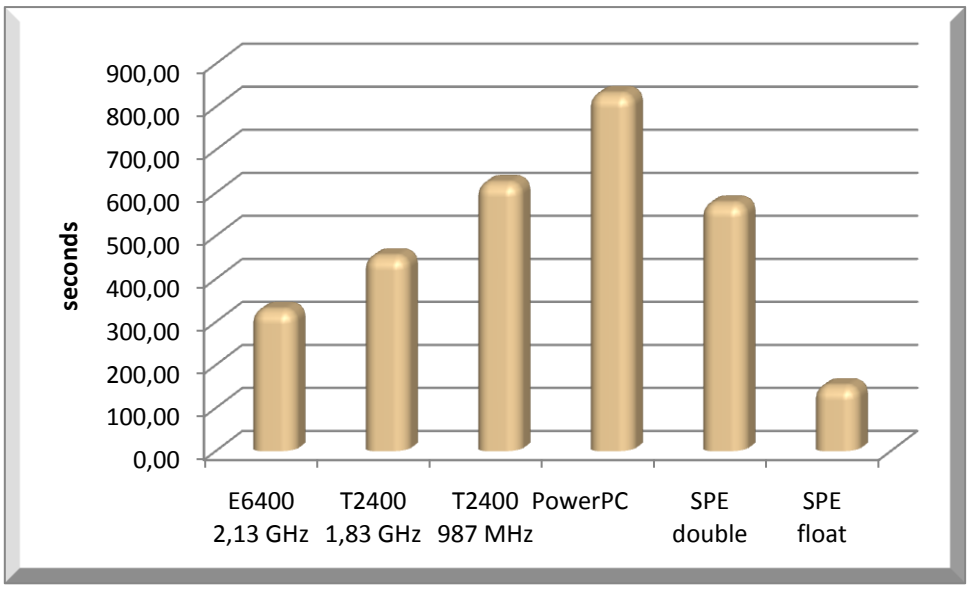


Figure 4-11 Baseline Algorithm Execution time

Seconds	Baseline Algorithm					
	E6400 2,13GHz	T2400 1,83 GHz	T2400 987 MHz	PowerPC	SPE double	SPE float
Total	328.1920	452.7460	623.2920	830.0983	575.6382	150.7546

