Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Bc. Václav Klecanda

Implementace algoritmů pro zpracování obrazu na IBM Cell

Kabinet software a výuky informatiky
Vedoucí diplomové práce: **Mgr. Václav Krajíček**
Studijní program: **Informatika, softwarové systémy**

Rád bych poděkoval magistru Václavu Krajíčkovi za jeho vedení, rady, názory a připomínky a vůbec za jeho podporu. Dále chci poděkovat svým rodičům a mé přítelkyni za trpělivost.

I would like to thank to Mgr. Václav Krajíček for leading, his advices, opinions and other support. I also want to thank to my parents and to my girlfriend for their patience during works on this thesis.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 4. Srpna 2009

Václav Klecanda
vlastnoruční podpis

Contents

List of Figures

List of Tables

Název práce: *Implementace algoritmů pro zpracování obrazu na IBM Cell*
Autor: *Bc. Václav Klecanda*
Katedra (ústav): *Kabinet software a výuky informatiky*
Vedoucí diplomové práce: *Mgr. Václav Krajíček*
e-mail vedoucího: *Vaclav.Krajicek@mff.cuni.cz*
Abstrakt:

Práce shrnuje dostupné informace o architektuře IBM Cell/B.E. tak, aby čtenář rychle získal potřebný náhled na problematiku programování pro tuto architekturu. Praktické informace jsou čerpány z vývoje aplikace která implementuje netrivialní algoritmus z oblasti zpracování obrazu, sparse field level set segmentation. Další část obsahuje popis vývoje této aplikace a řešení problémů, které mohou během něj nastat.

Práce zároveň srovnává klasickou a Cell architekturu a popisuje nutné podmínky pro vytvoření efektivní aplikace pro Cell/B.E. Dále obsahuje stručný postup instalace nejdůležitějších vývojových nástrojů. Tento postup si klade za cíl co nejrychleji připravit vše potřebné a zkrátit tak dobu přípravné fáze tak, aby čtenář mohl začít vyvíjet pro Cell/B.E.

Klíčová slova: *programování pro Cell, multicore acceleration, IBM, PS3*

Title: *Implementation of image processing algorithms on IBM Cell*
Author: *Bc. Václav Klecanda*
Department: *Department of Software and Computer Science Education*
Supervisor: *Mgr. Václav Krajíček*
Supervisor's e-mail address: *Vaclav.Krajicek@mff.cuni.cz*
Abstract:

This work summarize available information about IBM Cell/B.E. architecture to let the reader create a necessary overview for programming for this architecture. Practical information are based on development of an application that implements nontrivial image processing algorithm, sparse field level set segmentation. Next section contains description of the application development and associated problems solving.

The work compares common and Cell B.E. architectures and describes conditions necessary for creation of an effective Cell/B.E. application. The work also contains brief procedure of the most important development tools installation. This procedure has to prepare everything necessary as fast as possible and thus to shorten the duration of the preparation phase to let the reader to start development.

Keywords: *CellBE programming, multi-core acceleration, IBM, PS3*

Chapter 1

Introduction

The production of x86 platform processors has started a big frequency competition. The manufacturers have been releasing processors with higher and higher operating frequency. Behind this competition there has been a significant amount of research of new production technologies that allow to integrate more transistors onto a smaller area. Gradually the manufacturers started to realize that it is impossible to continue the competition forever.

More computing units started to be integrated into a single processor. The first among common desktop processors was Intel's Pentium® with hyper-threading. This processor was able to execute two threads at a time. Since then a quick boom of multi-core integration started even among other big processor manufacturers.

Integration of more execution cores allows less power consumption. This factor is nowadays especially important due to processor integration into laptops and even due to environmental issues. Therefore current effort of the processor manufacturers is to gain the best performance to power consumption ratio.

General purpose cores are integrated into the current common processors. It means they have a pipeline for instruction execution composed of several stages. Instruction can be in various states in the pipeline, e.g. fetched, awaiting operands, ready, executed. Instructions do not flow among the stages in the order that designates the program but the order is decided by the processor itself. The decision is based on variety of factors and predictions. One of the pipeline stages is a branch-prediction unit. It has to predict the most probable flow of the executed program. When this prediction is false the whole pipeline has to be discarded and execution of the right branch of the program has to be started. Processors suffer heavily from these mispredictions because they lead into big execution holes in which the processor pipeline stalls or is being reset.

Cache misses are another general purpose processor suffering. A cache miss occurs when the requested data are not within processor cache and have to be loaded from the main memory.

Although many improvements were implemented into the general purpose processors they still suffer from the described problems. This is one of the reasons why a collaboration of three big companies IBM, Sony, and Toshiba started development of the Cell/B.E. processor. It is a multi-core processor which has high performance to power consumption ratio and is able to overcome the problems that the general purpose processors suffer from. That is because it allows the programmer to manage processor cache and branch prediction unit to a certain degree. The next chapter will describe the processor in more details.

Chapter 2

Cell/B.E. platform

This chapter will introduce the Cell Broadband Engine processor (Cell/B.E.), the whole platform and its specific details. The particular Cell/B.E. processor will be described and illustrated. All the information were taken from [6].

Cell/B.E. processor is representative of a new generation of IBM's Cell/B.E. platform family. Cell/B.E. is an asymmetric, high-performance multi-core processor that combines eight synergistic processing elements (SPE) and a Power Processing Element (PPE), which is a general-purpose IBM Power PC$^{®}$ core. The next important part is a central memory element. The PPE can operate with the central memory directly while the SPE can access the memory indirectly through DMA. All the elements are connected through a high speed bus (EIB - Element Interconnect Bus). The whole layout is in the figure 2.1.

The Cell/B.E. achieves a significantly better performance per Watt and performance per chip area ratios than conventional high-performance processors. It is more flexible and programmable than single-function and other optimized processors such as graphics processors, or conventional digital signal processors. While a conventional microprocessor may deliver about 20+GFlops of single-precision (32b) floating-point performance, Cell delivers 200+ GFlops (under ideal conditions) at a comparable power consumption.

A number of signal processing and media applications have been implemented on the Cell/B.E. with excellent results. Advanced visualization techniques such as ray-casting, ray-tracing and volume rendering, streaming applications such as media encoders, decoders or encryption and decryption algorithms have also been demonstrated to perform about an order of magnitude better than a conventional processors.

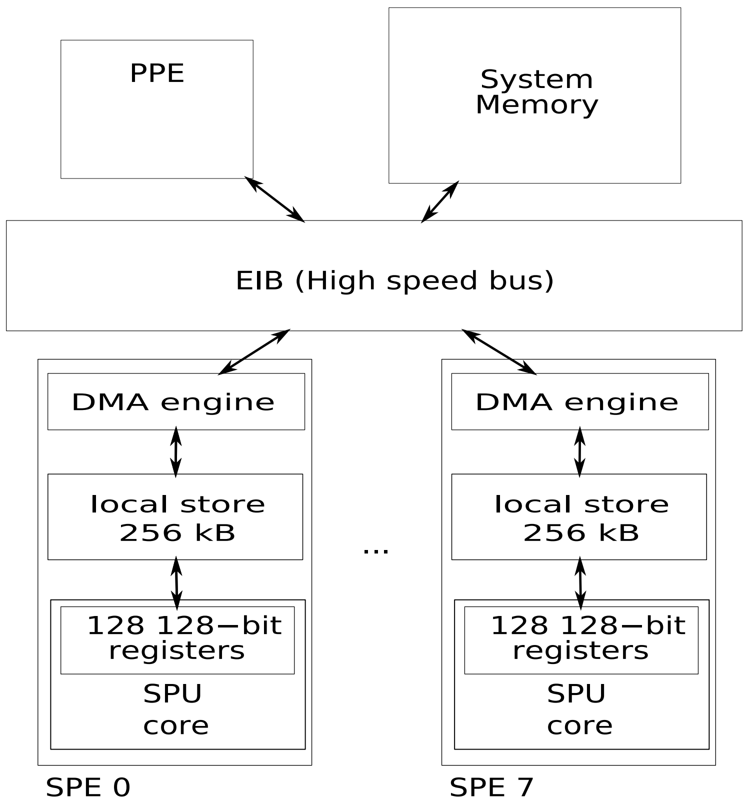


Figure 2.1: One PPE unit along with eight SPE stream processor units and system memory connected together with a high speed EIB bus

2.1 PPE - Power Processing Element

The PPE is derived from IBM Power PC® core. It has 512kB L2 on die cache. It supports the Power Architecture ISA, inherits the memory translation, protection, and SMP coherence model of mainstream 64-bit Power processors. Virtualization, logical partitioning, large pages, and other recent innovations in the Power architecture are supported as well. Programming for the PPE is the same as for conventional processors due to direct access to central memory.

2.2 SPE - Synergistic Processing Element

SPE is an autonomous processor (sometimes called accelerator) targeted for computational intensive applications. Each SPE has a SIMD core (SPU), a high-speed private local store memory and a direct memory access (DMA) engine.

The SPU unit has 128 128-bit wide unified general purpose registers to store all types of data in contrast to traditional RISC processors where registers are divided according data types. It supports a SIMD-RISC instruction set. The SPU has two pipelines, the odd one and the even one, so it can execute two instructions at a time (dual-issue) if some conditions are met. Vectorized operations in various data types configurations can be performed with these registers e.g. two double-precision floats or eight 32bit integers can be processed at single clock tick.

Unlike conventional microprocessors the SPE does not have a hardware cache. Its function represents the small on-chip local store memory under programmer's control. This allows code optimizations that can reduce cache misses. The local store is separated from the main memory i.e. the SPE has its own address space. Therefore any synchronization with other cores is not necessary. The SPE uses the local store as a cache of data stored in the central memory i.e. programmer has to create copies of the data within the local store. The data are transferred through DMA engine which manages transferring data from central memory to local store and vice versa as well as between two SPEs' local stores. We say that data is "DMAed" from source to destination. DMA commands can be issued in many ways such as in synchronous, asynchronous or in scatter-gather manner through DMA lists. The DMA list is an array of pointer-size pairs that defines pieces of memory that shall be transferred within single DMA request. The pieces must not necessarily be continuous. Therefore the Cell/B.E. processor can be viewed as a distributed memory multiprocessor. The local store memory management is a big part of programming for the Cell/B.E.

Programming for the SPE is a bit different compared to programming for a conventional processor. Programmer have always to count with the fact that he/she

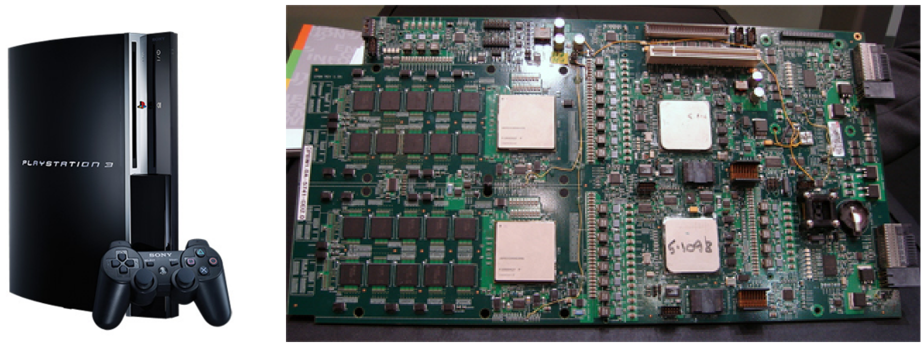


Figure 2.2: Images of Cell/B.E. based machines. Sony's Play Station 3, on the left (image taken from www.boygeniusreport.com), IBM Cell Blade board, on the right (image taken from www.ps3tester.com)

has only 256kB for the program and data. More details about this topic will be described in the next chapter.

The Cell/B.E. is embedded in game console Sony PlayStation 3 (PS3) as well as IBM Blade servers where are two processors on one board, building block. There can be more boards connected in one system forming a powerful and modular machine. We have two PS3 machines available for this work.

Chapter 3

Cell/B.E. programming

Cell/B.E. platform development tools will be described in this chapter. Our experience with the tools will be mentioned as well. Then particular SDK content and tools will be listed. Parallel systems and models will be mentioned later on as well as the relationship to the Cell/B.E. development along with a few design patterns. At the end core configurations and their advantages and disadvantages will be listed finishing with few practical approaches to the Cell/B.E. porting process.

3.1 Cell/B.E. platform development

IBM delivers a SDK for the Cell/B.E. application development. It is made for a Linux platform, in the concrete for the Fedora or the Red Hat distribution. It comes in two flavours. The first is the official non free SDK which has all the features needed for the Cell/B.E. development even for hybrid systems. The purchaser has also a support team ready to help. The next is a free one that is open to wide public and everybody can download it and start developing. The free one does not have full support for hybrid systems nor for development in other languages than C/C++. We have used the free one since we have developed only in C/C++ and for a clean Cell/B.E. processor.

Because the SDK is for Linux operation system its user has to have already a deeper knowledge about this system. There are a few bugs and parts that are not fully finished (see Appendix B) and without the deeper system knowledge is practically impossible to react on an unexpected behaviour during installation or development phase.

We have begun with SDK version 3.0 and Fedora version 8 which were the current versions of needed tools. We have faced a number of obstacles and before we were able to overcome them a new version of SDK (3.1) appeared. Because we wanted to use and describe the latest tools we had to begin from scratch because the new version brought new obstacles as well.

The new version was declared to be compatible with a new version of Fedora, 9 - Sulphur, that had been released at almost the same time as the new SDK version. The previous version of SDK (3.0) was for Fedora 7 Werewolf. We have tried all possible combinations of Fedora distributions and SDK packages to find out if they are compatible with each other. The only result from that testings was finding out that they are not mutually compatible. We have spent plenty of days on this discovery. The SDK is a huge package of software dependent on lots of third party libraries and solutions. They are treated differently within particular distributions and sometimes even versions of the same distribution. The resulting advise is to avoid combination of system versions nor SDK versions nor particular libraries that the SDK components are dependent on. The repository versions of the third party software should be used.

Although there are too much of troubles when different version are combined, a few efforts to get the SDK run on another distributions than Fedora were made. But we think the time spent on this goal is not worth the result.

Finally we installed Fedora 9 Sulphur and SDK 3.1. Although this combination is declared by IBM as tested we have run into few bugs and errors. The process of installation is described in the Appendix B.

3.2 SDK content

The Cell/B.E. SDK is divided into variety of components. Each component is contained in one or more rpm package for easy installation purposes. Here is a list of important available components:

1. Tool chain

   Is a set of tools such as compilers, linkers etc. necessary for actual code generation. There are two tool chains. One is for PPU and the other for SPU.

2. Libraries

   IBM provides several useful libraries for mathematical purposes e.g. linear algebra, FFT, Monte Carlo with the SDK. Another libraries set is for cryptography or SPE run-time management. Code of these libraries is debugged, highly optimized for running on SPEs and SIMDized. It is highly advisible to use the libraries i.e. adapt a code for using the libraries instead of programming own solution.

3. Full system simulator

   Program that can simulate the Cell/B.E. processor on other hardware platforms. It is used mostly in profiling stage because simulator can simulate actual computation of a code in cycle precision. It can be of course used when programmer has an actual Cell/B.E. hardware available, but the simulation is incredibly slow.

4. IDE

   IDE is in fact version 3.2 of Eclipse with integration of debugging, profiling, Cell/B.E. machine management and other features that makes development for the Cell/B.E. easier and more comfortable.

3.3 Parallel systems & Cell/B.E.

Parallelism depends on type of system where the program will be run. There are two basic kind of parallel systems:

1. shared-memory system

   Is a multi-processor system with one shared memory which all processor can see. Processors has to synchronize access to the memory otherwise race conditions will rise.

2. distributed-memory system

   Is system where each processor has its own private memory. There is no need for any synchronization.

In context of parallel systems Cell/B.E. is a kind of hybrid system. The SPEs matches a distributed-memory system due to private local stores while the PPE is a shared-memory system. The Cell/B.E. is sometimes called heterogeneous multi-core processor with distributed memory. Because Cell/B.E. processors can be composed into bigger units such as IBM blade server with two Cell/B.E. chips they can be viewed as either 16 + 2 cores in SMP mode or two non-uniform memory access machines connected together. Programmer has then to decide which view of the Cell/B.E. processor is better for the solved problem.

Because of separation of address spaces programming of the SPE is very similar to client/server application design. Roles depends on how the work is started. In case the PPU initiates the transfers, the PPU is a client and the SPE is a server because the SPE receive data for computation and offer a service for the PPE. Another possibility is that the SPE grabs the data from the central memory. In this case the SPE is a client of central memory. This scenario is preferred because the PPE is only one and would not be able to manage all the SPUs.

3.4 Cell/B.E. programming models

Implementation of parallel algorithms rely on a parallel programming model. It is a set of software technologies such as programming languages extension, special compilers, libraries through that actual parallelism is achieved. The programming model is programmer's view to the hardware. Choosing a programming model or mixture of models that will best fit for the solved problem is another decision that programmer has to make.

For the Cell/B.E. there is variety of parallel programming models. THe models differ in view of the hardware from each other and thus how many actions are performed implicitly by the model. The actions can be e.g. task distribution management, data distribution management or synchronization. The most abstract ones can perform many actions implicitly. Their advantage is ease of implementation but at cost no performance tuning ability. Differently act the most concrete models that see the Cell/B.E. processor with all the low level details. Their advantage is performance tuning ability in all application parts but at cost of more development.

There are several models that are targeted only for the Cell/B.E. platform and are contained in the SDK. While there are other models such as MPI, OpenMP that can be used as well but they would expose only the PPE. These will not be further described.

List of the programming models (frameworks) follows in order from the most concrete to the most abstract:

1. libspe2

   This library provides the most low level functionality. It offers SPE context creating, running, scheduling or deleting. DMA primitives for data transfer, mailboxes, signal, events, and synchronization functions for PPE to SPE and SPE to SPE dialogues are also provided by this library. More information can be found in [7] within "SPE Runtime Management Library" document.

2. Data Communication and Synchronization - DaCS

   Defines a program entity for the PPE or the SPE. It is a HE (Host Element program) for the PPE and an AE (Accelerator Element program) for the SPE. It provides variety of services for that programs. The services are e.g. resource and process management where an HE manipulates its AEs or group management, for defining groups in which synchronization events like barriers can happen or message passing by using send and receive primitives. More information can be found in [7] within "DACS Programmer's Guide and API Reference" document.

3. Accelerated Library Framework - ALF

   The ALF defines an ALF-task as another entity that perform computationally intensive parts of a program. The idea is to have a program split into multiple independent pieces which are called work blocks. They are described by a computational kernel, the input and the output data. Programming with the ALF is divided into two sides. The host and the accelerator one. On the accelerator side the programmer has only to code the computational kernel, unwrap the input data, and pack the output data when the kernel finishes. The ALF offers clear separation between the host and the accelerator sides of program parts. It provides following services: work blocks queue management, load balancing between accelerators, transparent DMA transfers etc. More information can be found in [7] within ”ALF Programmer’s Guide and API Reference” document.

Choosing a framework is important decision of writing Cell/B.E. application. It should be considered enough.

3.4.1 Cell/B.E. parallelism levels

The Cell/B.E. processor offers four levels of parallel processing. That is because it is composed of heterogeneous elements, the SPE and the PPE and the possibility of composition into a more complex systems. The levels are:

1. Server level

   Parallelism on this level means task distribution among multiple servers like within a server farm. This is possible in a hybrid environment at the cluster level using MPI or some other grid computing middle-ware.

2. Cell/B.E. chips level

   On this level tasks can be divided among multiple Cell/B.E. processors. This is possible if there are more such processors in single machine. It is e.g. IBM Blade server with two Cell/B.E. chips. ALF or DaCS for hybrid can be used for task distribution.

3. SPE level

   This parallelism level allows to distribute tasks among particular SPEs. Libspe, ALF, DaCS can be used to perform the distribution.

4. SIMD instruction level

   This level can increase the speed the most. Parallelism is achieved on instruction level that means more data are processed at a time by single instruction. Language intrinsics are used for this purpose. This will be explained later in part devoted to ”SIMDation”.

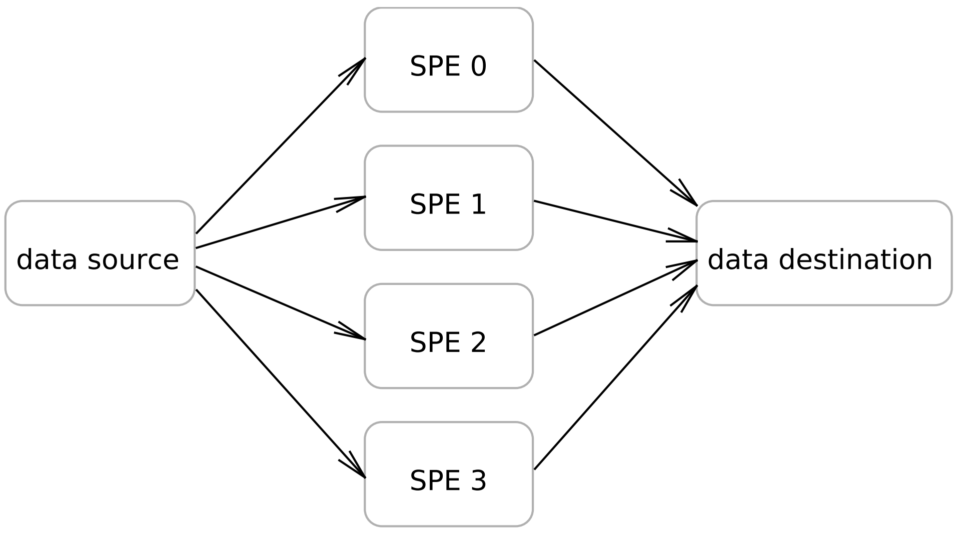


Figure 3.1: All SPE run the same code creating farm of processor that process same type of data.

3.4.2 Computation configurations

Because of the Cell/B.E.'s heterogeneous nature there are few computation configurations that can be used. Each of them differs in usage of SPEs:

1. Streaming configuration

   All SPEs serves as a stream processor (see figure 3.1). They run exactly the same code expecting the same type of data and producing also the same of data type. This configuration is well suited for streaming application for example filters where there is still the same type of data on input.

2. Pipeline configuration

   The SPEs are stages of a pipeline (see figure 3.2). Data are passed through one SPE to another. This configuration makes use of the fact that transfer among SPEs is faster than the transfer between SPE and PPE.

3. PPE centric

   This configuration is common approach to use the Cell/B.E. A program runs on the PPE (see figure 3.3) and only selected, highly computational intensive parts (hotspots) are offloaded to SPEs. This method is the easiest from a program development perspective because it limits the scope of source code changes and does not require much re-engineering at the application logic level. A disadvantage is frequent changes of SPE contexts that is quite expensive operation.

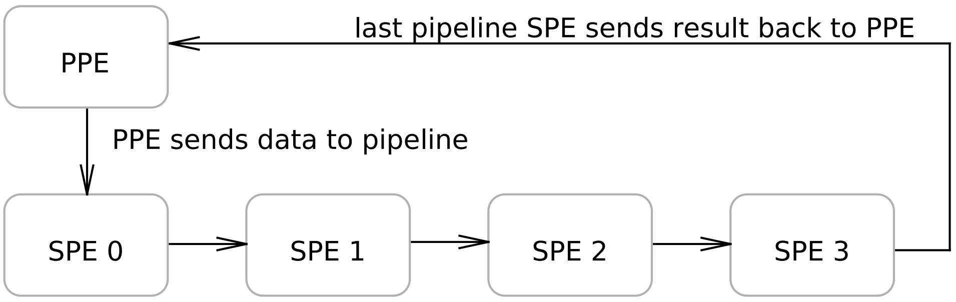

PPE
last pipeline SPE sends result back to PPE
PPE sends data to pipeline
SPE 0
SPE 1
SPE 2
SPE 3

Figure 3.2: SPE creates a pipeline. Each SPE represent one stage of that pipeline. Data are transferred only via SPE to SPE DMA transfers benefiting the speed of bus.

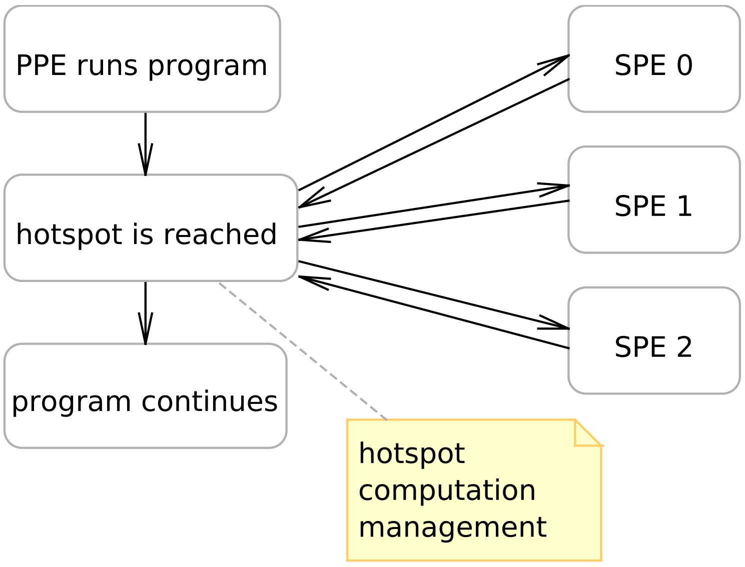


Figure 3.3: Program is run on PPE and only hotspots are offloaded to SPEs. Offloading means managing SPE context creation and loading as well as managing data transfer and synchronization between PPE and SPEs

4. SPE server

   Another configuration is to have server-like programs running on SPEs that sits and waits offering specific services. It is very similar to the PPE centric configuration. Only difference is requirement to the program to be small enough to fit into the SPU local store to avoid the frequent SPE context switching.