Table 4-5 Baseline Algorithm Execution time

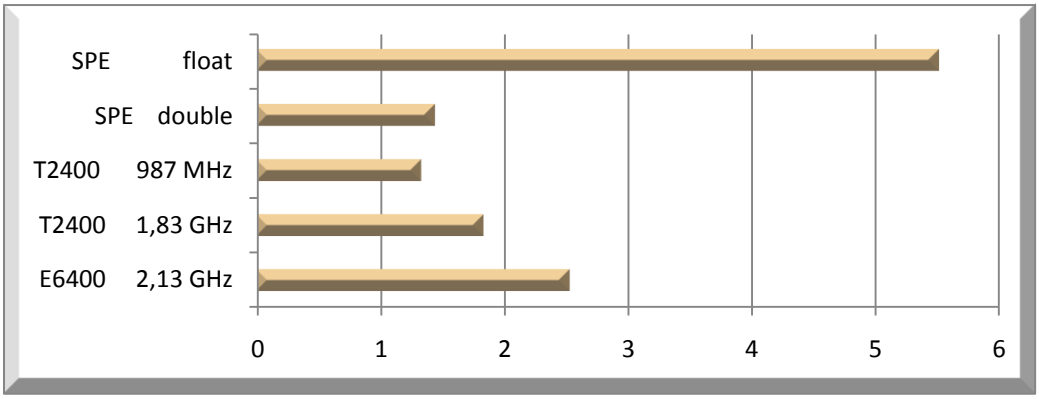


Figure 4-12 Baseline Algorithm speedup over PowerPC

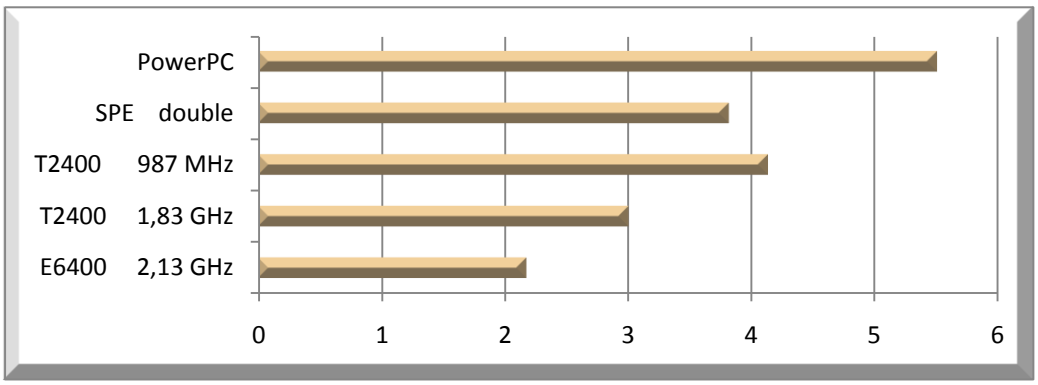


Figure 4-13 Baseline Algorithm SPE float speedup

From the total execution time of the algorithm, we concluded that the final optimization using the SPEs is quite far from the performance we had on individual functions. The execution speed of the rest of the PPE program reduces significantly the performance. Despite this reduction, the final results are satisfactory based on the capabilities of the Cell BE. The 5.51-time faster execution of the PPE is a result that is consistent with the original theory, it should be 6 times faster by using 6 SPEs. We also achieve a much better performance than the E6400 one. The 2.18-time faster execution leaves us satisfied to some extent by the capabilities of Cell BE.

4.5 Hardware and Software, Tools and Problems

In this thesis, we used the IBM SDK for Multicore Acceleration Version 3.0 which was installed in a desktop PC with a processor Intel E 6400 2.13 GHz, 2 GB RAM and Western Digital SATA II 7.200 rpm 250 GB hard drive. Due to the requirements of the functional simulator, Fedora 7 was used. The OS is old and does not support many of the parts may have a new pc. The 3.1 version of the SDK, even though is supported by the Fedora 9, could not be installed because there was a bug on Intel processors. With the Simulator, we can get very good measurements and move the debugging in-depth. Its disadvantage is the slow run in cycle mode which performs a complete simulation. Particularly, the final project would take months to complete an execution. But it was helpful in order to get measurements for more than 6 cores of Playstation 3, but only for small pieces of code.

The execution of experiments and the implementations became mainly in real hardware based on a Playstation 3 (2009 model). Although this model had a 80 GB hard disk, we replaced this with a hard drive Seagate SATA II 7.200 320 GB one for best performance. The Cell BE processor that contains the Playstation 3 provides the use of 6 SPEs. The OS that was installed was YDL 6.0 as it was considered as the most friendly to the case. The code was cross compiled and executed to the PC and the Playstation 3.

The hard disk of the Playstation 3 is quite slow for two reasons. Due to the standards and due to the Hypervisor that is installed on the Playstation 3. The Hypervisor is like a virtual machine that connect the OS with hardware. The first problem was overcame by changing the hard disk. Beyond this, by the careful optimization of the code, we accomplish to use negligibly the disk and achieve similar performance of a PC.

In GMM_EM, we tried to store the values calculated by the cores at the beginning (average, mean, covariance) to the LS and the other SPEs load them from there. This was accomplished and worked at IBM SDK simulator but a problem presented in the performance of the hardware. It presented a conflict error to calculation completion between the transfer value and the signal sending. Finally, we implement it by using the main memory for storing and we overcame this problem without having significant losses in performance and execution time.

During the SPEs using, we had to change the order of addition. Due to the fact that the floating point numbers addition is not associative, it was displayed an round-off error from the tenth digit after the decimal point. In this thesis, this error does not appear significant affection to the result and it can be ignored.

For the calculation of computing intensive points, it was used the Intel Vtune Analyzer 9. Although it is supported by fedora 9 and subsequent versions, it was installed and worked perfectly in Fedora 7 without any problems.