3.5 Building for the Cell/B.E.

Actual compilation process is performed using an appropriate tool chain. The PPE code requires the PPE tool chain and the SPE code requires the SPE one. But there is a difference between management of the code in linking stage between the PPE and the SPE object files. It is caused by difference of actual code usage. While the PPU code resides in the central memory, like in common architectures, the SPU code is loaded into the SPE dynamically and shall be somehow separated from the PPE code. It is similar to shader programs for graphic accelerators. They are also loaded into appropriate processors as soon as they are needed so they live separated.

There are two options for SPE code management. One is to build a shared library and load it explicitly when it shall be used. Another way is to build a static library and include it into the PPU executable using Cell/B.E. Embedded SPE Object Format (CESOF). This allows PPE executable objects to contain SPE executable i.e. the SPE binary is embedded within the PPE binary, see figure 3.4. The SPU program is then referenced as special external structure directly from the PPU code instead of performing shared library loading. Both ways have advantages and disadvantages which are the same as shared vs. static library usage. Shared library means better modularity and possibility of code alternation without whole executable rebuilding. On the other hand additional management of such library is necessary in contrast to a static SPE code into a PPE binary embedding.

3.5.1 Process of application porting for the Cell/B.E.

Common process of application porting for the Cell/B.E. processor (figure 3.5) consists of next two basic steps:

1. Hotspots localization

   Through profiling of the application on the PPE we find most compute intensive parts, hotspots. How to profile the application see chapter 5 of [5].

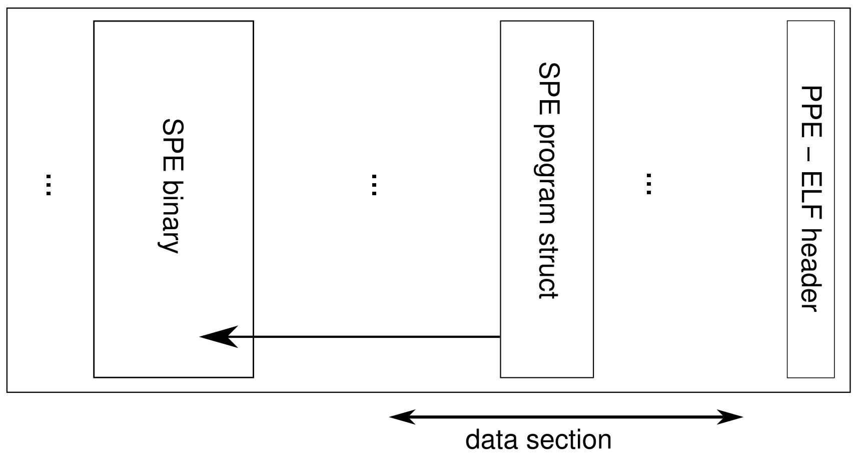


Figure 3.4: Illustration how is a SPE binary ”embedded” into a PPE binary. The SPE binary is another section of the PPE binary. It is reachable through extern struct variable, that contains a pointer to the SPE binary.

2. Hotspot porting for SPE

   Each hotspot computation is moved to the SPE i.e. the code adaptation for the SPE features shall be performed. This means DMA transfers instead of direct memory access, appropriate data structures utilization, etc. Data movement tuning e.g. different data structures usage can be then performed until satisfactory performance is obtained.

   Work distribution among available SPEs shall be performed to accelerate actual computation. Amount of work performed by particular SPEs should be equal to avoid mutual SPE waiting.

Following additional steps are necessary for application optimization and speed-up. Performing these steps leads to utilization of all the SPU features such as whole register set utilization, dual-issuing of instructions, SIMD execution and DMA transfers. More detail in [4], part 4:

1. Multi-buffering

   Data that resides within central memory and are processed by the SPE should be copied into local store before actual computation. When there are more of the places for the data (buffers) the program can take advantage from asynchronous DMA transfer and can process current buffer while the next data are being transferred into another buffer. Then the buffers are simply swapped and the SPU need not to wait until the transfer of next data is complete. See the figure in the paragraph named ”Hiding data-access latencies” in [17] for illustration.

2. Branch elimination

   Branch less instruction chain is a succession of instructions without any conditional jump. In other words there is no decision where to continue performed within such succession. Elimination of branches elongates the branch less instruction chain. In such a chain all data always go through the same instructions which makes possible to perform SIMDation. There is variety of branch elimination methods. Good information resource provides [1]. Branch elimination is probably the most complicated step due to necessity of complete code restructuralization.

3. SIMDation

   Means rewriting a scalar code into a vectorized one to be able to use SIMD instructions. In this step the most performance gain could be achieved because of multiple data processing by one instruction. Every single piece of data should go through the exactly same order of instructions in SIMDized code. Therefore is necessary to have long branch less instruction chain. The most important method is arrays of structure to structure of arrays conversion. The figure in the paragraph called ”SIMDizing” in [17] shall illustrate the data processing with SIMD instructions.

   SIMDizing brings also avoidance of usage a rotation instructions which are necessary to move unaligned data into preferred slot. Preferred slot is the beginning of a register e.g. for short integer it is the first 16 bits of the register.

4. Loop unrolling

   Loop body is the code inside curly brackets of the loop. This code is executed repeatedly until the loop condition is valid. Loop unrolling means putting more loop bodies serially into the code. This decrease loop count and elongate the loop body letting the compiler to make more optimizations. Example:

   ```
   for(uint32 i=0; i<32; i++)
   {
       printf(".");
   }
   ```

   become (by loop unrolling with factor 2)

   ```
   for(uint32 i=0; i<16; i++)
   {
       printf(".");
       printf(".");
   }
   ```

   The compiler can do more optimizations e.g. better instruction scheduling and register utilization.

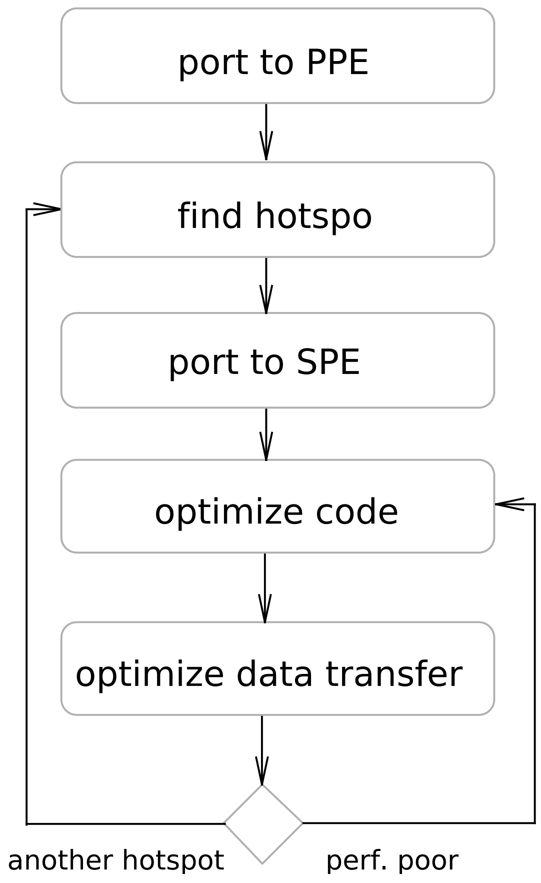


Figure 3.5: Diagram shows all stages of the process and loops for better performance tuning and other hotspots

5. Instruction scheduling

   Proper reorganization of instructions can give more performance in some cases. This step is performed by the compiler but it is possible to rearrange instructions manually in assembly language.

6. Branch hinting

   Gives a hint where the program is rather going to continue after future branch to the processor. It is done through insertion of special instructions. This step should be again accomplished by the compiler but it is possible to use appropriate assembly language instruction directly within the code.

3.5.2 SPE porting considerations

The local store size is the main SPE feature that everything spins around while porting a code to the SPE. On the one hand there are decisions about data transfers.

This means how the data that has to be processed by the SPE will be transferred into local store and vice versa. How many buffers will be used in case of multi-buffering. On the other hand is code complexity of the solved problem that influence the size of the final binary. There is one solution how to use bigger binaries than the local store, SPE overlays. It is based on division of the binary into segments that are loaded into the SPE on demand in run-time.

Programmer has to take into consideration all these things to make the final binary smaller than the local store. Everything is big trade-off between the processed data chunk sizes, number of buffers for that chunks and the code lenght.

After the first compilation of a SPU binary from original ported code the final executable will probably exceed the local store size even when the code does not seem as large. Then a big searching what part of code causes the huge size would begin. We have gone through several problems with code that is common in non SPE code but cause problems in the SPE code. Here is the list:

1. usage of keyword new

   There is no memory allocation in the SPE. So usage of the *new* keyword is meaningless. But the SPE compiler accepts it without any complain.

2. usage of std streams

   This code:

   ```
   #include <iostream>
   std::cout << "Hello" << std::endl;
   ```

   goes through the compiler without complaints but makes the final binary very big.

   The reason why the resulting code is too big is probably size of the code within headers that are included when using described features.

3.5.3 Speed and compiler options

There is variety of compiler options. Usage of them is worth nothing but can increase performance and avoid some kind of bugs.

Mike Acton explains strict aliasing in [2]. One advantage of usage of this feature is positive impact on performance. Another advantage is fact that it can avoid bugs that would appear as far as in release stage when optimizations flags are used during compilation. In this stage is really hard to track and debug this kind of bugs.

Another option advises are in [17]

3.6 Profiling

Profiling of Cell/B.E. application means rather profiling the SPE part of the application. There is variety of profiling tools. The basic one is a dynamic performance analysis which can provide many useful information such as how much time SPE stalled, reasons of the stall, the CPI (cycle per instruction) ratio, branch count, etc. The next one is a static performance analysis which can illustrate run of a SPE in instruction precision. These two analysis are evaluated from program run within full system simulator. Both the methods are well described in tutorial in the cell IDE help which is accessible through menu → Help → Help Content in the IDE.

Another profiling tools are:

1. PDT - performance debugging tool
2. OProfile
3. CPC - cell performance counter

These tools collect profiling data that can be further processed with VPA (visual performance analyser), an external tool provided by IBM. This tool can display the collected data in different charts, time lines or can highlight parts of the code that are worth to improve and many other useful features. Usage of all these performance tools is described in SDK document ”Performance Tools Reference” in [7]. We wanted to test them all but when we followed the manual instructions we experienced a few obstacles because we worked on PS3. Lately, we have found out on forums that unfortunately there is poor or none support for these performance tools on PS3.

Chapter 4

Image segmentation

Image segmentation will be described in this chapter as well as several basic segmentation techniques. Subsequently level set techniques will be introduced, defined and explained in more detail. Then level set computation issues will be described along with mentioning of two basic speed-up approaches. After that level set method relation to image segmentation will be mentioned. After all some features of the level set computation on streaming architectures will be listed along with comparison to the Cell/B.E. features.

4.1 Problem formulation

Image segmentation is process when pixels of an input image are split into several subsets, segments, based on their characteristics or computed properties, such as colour, intensity, or texture. The pixels in such segments have similar features and compose an object in the image.

In more formal way it is a function that assign a segment to a pixel:

$$S : S(p) = k \tag{4.1}$$

where $p \in$ pixels of the image and $k \in$ set of segments.

Image segmentation is used in many domains such as medicine (locating organs, tumors, bones, etc.), satellite images classification for maps (location buildings, roads, etc.), machine vision (fingerprint recognition, face, eyes, or other features recognition). Other example of image processing application can be a simple tool as the well known ”magic-stick” tool in popular graphics editing software like Photoshop.

Although there were some attempts to find general-purpose segmentation solution, results were not satisfactory. So there is not yet a general solution. Each domain needs extra approach how to perform the segmentation. Some of them are not even fully automatic so they need assistance of an operator. They are called semi-autonomous approaches. These methods need an operator who inputs some region and thus gives a hint to the algorithm. This is favourite approach in segmentation of structures in medical images like organs, tumors, vessels, etc. A physician then plays the role of the operator because of his knowledge of images' content. Some methods are autonomous but need some apriory knowledge of the segmented object properties.

4.2 Image segmentation methods overview

Image segmentation methods can be divided into following basic categories (information based on [19]):

1. Clustering

   These methods are used to partition an image into *N* clusters that cover the entire image. Two main subsets of the methods are bottom-up and top-bottom. The first one takes each pixel as separate cluster and then iterate joining these initial clusters based on some criterion until there are *N* clusters. The second one picks *N* randomly or heuristic chosen cluster centres. Then these two steps are repeated until some convergence condition is met e.g. no pixels change clusters: assign pixels to clusters based minimalization of the variance between the pixel and the cluster centre and re-compute the cluster centres by averaging all of the pixels in the cluster.

2. Histogram-based

   Firstly a histogram is computed from pixels of the image. Then peaks and valleys in the histogram creates the segments in the image. Result can be refined by recursively repeating the process. The recursion is stopped when no more new segments appear.

3. Edge detection

   These methods segment an image based on its edges. Therefore core of such methods is an edge-detection algorithm such as Canny, Sobel.

4. Region growing

   This set of methods are very similar to the flood-fill algorithm. It takes a set of seed points and a segmented image. Each seed point is something like pointer to segmented object on the image. Seed points form an initial set of segments. Then iteration through the neighbouring pixels of the segments is performed. In every step of that iteration a neighbour pixels of a segment is compared with the segment i.e. similarity function is calculated. If the pixel is considered similar enough it is added to the segment. Method is highly noise-sensitive. The initial seeds can be misplaced due to the noise. Therefore there is another algorithm that is seedless. It starts with a single pixel that is an initial region. Its location does not significantly influence the final result. Then the iteration over the neighbouring pixels are taken just as in seeded growing. If a neighbour is different enough new segment is created. A threshold value is used as similarity measurement but particular approaches differs in definition of the similarity function. While one group uses pixel's properties like intensity or colour directly another computes some statistical test from the properties and the candidate pixel is processed according the test is accepted or rejected.

5. Graph partitioning

   This approach converts an image into a graph. The pixels correspond to the vertices. There is edge between every pair of the pixels. Edges are weighted with similarity function of the two connected pixels. Then a graph algorithm that cuts off edges is run partitioning the graph resp. image. Popular algorithms of this category are the random walker, minimum mean cut, minimum spanning tree-based algorithm, normalized cut, etc.

6. Watershed transformation

   The watershed transformation considers the gradient magnitude of an image as a topographic surface. Pixels having the highest gradient magnitude intensities correspond to watershed lines, which represent the region boundaries. Water placed on any pixel enclosed by a common watershed line flows downhill to a common local intensity minimum. Pixels draining to a common minimum form a catch basin, which represents a segment.

7. Model based segmentation

   The main idea of this method is to describe the segmented object statistically, constructing a probabilistic model that explains the variation of the object shape. In segmentation phase is the model used to impose constraints as prior. Searching for such model contains steps like: registration of the training examples to a common pose, probabilistic representation of the variation of the registered samples and statistical correspondence between the model and the image.

8. Level set

   It is a method that uses a mathematical model of the segmented object. It is represented by a level set function. Segmentation is performed by deformation of an initial isoline (for 2D case), hyperplane of the level set function, with forces that are computed from the segmented image.

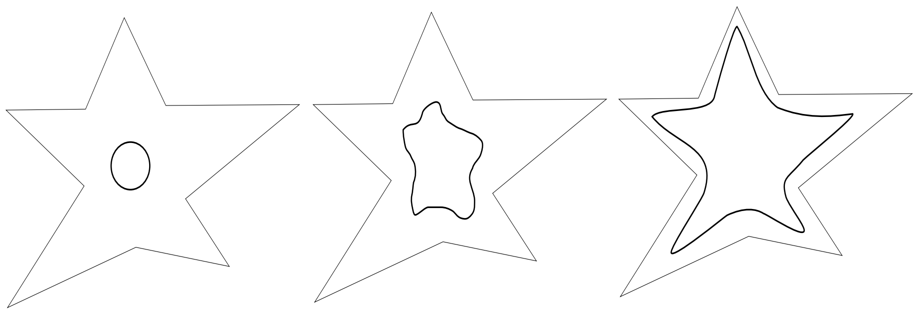


Figure 4.1: An initial shape, the circle, grows and floods the object on the background. In contrast to common flood-fill approach, level set method has several parameters that can e.g. prevent flooding beyond the object borders through small holes.

Whole process can be illustrated in very similar way to the flood-filling, see the figure 4.1. The initial isoline is deformed with forces that has direction of an isoline normal. For 2D case the initial isoline can be e.g. a simple circle as a hyperplane of a distance function from a given point. When it approaches object borders the propagation slows down. On the object borders the propagation stops because the forces are zero there.

Another illustration uses a landscape with a lake. Water is always at a constant altitude and the surface of the landscape changes in time. With the changes of the landscape the shoreline of the lake changes as well. The landscape represents the level set function and the water surface represent the isoline i.e. *k*-level set.

Advantages of the level set method are lack of special treatment of merging and splitting surfaces necessity, few intuitive parameters, ability of topology changing. The most suiting advantage for our purpose is ability of performance in all dimension without explicit changes in method because we will perform volume segmentation i.e. 3D case of level set.

4.3 Level set

Level set method as proposed by Osher and Sethian [15] provides numerical and mathematical mechanisms for surface deformation computation as time varying isovalues of level set function using partial differential equations (PDE).

4.3.1 Level set theory

Information in this and following paragraphs are based on [20] and definitions will be for 2D case. The level set function is a signed scalar distance function

$$\phi : U_{x,y} \rightarrow \mathbb{R}, \tag{4.2}$$

where $U \subset R^2$ is the domain of the function. ϕ is called embedding and is implicit representation of the segmented object. Isoline is then a subset of the level set function values, a hyperplane

$$S = \{\vec{x} \mid \phi(\vec{x}) = k\} \tag{4.3}$$

The symbol S represents a k-isoline or k-level set of ϕ. The variable k can be chosen freely, but in most cases it is zero. The isoline is then called zero isoline, zero level set or dimension insensitively front (will be used further).

Deformation of the front is then described by an evolution equation. One approach, dynamic, uses one-parameter family of ϕ function i.e. $\phi(\vec{x},t)$ changes over time, $\vec{x}$ remains on the k-level set of ϕ as it moves and k remains constant. Resulting equation is

$$\phi(\vec{x}(t),t) = k \Rightarrow \frac{\delta\phi}{\delta t} = -\Delta\phi \cdot \vec{v}. \tag{4.4}$$

Where v represents movement of a point x on the deforming front i.e. positions in time. All front movements depend on forces that are based on level set geometry which can be expressed in terms of the differential structure of ϕ. So following version of equation 4.4 link formulated:

$$\frac{\delta\phi}{\delta t} = -\Delta\phi \cdot \vec{v} = -\Delta\phi \cdot F(\vec{x}, D\phi, D^2\phi, ...), \tag{4.5}$$

where $D^n\phi$ is the set of n-order derivatives of ϕ evaluated at $\vec{x}$. The term $F(\vec{x}, D\phi, D^2\phi, ...)$ represents the force that influence the movement of a surface point. This equation can apply to every values of k i.e. every level set of function ϕ and is basic equation of level set method.

4.3.2 Level set computation

Computation of surface deformations has to be discretized which means it is performed on discretized space i.e. grid. Front propagation is then computed from initial model in cycles representing discrete time steps using this update equation:

$$\phi^{n+1}_{i,j} = \phi^n_{i,j} + \Delta t \Delta\phi^n_{i,j}, \tag{4.6}$$

where the term $\phi^n_{i,j}$ is discrete approximation of $\frac{\delta\phi}{\delta t}$ referring to the n-th time step at a discrete position i,j which has a counter part in continuous domain $\phi(x_i, y_j)$.

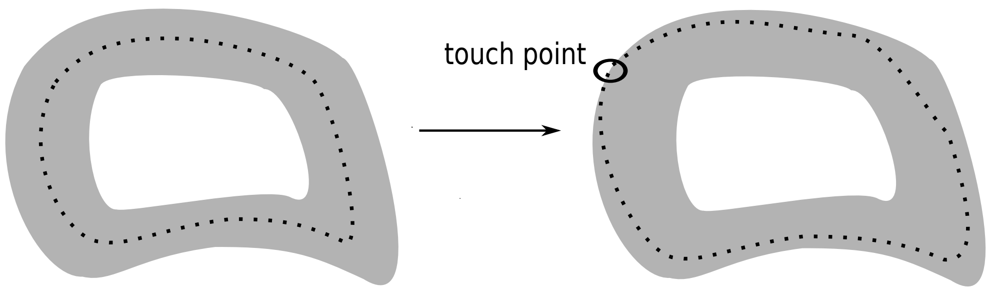


Figure 4.2: Embedding computation is performed only within narrow band (highlighted in grey). When level set touches (highlighted by the circle) the border or the band, new band has to be computed i.e. reinitialized.

$\Delta t \Delta \phi^n_{i,j}$ is a finite forward difference term representing approximation of the forces influencing the level set, the update term. The solution is then succession of steps where new solution is obtained as current solution plus update term.

Discretization of the level set solution brings two problems. Fist one is need of stable and accurate numeric scheme for solving PDEs. This is solved by the 'upwind scheme' proposed by Osher and Sethian [15]. The second one is high computational complexity caused by conversion problem one dimension higher. Straightforward implementation via d-dimensional array of values, results in both time and storage complexity of $O(n^d)$, where n is the cross sectional resolution and d is the dimension of the image. In case of pictures with size about 512^3 voxels the level set computation takes very long time.

4.3.3 Speed-up approaches

Because of computational burden of straightforward level set solving some speed-up approaches has been proposed. They are useful only when only single level set is computed which is the case of image segmentation. Then is unnecessary to compute solution for given time step over whole domain but only in those parts that are adjacent to the level set. Beside the most known and used Narrow Bands and Sparse Fields there is an octree based method proposed by Droske et al. [9].

Narrow Band, proposed by Adalsteinsson and Sethian [3], computes embedding only within narrow band, tube. Remaining points are set constant to indicate that they are not in the tube. When level set reach the border of the tube, a new tube has to be calculated based on current level set. Then new run of computations are performed on this new tube until involving level set reaches tube borders again or the computation is stopped.

Sparse Fields method, proposed by Whitaker [18], introduces a scheme in which updates of an embedding are calculated only on the level set. This means that it performs exactly the number of calculations that is needed to calculate the next position of the level set. This is the biggest advantage of the method.

Points that are adjacent to the level set are called active points and they form an active set. Because active points are adjacent to the level set, their positions must lie within certain range from the level set. Therefore the values of an embedding in active set positions must lie on certain range, the active range.

When active point value move out from the active range, it is no longer the active point and is removed from the active set. And vice versa, the point whose value comes into active range is added into active set. Along the active set there are few layers of points adjacent to the active set organized like peels of an onion, see the figure 4.3.

Process of front propagation can be imagined as a tram that lays down tracks before it and picks them up behind.

Algorithm (from [20]):
layer, L_i - set of points that are close to the level set. i is order of a layer, negative for inner layers, positive for outer ones. Zero is for the active set layer. See the figure4.3
statuslist, S_i - list of points within i-th layer that are changing status

DO WHILE (stop condition is met):

1) FOREACH (point $\in$ active set, the zero layer (ZL)
a) compute level set geometry $(\vec{x})$
b) compute change using the upwind scheme in point $(\vec{x})$

2) FOREACH (point $\in$ active set compute new embedding value $\phi^{n+1}_{i,j,k}$, which means computing 4.6.
Decide if it falls into [-$\frac{1}{2}$,$\frac{1}{2}$] interval. If $\phi^{n+1}_{i,j,k}$ moved under the interval, put the $(\vec{x})$ into lower status list, resp. into higher if $\phi^{n+1}_{i,j,k}$ moved above the interval.

3) Visit points in other layers L_i in order $i = \pm 1, \dots, \pm N$, and update the grid point values based on the values of the next inner layer $L_{i\pm 1}$ by adding resp. subtracting one unit.
If more than one $L_{i\pm 1}$ neighbour exists then use the neighbour that indicates a level curve closest to that grid point. i.e. use the point with maximal value for the outside layers resp. point with minimal value for the inside ones. If a grid point in layer L_i has no $L_{i\pm 1}$ neighbours, then it gets denoted to the next layer away from the active set, $L_{i\pm 1}$.