The computers used for measurements were a desktop PC with a processor Intel E 6400 2.13 GHz, 2 GB RAM and Western Digital SATA II 7.200 rpm 250 GB hard drive and a laptop with a processor Intel T2400 1.83 GHz/987MHz, 2 GB RAM and hard Disk Seagate SATA II 7.200 320 GB. Fedora 7 had been installed to all.

Double precision operations have latency 13 clock cycles. The first six cycles are actually stalls in which no other instruction can be issued. The SPU is not only able to process multiple values at once through its pipeline, it can also dual-issue instructions through different pipelines. The SPU has two pipelines, even (sometimes called pipeline 0 or the execute pipeline) and odd (sometimes called pipeline 1 or the load pipeline). While SPU execute a double precision operation, Dual-issue is not allowed with these instructions either. [32]

4.6 Verification

At this point we mention the verification procedure of the modified code. There were four verification stages. The verification was simple, we examined if the similarity value was equal to the values of the original code. The first stage we had to examine the results after porting the code to PPE only. In the next stage we verified the results of GMM_EM and Fast Similarity separately. During GMM_EM results verification there were some differences, which were caused by the order of addition, but as we mentioned they were not important. On the other hand, Fast Similarity verification was successful. At the final verification stage we examined the similarity values, while we were using optimized GMM_EM and Fast Similarity. The values were equal to the initial similarity values.

5 Conclusions and Future Work

5.1 Conclusions

Having as main objective of this thesis to study and develop the capabilities of Cell BE through the Baseline Algorithm, we can say that this was succeeded to a large extent. The parallelization of computing intensive functions implemented achieving a very good performance.

The Cell BE programming is quite difficult and requires very good knowledge of programming, hardware and proper management of memory. During the implementation several problems were faced both at the software and hardware level. Most of these were confronted and many of these that they have failed to be solved were overcome with alternative methods.

The final result was satisfactory and the change of data from double precision to single precision did not play any role. Otherwise the results would not be satisfactory. This is a disadvantage of the Cell BE as most scientific applications require high precision.

While the Cell BE was designed for gaming, it may eventually be used for scientific purposes. But an important prerequisite for a project in order to have very good performance and very good results is to be completely designed on this processor from the beginning.

5.2 Future Work

The final performance that we got for the Baseline Algorithm compared to a modern processor was just satisfactory. The improvement of double precision execution speed and the existence of double vector in PPE would help the algorithm improvement. This thesis has as main method the Function-offload method. Maybe, the choice of another model, such as Streaming model, have better results. However, the complexity of such a model usage is large.

A different approach to the problem could be done with two equally modern methods: the algorithm programming to GPU or the implementation of a FPGA. A CPU vendor has been forced to integrate multiple cores onto a single die by difficulties in scaling single-thread performance without undue power dissipation. On the contrary, General Purpose computing on Graphics processing Units and Field-Programmable Gate Array, that are based on software/hardware co-design [33], are becoming increasingly popular in order to assist general purpose processors in performing complex and intensive computations on accelerator hardware.

5.2.1 General Purpose Programming using GPU's

Future computer systems will certainly include some accelerators and particularly the GPU. Today, accelerators are primarily available as add-in boards. In the future they will probably be located on-chip with the CPU, so as to reduce communication overhead. [33]

GPUs have a lot of advantages as they are fast, cheap, low-power (watts per flop), and amplify the streaming method which is the future, as it is commonly accepted. On the other hand, they are still enough specialized, hard to program and they deal some bandwidth problems. Nowadays, GPUs score performance over 500 Gflops, Cell BE scores approximately 100 Gflops while a current processor can reach the performance of about 70Gflops (Intel Core *i7* 965 XE)
GPU Architecture is faster because:

- it has simplified pipeline architecture,
- it is highly paralleled,
- its memory is near the processor in order to reduce the cache needs,
- it allows scaling by hiding its true parallelism, that can increase number of processors
- and finally it allows to a system having a lot of GPUs [34]

CUDA, (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA, and CAL (AMD Compute Abstraction Layer) are new language APIs and development environments for programming GPUs without the need to map traditional OpenGL and DirectX APIs to general purpose operations. Domain specific parallel libraries, such as a recent scan primitives implementation can be used as building blocks to ease parallel programming on the GPU.