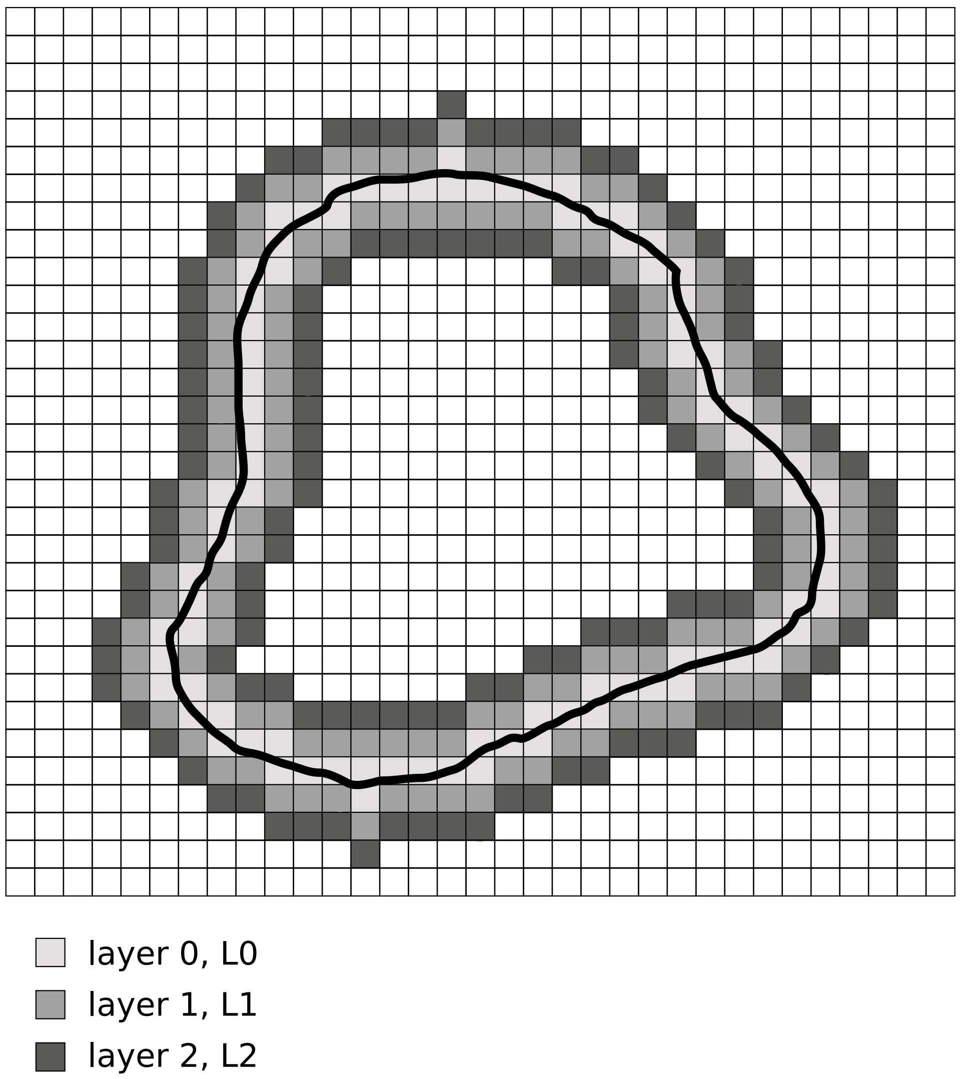


Figure 4.3: Embedding is calculated only at points that are covered by the level set (the white line). Those points (active set) are coloured in black forms the zero layer. Other layers embrace the zero layer from both inner and outer side, formed like onion peels

4) For each status list $S_{\pm 1}$, $S_{\pm 2}$, ..., $S_{\pm N}$ do the following:
a) For each element x_j on the status list S_i, remove x_j from the list $L_{i\pm 1}$ and add it to the L_i layer. Or in the case of $i = \pm(N+1)$, remove it from all layers.
b) Add all $L_{i\pm 1}$ neighbours to the $S_{\pm 1}$ list.

The stop condition is specified by maximal count of iterations. Another stopping criterion is based on a measurement of the front movement. When the front does not move anymore, calcultation is stopped before maximal count of iterations is reached.

4.3.4 Level set image segmentation

Image segmentation using a level set method is performed based on a speed function that is calculated from the input image and that encourages the model to grow into directions where the segmented object lies. There is variety of the speed functions. In this work we used speed function based on a threshold T_{low} and T_{hi} of the intensities if pixels from the input image. If a pixel has intensity value that is within the threshold interval the level set model grows, see the figure 4.4. Otherwise it contracts as fast as the pixel has value further from the interval. The function D is defined as:

$$D(\vec{x}) = \begin{cases} V(\vec{x}) - T_{low} & \text{if } V(\vec{x}) < T_{mid} \\ T_{hi} - V(\vec{x}) & \text{if } V(\vec{x}) > T_{mid} \end{cases} \tag{4.7}$$

where $V(\vec{x})$ is pixel value in point $\vec{x}$ and T_{mid} is the middle of the thresholding interval.

This is quite natural definition of what we need from the process i.e. grow as fast as possible where the segmented object lies and contract otherwise.

The update term from equation 4.6 can be rewritten into following form that consist of few terms:

$$\phi_t = \alpha|\bigtriangledown\phi|H + \beta\bigtriangledown|\bigtriangledown I|\cdot\bigtriangledown\phi + \gamma|\bigtriangledown\phi|D \tag{4.8}$$

where $|\bigtriangledown\phi|D$ represents speed function term, $\bigtriangledown|\bigtriangledown I|$ is edge term that is and $|\bigtriangledown\phi|H$ represent curvature term. α, β and γ are weights of particular terms.

Edge term is computed from second order derivatives just like Canny and Marr-Hildreth algorithms for edge detection. It shall to push level set towards edges, i.e. border of segmented object.

Curvature forces the resulting level set model to have less surface area and thus protect negative effects like leaking into unwanted places shown in the figure 4.5. Note: if $\alpha = \beta = 0$, the result is the same as flood-fill method result because there is only the speed term taking place in the calculations.

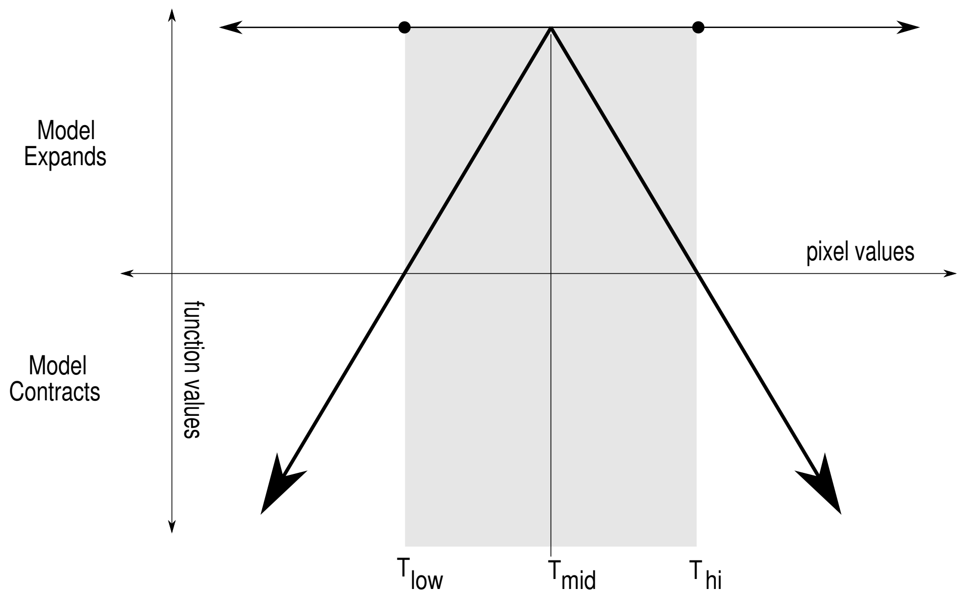


Figure 4.4: Gray rectangle encloses interval where the speed function is positive, i.e. the model expands. The fastest expansion is in the T_{mid} point

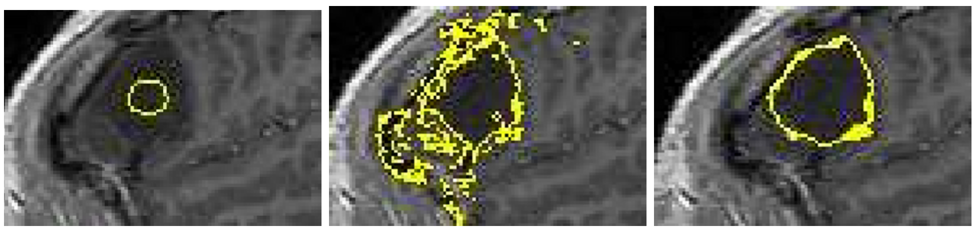


Figure 4.5: Illustration of leaking artefacts. Initial level set - circle (left). Without curvature forces, segmentation leaks into unwanted places (center). Segmentation with curvature forces (right).

We omitted the edge term so there are only two parameters in our method. Tuning of the term weights has to be performed in order to have the best results.

4.3.5 Level set methods on streaming architectures

There were some attempts for porting level set method onto special stream device. There are some obstacles due to streaming architecture that has to be overcomed to efficiently solve the problem. Firstly the streams of data must be large, contiguous blocks in order to take advantage of streaming architecture. Thus the points in discrete grid near the level-set surface must be packed into data blocks that can be further processed by streaming processors. Another difficulty is that the level set moves with each time step, and thus the packed representation must be quickly adapted.

For example Cates at al. [11] or Lefohn at al. [13] ported level set segmentation method to GPU. GPU is a streaming architecture with many, nowadays hundreds, of streaming cores. They run short programs called shaders. In porting to GPU architecture a texture memory is used to store input data in a large continuous block. Actual computation is then managed by vertices that flow into the shader and play a role of pointers to the texture memory. This is some kind of trick because the texture memory is not addressed directly by address number like in single dimension continuous address space in common processors but instead by a 2D coordinate vector. Because vertices comes as 3D points, virtual memory system that map 3D vertices to 2D texture coordinates has to be created. Such system proposed Lefonh at al. [13]. See the figure 4.6.

Another workaround has to be performed when computed data is transferred back to the CPU. This direction is much slower than the CPU to GPU direction and thus the results has to be somehow packed. Lefonh at al. [13] describes this packaging as well. There are although some advantages. One is the high count of the processors and extreme fast dedicated memory so the results can be impressive. Another is that the calculation can be directly visualized by the GPU.

Although the Cell/B.E. has some parts of the approach in common with GPU it need not to overcome the GPU obstacles. For instance no virtual memory system need to be implemented because the SPE has its own flat address space by default. Also the result packing for sending back to CPU is not necessary because transmission of data from and to SPE has the same speed and can be performed directly. All these Cell/B.E. processor features could result easier and more straightforward process of porting of level set method. But speed of the Cell/B.E. result will not probably exceed the GPU solution speed.

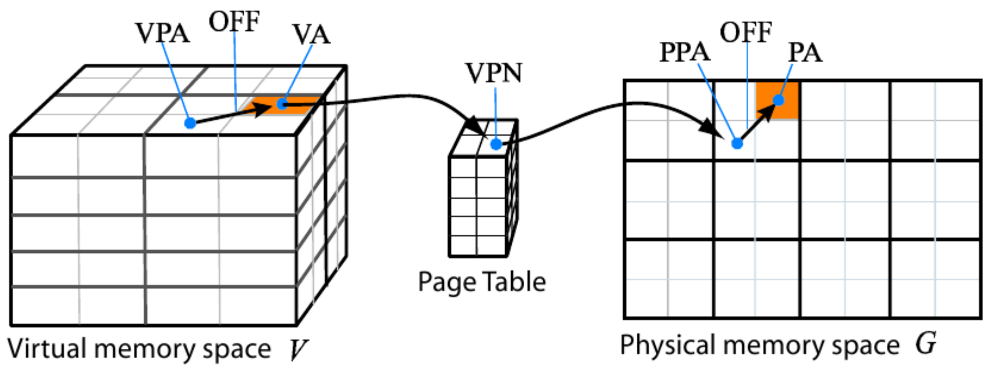


Figure 4.6: Illustration of virtual memory system (taken from [13]). 3D space level set domain (that incoming vertices come from) is mapped via page table to 2D texture coordinate system.

Chapter 5

Design and implementation

This chapter will describe details of implementation and design of our test application. It will start with listing of used frameworks continuing with description of the process of the test application incorporation into the frameworks. After that results of profiling of the application will be summarized. Followed by a new design description which was necessary due to unexpected profiling results. The rest of chapter will present actual porting process with all its problems, solutions, recommendations and all the usable information that we discovered during the porting process.

5.1 Original idea of the porting process

We wanted to follow the common scenario of porting process as described in 3.5.1. In our case this means:

1. choose base implementation
2. clean it up
3. port it to PPE
4. profile it to find hotspots
5. offload hotspots to SPEs and right away to use multi-buffering technique for DMA transfers
6. optionally try some optimization steps if the results were not satisfactory

5.2 Chosen algorithm and frameworks

We decided to choose sparse field algorithm of level set solving for porting to Cell/B.E. It is a quite complex image processing algorithm that could test the Cell/B.E. programming as a whole.

We took ITK [12] implementation of the algorithm as a base. Therefore we had to get familiar with this huge project. It contains many algorithm implementations as well as necessary infrastructure content such as loading and saving variety of formats. The base concept of this project is a pipeline and filters.

To get some work done a pipeline has to be build from filters. Filter is an entity that represents an algorithm. When a pipeline is created the last filter is started. Starting event then propagates towards the beginning of the pipeline where actual computation starts. Output from one filter is input of the following one. Filters thus create a building blocks for a more complicated method.

After several first test with examples and tutorials we wrote our own testing application (originally with code name 'pok'). It was able load an image, run a level set filter and save the results. Some reasonable parameter values were found with the pok application. It was controlled via bash scripts that is not much easy nor user friendly solution. There was also no way how to visualize the results. Therefore we decided to use another framework to overcome these problems, the MedV4D project [16].

This project was originally started as a software project and is basically framework for creation of medical applications. Its purpose is to simplify the process of GUI creation as well as actual computation model design. It let the programmer to focus only on actual problem solution. Filter is the basic building block as well in this framework. Filters can be composed into pipeline just like in ITK. But the MedV4D filters are more low-level and thus faster than ITK ones. The pipeline then offer some implicit locking of data set parts to allow parallel computation.

5.3 Incorporation into MedV4D framework

The most convenient way how to use an ITK pipeline that can be run on the Cell/B.E. seemed the client/server architecture. The part of the application that is to be run on the Cell/B.E. is a server. While client part loads initial data or saves the results, visualize the results and act as GUI with controls for parameter setting.

Whole process can be described as following: a client loads the input data sends them to a server and waits for results. As soon as the results are read back they are visualized. Then the result can be saved or sent to the server again for computation

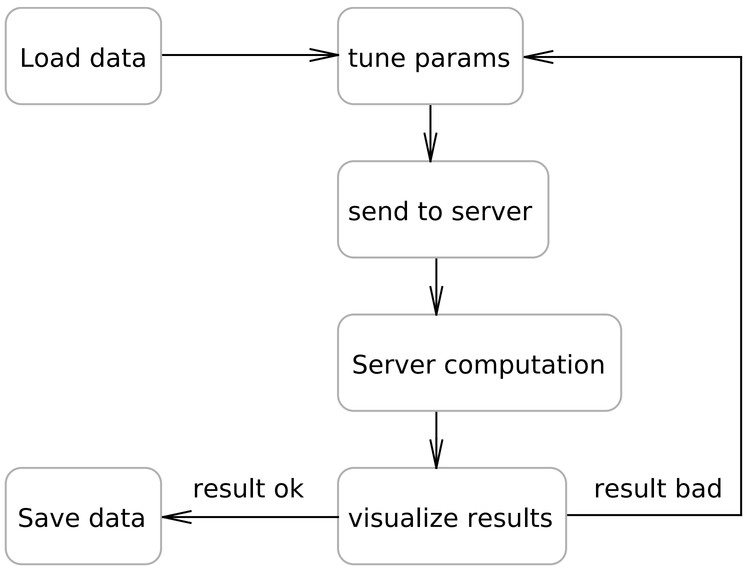


Figure 5.1: Client acts like a GUI for the server side that performs actual computation

with another parameters. See the figure 5.1 showing how the application with code name 'LevelSetClient' works.

There were two main goals which were necessary for incorporation pok application into MedV4D framework:

1. Remote computing infrastructure

   Infrastructure for sending commands to server along with data or parameter values as well receiving response messages along with resulting data had to be implemented into the MedV4D. It lead into designing whole new library of the MedV4D called remote computing (RC). On the client side there is a remote filter that encapsulates the whole infrastructure necessary for sending of a pipeline to the server as well as the result handling. The server side had to be designed completely as a whole.

2. ITK integration

   This is performed by a wrapper MedV4D filter that is connected into the MedV4D pipeline. Within this filter there are two ITK images that serves as input and output for inner ITK pipeline. Actual data of this ITK images point to data of the wrapping MedV4D filter (see figure 5.2 for details).

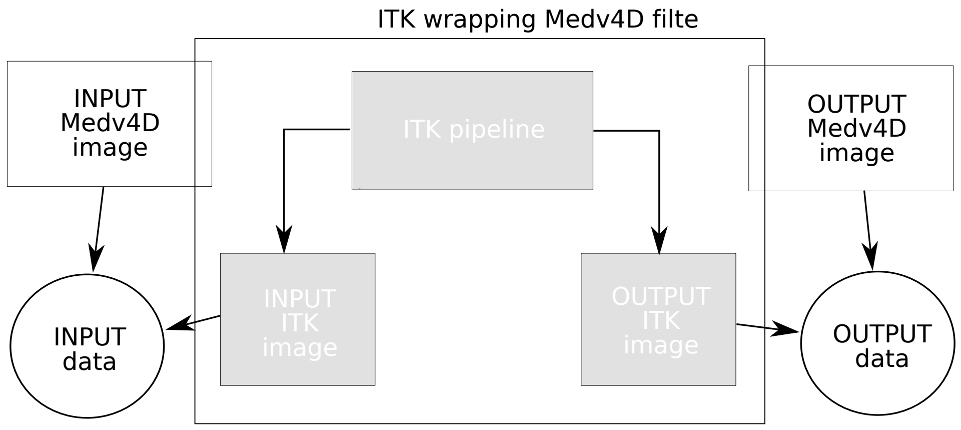


Figure 5.2: Basic elements are the two ITK images whose data are actually MedV4D images' data

5.3.1 Client part

As mentioned above the base element of client RC part is a remote filter. It implements actual command sending and result receiving functionality. It is derived from a pipeline MedV4D filter so it can be added into a pipeline and thus represent a part of the pipeline that run on a remote server. Listing of commands that the remote filter issue to the server follows:

1. CREATE

   This command is a create request. It identify the type of the filter that the remote filter represents and that should be instantiated on the server side. Server parses the command message and instantiate appropriate filter along with the whole pipeline (remote pipeline).

2. DATASET

   Tells the server to read actual data set that the computation will be performed on. The data set is parameter of the command.

3. EXEC

   This command requests actual execution of the remote pipeline. But filter parameter values should be parsed before the actual execution. These values are within the only parameter of this command. After the parsing and association of the filter parameters with the actual filter the remote pipeline is executed.

Purpose of the commands is to divide actual execution into stages and thus to define a state of remote execution. This is because it would be worthless to send actual data set to server again when user wants to execute the remote pipeline again with the same data set but only with different parameters. Commands allow this

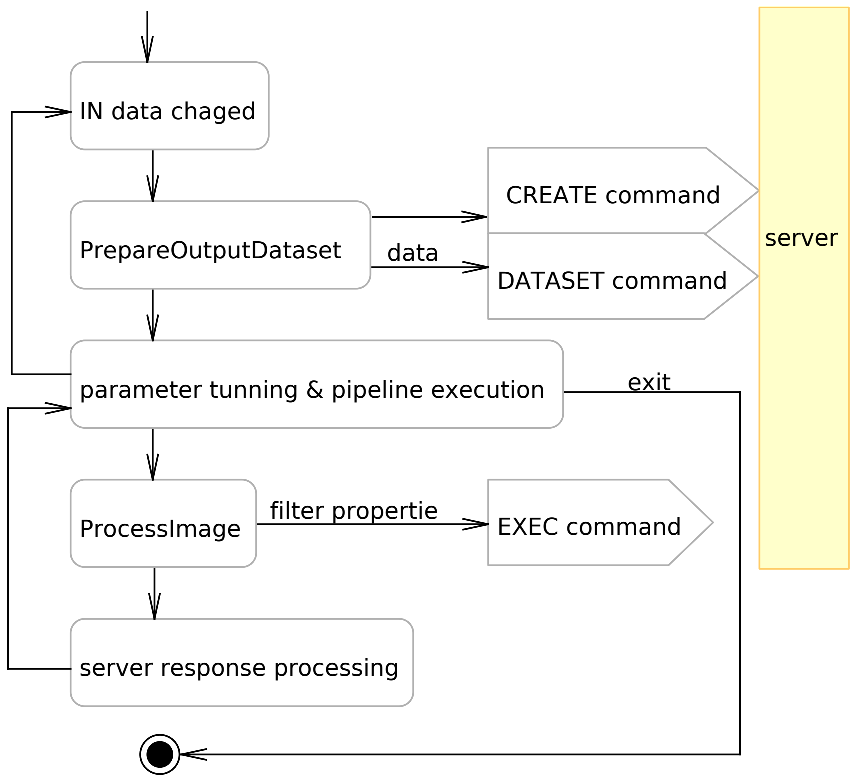


Figure 5.3: Shows three basic states of a remote filter and when particular commands are sent to a server.

because remote pipeline has a state telling 'data already received, now waiting for EXEC command as many times as wanted without no more input data transmission'.

The MedV4D pipeline filter defines also some stages that the behaviour of remote filter benefits. One of them is a method that is called only when input data changes (*PrepareOutputDataset*). This is perfect place to send DATASET command to server. Because this is called only on input data change thus DATASET command will be issued on input data change as well. CREATE command has to be sent before the DATASET command to build the remote pipeline before data set is transmitted. CREATE command is sent with DATASET command because remote pipeline has to be recreated every time a new dataset arrives.

The EXEC command is sent within a function that is called when the pipeline is executed making actual computation started (*ProcessImage*). Whole cycle shows the figure 5.3.

Server's response can be either OK or FAILED. In case of OK resulting data set is received in contrast to FAILED case when no data set is expected.

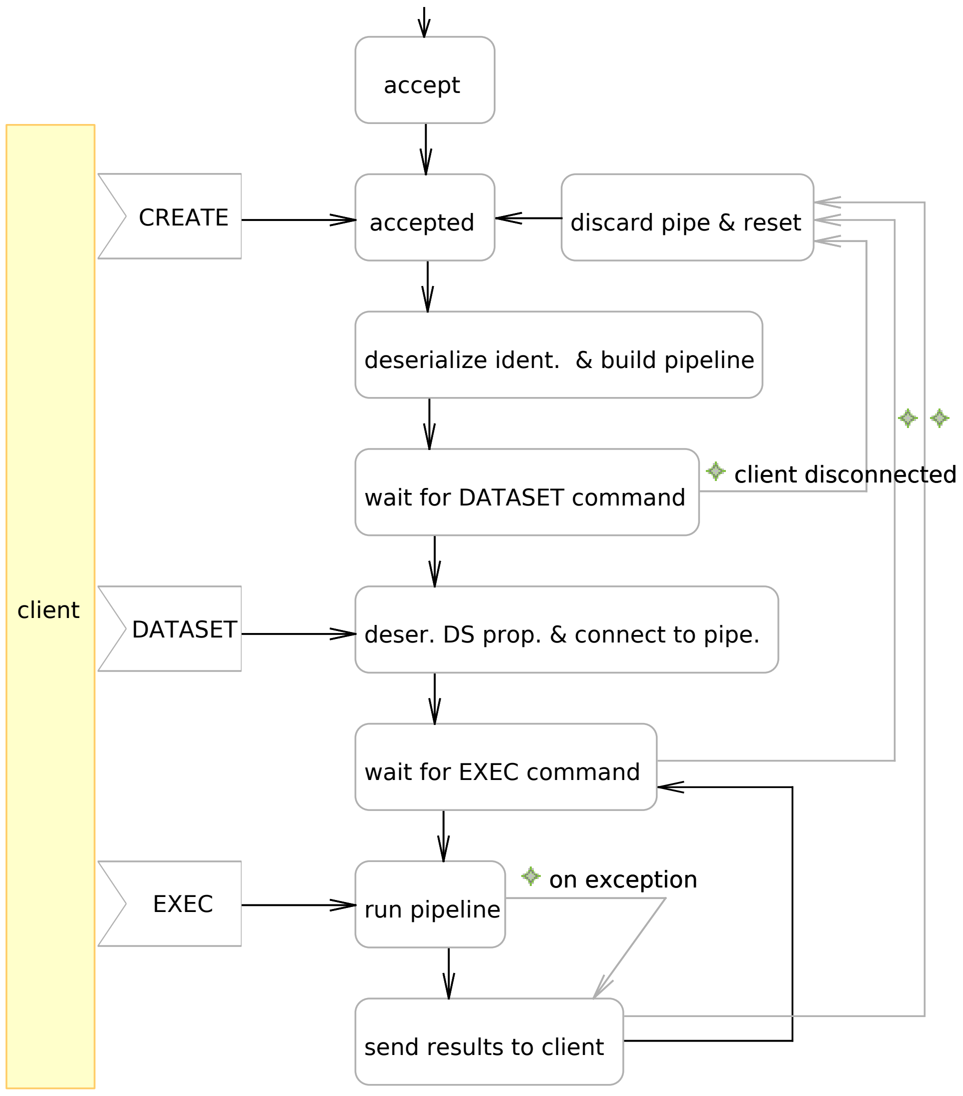


Figure 5.4: Illustration of server state diagram. The states correspond to the commands that are accepted by the server.

5.3.2 Server part

Server part is counter part of the client one so the design reflects this. Goal of server is to sit and wait for an incoming connection. One connection means one session of computation. Currently only one session at a time is held. In context of a session command from the connected client are parsed and appropriate actions performed (see figure 5.4).

Like in every client/server application some kind of stubs are needed. In our application serialization and de-serialization methods are the stubs. Goal of the methods is to ensure that the data that the client sends will be received in exactly same order and data types.

Good example is the CREATE request. In this request identifier of remote filter is sent along with filter class template parameters identifiers. In case of mismatch of that identifiers completely different class would be instantiated on the server side Hierarchy of virtual methods of data set classes defines interface for such stubs. Interface of remote filter properties class hierarchy does the same for the remote filter.

Another issue is endianess. Endianess identifier is sent along every command. On the other side is made decision if byte swapping should be performed. This allows to perform byte swapping only when it is really necessary.

Currently only one remote filter is implemented - the level set segmentation. But other filters can be easily added by appending one switch branch in remoteFilterFactory.cpp source. The level set segmentation filter is implemented as a successor of ITK filter that contains appropriate ITK pipeline. This pipeline is the most interesting part related to this work so the further content will decribe it.

5.4 Level set segmentation pipeline

This pipeline contains three ITK filters.

1. fast marching filter

   Is responsible for initial level set computation. Parameters of this filter are point $\vec{x}$ in data set and distance d. Output is data set of distances from a ball shaped object with centre in the $\vec{x}$ with radius d. This data set is the initial level set front.