GPUs are inexpensive, commodity parallel devices with huge market penetration. They have already been employed as powerful coprocessors for a large number of applications including games and 3-D physics simulation. The main advantages of the GPU as an accelerator stem from its high memory bandwidth and a large number of programmable cores with thousands of hardware thread contexts executing programs in a single program, multiple data (SPMD) fashion. Moreover, GPUs are flexible and easy to program using high level languages and APIs which abstract away hardware details.

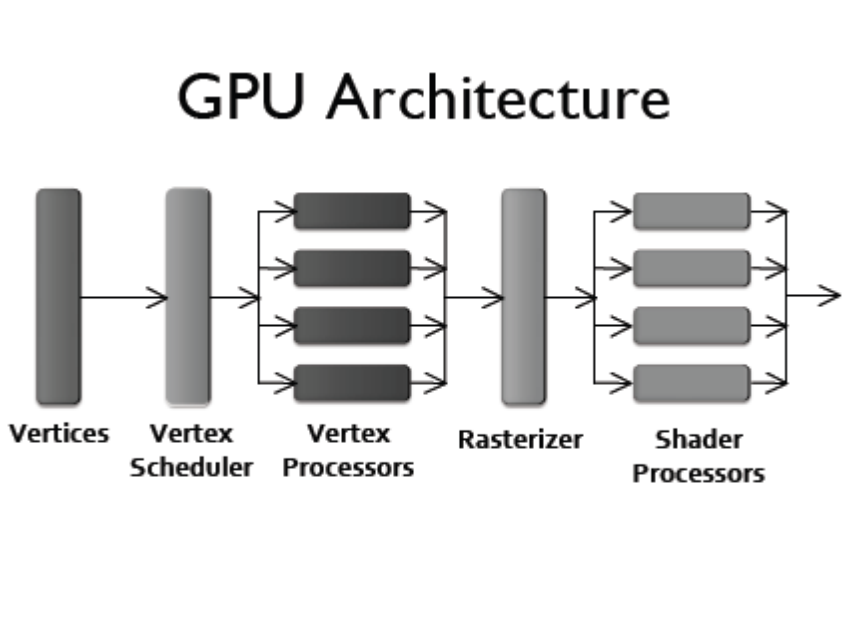


Figure 5-1 Simplified Block Diagram of GPU Architecture [35]

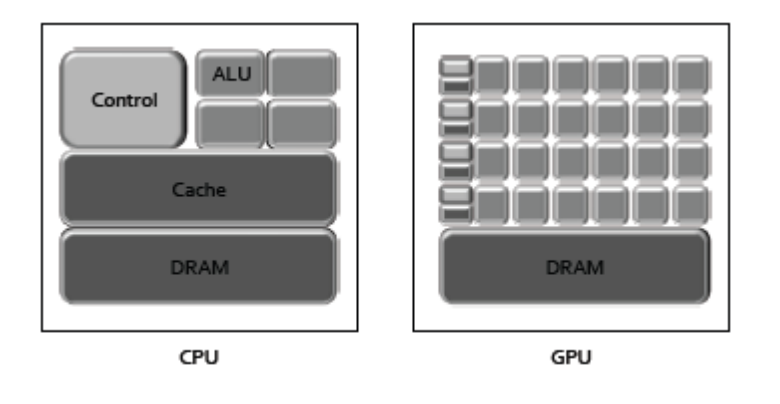


Figure 5-2 CPU Block Diagram versus GPU Block Diagram [35]

Furthermore, GPUs have their own off-chip device memory, and data must be transferred from main memory to device memory via the PCI-Express bus. Memory transfer from system memory to GPU memory (host to device and device to host) remains relatively slow and can often be a bottleneck in the applications. Most current GPUs offer support for only single precision, while many scientific applications require double precision support. However, using CUDA's streaming interface, programmers can batch kernels that run back to back, and increase program execution efficiency by overlapping computations with data communication.