2. level set segmentation filter

   Performs actual level set segmentation method. Parameters of this filter are threshold interval, maximal count of algorithm iterations, curvature and speed scaling (explained above).

3. binary thresholding filter

   Purpose of this filter is extract resulting object. It is thresholding that select pixels with values less that zero that corresponds to inner part of the resulting level set.

The fast marching and binary thresholding filter have not been changed and are used as is part of the ITK framework. The only filter that has been changed was the level set segmentation (LS) filter. This filter performs the sparse field level set solving algorithm we have chosen to port to Cell/B.E.. This algorithm uses linked lists to represent the sparse field layers. The actual algorithm, as described higher (4.3.3), is implemented in several classes. These classes form an original LS hierarchy (OLSH).

5.5 Pre-porting steps

Due to the mapping of the algorithm to Cell/B.E. and due to poor lucidity and high universality of the ITK code radical changes were necessary. We decided to rebuild appropriate part of the OLSH responsible for the sparse field level set computation. Our own LS image segmentation filter should be the result of that changes.

There are actually two class hierarchies in the OLSH. One represents the filter that performs level set algorithm, the filter hierarchy. And the other computes the FDE using the upwind scheme [15], the finite *difference function* hierarchy.

At the top of the function hierarchy there is *FiniteDifferenceFunction* that computes the upwind scheme with assistance of virtual methods that are implemented in successors. Successors are:

1. LevelSetFunction

   It provides curvature term computation methods.

2. SegmentationLevelSetFunction

   It manages speed image computation infrastructure.

3. ThresholdSegmentationLevelSetFunction

   It computes actual speed image.

The base of the filter hierarchy is *FiniteDifferenceImageFilter*. It computes the main loop of level set calculation (see step 1 in 4.3.3). Virtual methods of its successors are used to implement the appropriate sub steps.

The first successor is the *SparseFieldLevelSetImageFilter* providing implementation of algorithm's Step 1a through the *update calculation* function. Other steps are performed by the *apply update* function. Next successors *SegmentationLevelSetImageFilter* and *ThresholdSegmentationLevelSetImageFilter* only manage *difference function* in appropriate manner. The *ThresholdSegmentationLevelSetImageFilter* calculates speed function as described in 4.7. The function is computed at the beginning for the whole data set into pre-allocated image. This image is another notable amount of memory that cannot be accepted for our purpose (see paragraph B.4).

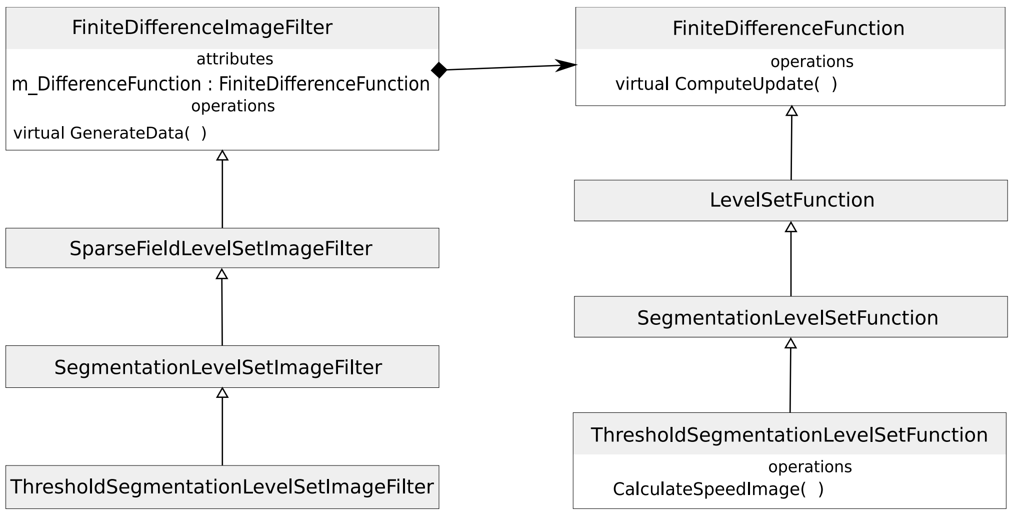

FiniteDifferenceImageFilter
attributes
m_DifferenceFunction : FiniteDifferenceFunction
operations
virtual GenerateData( )
FiniteDifferenceFunction
operations
virtual ComputeUpdate( )
LevelSetFunction
SparseFieldLevelSetImageFilter
SegmentationLevelSetFunction
SegmentationLevelSetImageFilter
ThresholdSegmentationLevelSetFunction
operations
CalculateSpeedImage( )
ThresholdSegmentationLevelSetImageFilter

Figure 5.5: Illustrates the original ITK *FiniteDifferenceFunction* hierarchy and the *FiniteDifferenceImageFilter* hierarchy and their relationship

Our approach calculates the speed function every time it is needed without any pre-calculations. This approach could be possibly better for the Cell/B.E. streaming nature.

We have simplified these two hierarchies. One reason of the simplification was the removal of the pre-calculated image. The other one was code clean-up and refactorization. Result of these changes is our own filter (*ThreshSegLevelSetFilter*, OOF). It omits all unnecessary part of the OLSH and uses reasonable parts of the original ITK level set segmentation filter (see the figure 5.6). It is also ready to be ported for the Cell/B.E.

In the function hierarchy only the base class that the resulting *ThresholdLevelSetFunc* class is derived has left. This new class does the same job as original LS function hierarchy and omitts the pre-allocation of the speed image. The computation of particular up-wind scheme terms was separated into standalone classes for more code readability and modularity.

The filter hierarchy was shortened and begins already in *SparseFieldLevelSetImageFilter*. All its successors in the original hierarchy was omitted since they did anything reasonable for our purpose. Some function implementation from the *SparseFieldLevelSetImageFilter* was borrowed into the new OOF to be ported for the Cell/B.E.

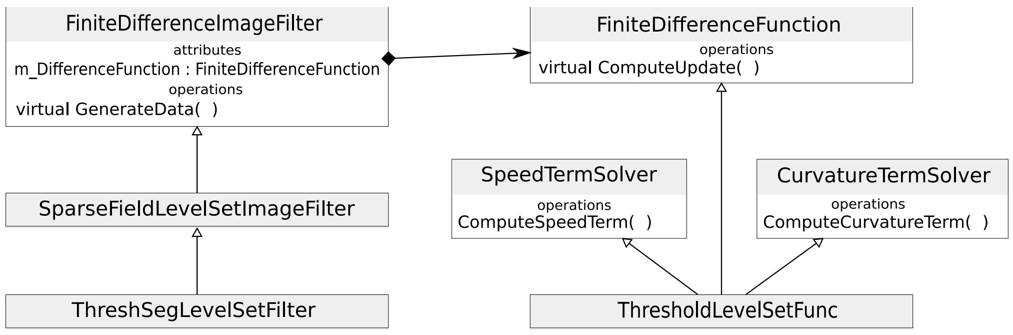


Figure 5.6: Show result of original LS hierarchy rebuilding. Some unnecessary parts was omitted to clean-up the code and to change behaviour towards a streaming architecture as well as the term computation was separated into supporting classes for modularity

Profiling results			
function name	subroutine	time spend in calculations (in seconds)	percent %
ApplyUpdate()		20.15	75.21
	PropagateAllLayerValues()	16.64	62.11
	UpdateActiveLayerValues()	2.27	8.47
CalculateChange()		6.11	22.8
	ComputeUpdate()	3.97	14.82
TOTAL		26.79	100

Table 5.1: Results of profiling showed that *ComputeUpdate* step that was originally thought to be hotspot takes only 14.82% of computation time.

5.6 Profiling

As the first step of the porting process the server application with the OOF within was build and profiled with following results:

The profiling results (see Table 5.1) show that the most time consuming part of the program is not the difference solving in update calculation step but the update application step. The original idea was to offload only the difference solving within the update calculation step which is performed on 3^3 voxel matrix and calculated independently of the others which makes this job perfectly suited for offloading to the SPE. But the time necessary for computation of this part is only the fragment of the whole. This is the reason for another changes to the OOF.

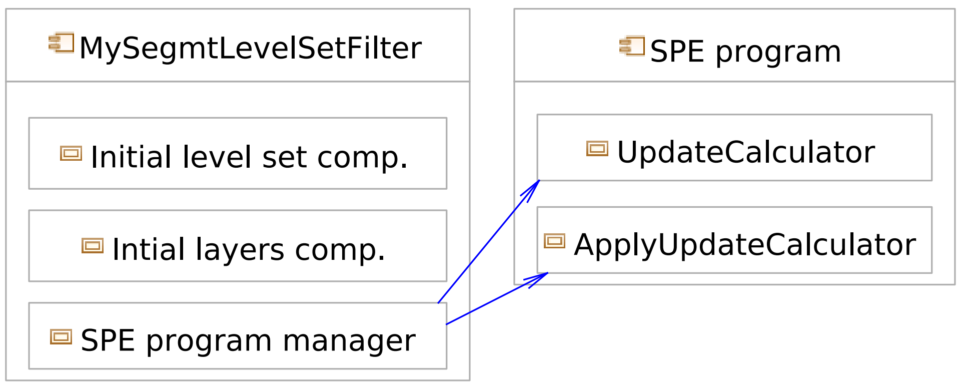


Figure 5.7: Diagram of new design components. Calculation of only the initial states is performed by the PPE. The rest is moved to the SPEs through the SPEManager that perform all necessary steps to run the SPEs.

5.7 New design

Actually the whole OOF had to be rebuild and from original ITK class hierarchy last nothing. Everything replaced by the OOF and our own version of original *FinititeDifferenceImageFilter* (FDIF) where the main loop of the algorithm as well as stopping conditions resides. The reason of replacing even the FDIF is that it suppose usage of a difference function and its virtual methods. But in the new design the difference function is offloaded to the SPE so it was taken out completely through the FDIF.

In the figure 5.7 can be noticed that almost whole original ITK pipeline is offloaded to SPE. Only initialization routines are left to the PPE. This lead to create the SPE program manager that will manage computations on SPEs. It is responsible for SPE thread initialization and run, and SPEs synchronization.

SPE part consists of two main parts. The *UpdateCalculator*, performing *update calculation* and the *ApplyUpdateCalculator*, performing *update application*. The *UpdateCalculator* traverse over *layer0* and computes update values for its points. It performs STEP 1 of sparse fields algorithm 4.3.3. The computed values are stored in a *update buffer*.

Then is the *ApplyUpdateCalculator*'s turn that performs the rest of that algorithm on the calculated update values within the *update buffer*. In context of our implementation the particular steps mean:

- STEP 2

  For every *layer0* compute new level set value and perform the test if it stays in the interval [-$\frac{1}{2}$,$\frac{1}{2}$]. If not, move the point into appropriate status list. This is performed by the *UpdateActiveLayerValues* method.

- STEP 3

  Is performed by sub-component of the *ApplyUpdateCalculator*, the *LayerValuesPropagator*. This traverses over all layers, process their values and remove nodes if they are no longer in a layer. Step processed by the *PropagateAllLayerValues*.

- STEP 4

  Traverse over status lists in innermost to outermost order and process their nodes. A node is moved to inward (outward) status list and simultaneously appropriate layer if needed. This is performed by the *ProcessStatusLists* method.

5.7.1 Data flow

In porting process is necessary to know the data flow i.e. find out what data are sent and where. What data are there produced and especially the size of all the data. This is because of decision where they will be stored. Whether in the SPE local store or in the central memory. For the first case their size has to be limited because of limitation of the local store. For the second case a communication via DMA will be necessary but the data size is not limited.

There are both cases in our application. The *ProcessStatusLists* method can be performed completely within the SPE without loading any data from the central memory. But the rest of processed data is too big and can not reside within the SPE. It has to be DMAed in chunks from the central memory. The big data are statuses, actual level set values and features that are stored within status, output and feature images. So it is necessary to load and store parts of that images. Computation is performed on small neighbourhood of voxels 3x3x3 (neighbourhood). So 27 voxels (resp. statuses) has to be transferred for one node processing. Another big data are nodes within actual layers which are linked list chains of nodes. Traversal over the chains is performed sequential by loading one node after another. For each loaded node one or more neighbourhoods shall be loaded. The computation is then performed on those neighbourhoods.

There are other data that have to be stored within central memory and that contribute to the data flow as well. Next list describe what data are processed in most important methods resp. steps within our application:

- UpdateCalculator

  It needs an array which is as long as the layer0. The size of the layer can be very big so it is impossible to store it within SPE local store. Therefore the array has to reside in central memory and its content has to be load into SPE local store buffer while the list traversal.

- *UpdateActiveLayerValues*

  It process the array from UpdateCalculator so it has to load it from the central memory. It operates on layer0. Some nodes are moved into status list and simultaneously *UNLINKed* from layer0 list. Layer resides in central memory so beside the loading nodes for traversing, some special operation has to be defined on the layers which perform the *UNLINK* action.