Particularly for the Baseline Algorithm, GPU usage as accelerator would be ideal. As it is not demanded double precision usage neither to GMM_EM nor to Fast Similarity we can overlook the disadvantage of some GPUs which do not support double precision. In addition, the data amount which must to be transferred to GPU memory is quite little for both functions. GPU can eliminate this problem by using some API, like CUDA.

Generally, it could be succeeded larger performance acceleration, than the one we succeeded with Cell BE, as we would use a faster control processor that PPE and GPU would score a higher performance than the SPEs one. The last point that should be mentioned is the complexity of the code development, it is as difficult on CUDA API as for Cell BE.

5.2.2 Field-Programmable Gate Array (FPGA) Programming

FPGAs are essentially high density arrays of uncommitted logic and are very flexible in that developers can directly steer module-to-module hardware infrastructure and trade-off resources and performance by selecting the appropriate level of parallelism to implement an algorithm. In the FPGA co processing example, the hardware fabric is used to approximate a custom chip. [33] FPGAs consist of hundreds of thousands of programmable logic blocks and programmable interconnects that can be used to create custom logic functions, and many FPGA products also include some hardwired functionality for common functions. For instance, the Xilinx Virtex II Pro FPGA also integrates up to two 32-bit RISC PowerPC405 cores. The dataflow of an application is exploited in FPGAs through parallelism and pipelining.

Moreover, FPGAs accelerate high performance computing applications by their ability to exploit the parallelism inherent in the algorithms employed. [36] There are several levels of parallelism to address. A good starting point is to structure the high performance computing application for multi-threaded execution suitable for parallel execution across a grid of processors. This is task-level parallelism, exploited by cluster computing. There are software packages available that can take legacy applications and transform them into a structure suitable for parallel execution. A second level of parallelism lies at the instruction level. Conventional processors support the simultaneous execution of a limited number of instructions.

Furthermore, FPGAs offer much deeper pipelining, and therefore can support a much larger number of simultaneously executing “in-flight” instructions. Data parallelism is a third level that FPGAs can readily exploit. The devices have a fine-grained architecture designed for parallel execution. Thus, they can be configured to perform a set of operations on a large number of data sets simultaneously. This parallel execution performs the equivalent work of numerous conventional processors all in a single device.

FPGA applications are mostly programmed using hardware description languages such as VHDL and Verilog. Recently there has been a growing trend to use high level languages such as SystemC and Handel-C which aim to raise FPGA programming from gate-level to a high-level, modified C syntax.

The reconfiguration overhead for FPGAs needs to be taken into account. The reconfiguration (or initialization) process generally takes on the order of seconds for applications on the FPGA boards. To reduce the impact of this overhead, developers can potentially overlap reconfiguration with other non-conflicting tasks. Furthermore, users may not need to reconfigure after initialization, in which case the configuration is represented by a small, onetime cost. An FPGA coprocessor programmed in order to hardware-execute key application tasks can typically provide a 2X to 3X system performance boost while simultaneously reducing power requirements 40% as compared to adding a second processor or even a second core. Fine tuning of the FPGA to the application’s needs can achieve performance increases greater than 10X.

It is generally accepted that FPGAs is the best solution for high performance computing applications. The same applies in the case of the Baseline Algorithm. The data amount of computing intensive functions is not large enough to create big overhead by using FPGA. Considering that desirable results could be suceeded by single precision operations, the chip space of necessary operations is being reduced, while the simultaneous operation number is being increased. Finally, this method disadvantages are still the cost production and the highly demanding code development.

6 References

[1] IBM “Programming the Cell Broadband Engine: Examples and Best Practices” December 2007, p. 3-6.

[2] “Wikipedia - Cell(microprocessor) History”, http://en.wikipedia.org/wiki/Cell_microprocessor

[3] IBM Corporation; Sony Computer Entertainment Incorporated; Toshiba Corporation “Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor” 12 May 2008, p. 51-54, 65-67.

[4] IBM Corporation; Sony Computer Entertainment Incorporated “Cell Broadband Engine Architecture” 11 October 2007

[5] Sony Computer Entertainment Incorporated/Toshiba/IBM “Cell Broadband Engine CMOS SOI 65 nm Hardware Initialization Guide” 8 June 2007, p. 21.

[6] Dr. J. Nieplocha, Dr. A. Marquez, Dr. F. Petrini, and Dr. D. Chavarria “UNCONVENTIONAL ARCHITECTURES for High-Throughput Sciences” http://www.scidacreview.org/0703/html/hardware.html 2007

[7]” Wikipedia Playstation 3 Hardware” http://en.wikipedia.org/wiki/PlayStation_3_hardware

[8] Kiran “Debian-Installer for PLAYSTATION 3”, http://www.keshi.org/moin/PS3/Debian/Installer 24 November 2008

[9] Fedora PS3 PlayStation Port of Fedora, http://fedoraproject.org/wiki/PlayStation, 2 November 2009

[10] Gentoo “Gentoo Linux for PS3 Development” 6 February 2008

[11] OpenSuse “Open Suse on PS3”, http://www.gentoo.org/proj/en/base/ppc64/ps3/ 20 January 2010

[12] Ubuntu “psubuntu.com Linux on a Playstation 3”, http://psubuntu.com/ January 2010

[13] YellowDogLinux “A place for Yellow Dog Linux users”, http://www.yellowdoglinux.com/ January 2010

[14] S. Sarkar, P. J. Phillips, Z. Liu, I. Robledo Vega, P. Grother, K. W. Bowyer, The “HumanID Gait Challenge Problem: Data Sets, Performance, and Analysis”, IEEE Transactions on pattern analysis and machine intelligence, vol. 27, no. 2, February 2005, p. 162-169.

[15] Sony Computer Entertainment Inc. “PlayStation 3 "Safety & Support" manual” http://www.playstation.com/manual/pdf/PS3-01-1.6_1.pdf 2007

[16] “PlayStation 3 Secrets” http://www.edepot.com

[17] IBM, “VectorSIMD Multimedia Extension Technology,” October 2006.

[18] IBM Corporation; Sony Computer Entertainment Incorporated; Toshiba Corporation “STI Center of Competence for the Cell Broadband Engine Processor” http://sti.cc.gatech.edu/index.html

[19] Wetzel/Poughkeepsie/IBM “PowerPC User Instruction Set Architecture” Version 2.02 Joe 28 January 2005

[20] IBM Corporation; Sony Computer Entertainment Incorporated; Toshiba Corporation ”Synergistic Processor Unit Instruction Set Architecture” 27 January 2007

[21] R. Duda, P. Hart, and D. Stork, Pattern Classification. Wiley, 2001.

[22] Teknomo, Kardi. “Similarity Measurement” “Mahalanobis distance” 2006

[23] Intel Corporation.” Intel® VTune™ Performance Analyzer 9.1 for Linux” 2009.

[24] SCEI , Toshiba , IBM “C/C++ Language Extensions for Cell Broadband Engine Architecture” 27 February 2008

[25] IBM “SIMD Math Library API Reference” 2007

[26] IBM “SDK for Multicore Acceleration, Programming Tutorial 3.1” 2008, p. 123-126

[27] IBM “CBEA JSRE Series: SPE Runtime Management Library Version 2.3” October 2008

[30] IBM “Performance Analysis with the IBM Full-System Simulator” September 2007

[31] IBM “IBM Full-System Simulator User’s Guide” September 2007

[32] Jonathan Bartlett “Programming high-performance applications on the Cell BE processor, Part 4: Program the SPU for performance” http://www.ibm.com/developerworks/power/library/pa-linuxps3-4/index.html 6 March 2007

[33] S. Che, J. Li, J. W. Sheaffer, K. Skadron, John Lach “Accelerating Compute-Intensive Applications with GPUs and FPGAs”

[34] Tomas Oppelstrup "Introduction to GPU programming" 2008

[35] Wolfgang Banzhaf "Accelerating Evolutionary Computation through Graphics Processing Units" 2009

[36] XtremeData, Inc. "FPGA Acceleration in HPC: A Case Study in Financial Analytics" November 2006

[37] "Wikipedia – Gait Definition", http://en.wikipedia.org/wiki/Gait