  Status lists are temporary objects. They live only during one *ApplyUpdateCalculator* turn. So they can reside within SPE's local store. Therefore no special operation communicating with the central memory has to be defined. But processing of the list0 has to be changed. In original ITK code the *UpdateActiveLayerValues* operates on the whole layer0. One call to this method can produce too long status list that would not fit into the local store. So iteration over layer0 has to be limited to produce limited length status lists. We have defined the limit with constant MAX_TURN_LENGHT and call processing the limited segment of the layer0 a 'turn'.

- *ProcessStatusLists*

  It works on the limited length status lists. During the lists processing some nodes are moved from one to another layer which means they have to be unlinked from one and linked into another. Linking into another layer defines another layer operation, *PUSH*. One status list is processed untill it is empty therefore all status lists remain empty and thus ready for the next *UpdateActiveLayerValues* turn after the *ProcessStatusLists* method finishes.

- *PropagateAllLayerValues*

  This method traverse over all the layers and performs moving nodes among the layers. This means operations *PUSH* and *UNLINK* as well as layer traversal and appropriate neighbourhoods loading.

There are some actions and operations defined above that need communication with the central memory. These are parts of the program where PPE - SPE communication features of the Cell/B.E. take place. Other SPE code need not to know if it is run on the SPE or the PPE. Therefore set of tools that performs the PPE - SPE communication was developed.

5.7.2 Tools

Tools are parts of the program that perform data transfer between the PPE and the SPE. All these tools perform multi-buffering to avoid waiting for data.

1. *NeighbourhoodCell*

   Represent the part of an image (3^3 voxel matrix) needed for one node computation. It uses DMA transfer list that allow data handling in scatter-gather manner. Neighbourhoods are grouped within *PreloadedNeigborhoods* container that manages which neighbourhood is being loaded, used in computation or saved i.e. actually performs multi-buffering.

2. *RemoteArrayCell*

   Represents an array stored in the main memory. This tool is used for the *UpdateCalculator* to save computed update values and in the *ApplyUpdateCalculator* to retrieve the values which the *UpdateCalculator* has stored. Its role is to perform DMA transfers not for every single value save but on a buffer of that values.

3. *LinkedChainIteratorCell*

   It traverse over a layer which is linked list data structure. As soon as one item is retrieved, transfer of the next one is immediately started before the retrieved item is processed letting the started transfer complete before the loaded item processing finishes.

PUSH and *UNLINK* operations are related to linked chains of nodes, the layers. We decided to implement them using mailboxes. Mailbox is another SPE to PPE communication channel which is able to transfer 32bit integers synchronously.

Because of the 32bit integers transfer limitation actual *PUSH* and *UNLINK* parameters have to be decoded and encoded.

1. *PUSH*

   Has to push a node into specified layer. Node coordinates and number of layer is encoded and sent over mailbox to the PPE. On the other side the PPE creates a new node appropriately according decoded coordinates and puts it into specified layer.

2. *UNLINK*

   Operation has to unlink a node from a specified list. Address of the node and the layer number are the parameters. Address is transferred in 32 bit chunks (because mailbox has 32 bit size) which are decoded on the PPE side to actual node address that is then unlinked from specified layer.

Another support tool that is worth mentioning is *ObjectStore* which is simple memory allocator templated with class of item it provides and size of an array that the items are taken from. Provides two main methods, *Borrow* and *Return*. It is used for allocation of status layer nodes. They reside in the local store and thus should be allocated on the stack.

Great advantage of the tools is also that the whole code uses only the tools for the central memory communication. Therefore debugging transfer issues means debugging the tools.

5.7.3 Work balancing

The sparse field layers are the central part that defines the amount of work to be performed. So it is necessary to balance their length among the SPEs that process them. This work is left to *Work manager*. Its goal is to ensure that all the layers are divided among SPEs uniformly.

For this purpose the *UNLINK* and *PUSH* operations implemented using mailboxes fits well. The idea behind is that actual operation on the linked list is delegated to the *Work manager*. It decides which SPE layer segment should be the node appended into.

The whole process can be compared with a company department where are several workers doing actual work and where is one manager who distributes the work among the workers.

5.8 Actual porting process

Here we will describe our experience with actual porting procedure, problems that the procedure has brought and our solutions of those problems.

5.8.1 PC as Cell/B.E. simulator

Because remote debugging program running on PS3 is quite time consuming i.e. seconds for step into command and the like and because of small amount of memory (see paragraph B.4) we decided to left actual porting to the very end of the process. Features that are needed for running on the SPE were gradually added into the original code. Some parts were rewritten e.g. the *UpdateActiveLayerValues* turn to allow some data to live in the SPE's local store. All the changes have not changed the programs' output, so one can say that the all the programs in every step were equivalent. All the debugging was performed on PC platform locally and thus quickly. The Cell/B.E. special features like DMA transfer was simulated by the memcpy function or the mailbox issues trough a simple queue.

5.8.2 Moving to PPE

It seems that moving the code to PPE is easy and there could be no problem. But we faced a problem that is worth to mention. Because our code uses a lot of third party libraries there are quite much paths to include folders. It is necessary to manage them well and not to mix architecture dependent ones.

We have mixed up include files of the boost library when cross-compiling and experienced a totally strange behaviour. We thought that we can use boost library includes that come from repository for i686 architecture. The code crashed on boost code that should be debugged and stable. For instance opening a file has crashed for an unknown reason. When program was compiled with includes from ppc(64) repository all the problems have disappeared.

5.8.3 Tools porting

Next step was to port the tools for SPE. In fact is to rewrite usage of memcpy that simulate the DMA transfer to use real DMA transfer.

Data that are transferred through DMA (DMAed) within the Cell/B.E. should meet size and align conditions (see [5], Chapter 4). Data that does not meet this condition will generate a BUS error. This condition force that all data that are being DMAed should be allocated to aligned addresses (see objectStore.h or updateVal-sAllocator.h).

Debugging within this step have been performed already on a Cell/B.E. machine. We used both PS3 and systemsim. Systemsim can detect which DMA transfer brakes align rules and thus cause the BUS error but it is really slow. Instead using systemsim we have implemented a DMAGate (see DMAGate.h) that all DMA transfer go through and where are all the conditions checked. Such central point for all DMA transfer is really important part of Cell/B.E. program because it gathers all DMA stuff into one place making debugging much more easier.

5.8.4 Memory checking tools

Gradual code porting for SPE was really time consuming due to fact that tools operates with a stack memory. Debugging such parts needs extra care for what is where rewritten. Since a stack memory is used some part of call stack would become corrupted and the program becomes undefined. The worst thing is that it can continue without crash or to crash on totally different place. Errors of this type are always hard to track. There is need of usage of some memory checking tools. For us memcheck tool, part of Valgrind, proved to be useful to detect stack corruptions.

Usage of DMA transfers (or memcpy) is also dangerous. It rewrites destination memory without checking what is within that memory. Then many errors causing segmentation faults arise.

We have spent lots of time debugging plenty of such errors. Even in resulting application can occur some errors of this kind. Debugging every single this kind error is a never ending story. So we recommend either to develop additional tools that will check such errors or to use memory checking software such as Valgrind [8]. We used the Valgrind but not from the beginning. So when we check the program there was too much warnings of the same kind. Solving them was impossible in that time.

Checking the program with memory checking tools is necessary already from the very beginning of the development and quite often. It is advisable not to make huge code changes not only due to memory checking but due to actual process as such. It is better to develop gradually in a small steps. One can then find an error more simply. This is universal rule for programming but we believe it is valid specially for the Cell/B.E. porting process.

Chapter 6

Results

In this chapter speed measurements of our application will be presented as well as a few pictures of the results. Because we have not accelerated the computation over traditional processors a discussion of the reasons will follow. We will mention possible changes in design to speed up the execution. Then some consideration how shall the algorithm that is well suited for the Cell/B.E. look like. Chapter will be finished with comparison of complexity of programming for the Cell/B.E. and conventional processors.

6.1 Speed measurements

Actual speed measurement is performed within *GenerateData* method of the *FiniteDifferenceFilter*. Counter is started right after initialization phase i.e. before main algorithm loop and stopped right after the loop. The program memory usage was tuned to use only really necessary amount of memory within the measured interval. Valgrind's massive tool was used for the memory usage tuning. This was necessary because of the PS3 limited memory, see paragraph B.4

Beside the server memory usage tuning other changes was made also within the client part. There has been a special filter developed. The filter shrinks a data set to a given size and cast its voxels to float. The shrinking is performed by a linear interpolator. The purpose of the float casting is avoidance of allocation of additional memory on the server side that would be necessary for float data set.

It is strange that the command *top* shows far bigger memory usage than the Valgrind's massif. We have not cared much about it but the idea is that the *top* shows all memory requested from the system while the massif shows exact memory used by the process.

Measurement results						
data set	size	seed	init. dist-ance	max. itera-tions	arch.	time spend (sec)
3slices (skull 1)	512x512x3	[256,256,1]	40	800	Cell/B.E. i686 .	16.48 1.89
skull 1	256x256x80	[128,110,20]	20	500	Cell/B.E. i686	471.89 95.23
skull 1	256x256x80	[128,110,20]	4	500	Cell/B.E. i686	325.93 61.74
skull 2	256x256x80	[128,110,20]	4	500	Cell/B.E. i686	319.47 55.88
skull 2	256x256x80	[128,128,40]	4	500	Cell/B.E. i686	366.63 76.9

Table 6.1: Results of speed measurement. The first difference is a bit big probably because of the small data set and insufficient time to take advantage of parallelism of six SPE. The second measurement shows that the PC implementation is about five times faster. The rest of measurements, performed on different data set and parameter configurations prove the coefficient five.

Results of the speed measurement is summarized in the table 6.1. Every measurement was run with the same curvature and speed-scaling factors. But with different initial distance, maximum iterations and seed parameters. These parameters were set according to data set size.

Three different data sets were measured. All of them was CT images of a skull that were scanned for anthropological purposes. Measurement of some volumetric parameters of such images is a valuable source of data for the anthropologist. Method implemented in our application could be used in the praxis for such purposes if the application is quick enough. Therefore speed-up of the current methods is necessary.

6.2 Reasons of slowdown and possible improvements

Porting the code to run on SPEs and distribution of the calculations among the available SPEs is not sufficient to get more speed from the Cell/B.E. over traditional processors. The optional speed-up porting phases are necessary to be performed. But our program has another speed pitfalls.

The biggest problem is the CellNeighbourhood that represent a small part of an image (the 3^3 voxel matrix). It is transferred for every layer item. In some parts

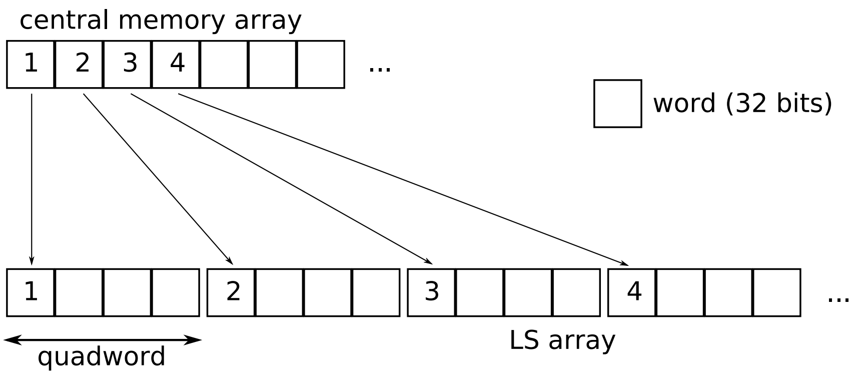


Figure 6.1: Illustration of transfer of data that are smaller than quad-word. Hardware automatically increases address within the local store buffer in such way that every transferred item is quad-word aligned.

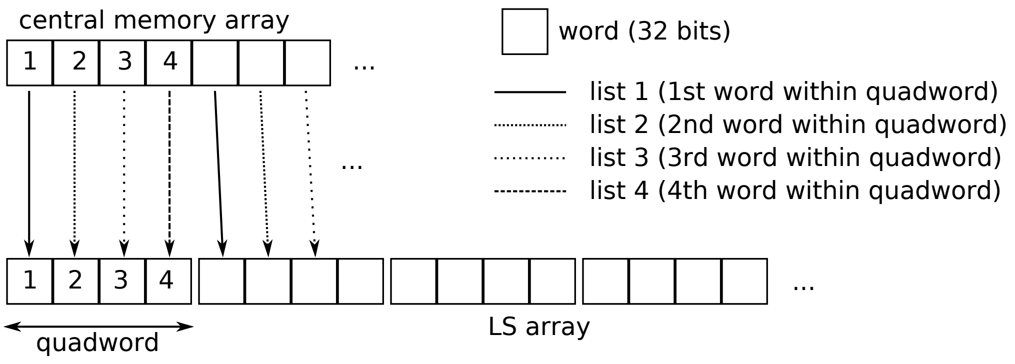

central memory array
1
2
3
4
...
word (32 bits)
list 1 (1st word within quadword)
list 2 (2nd word within quadword)
list 3 (3rd word within quadword)
list 4 (4th word within quadword)
...
1
2
3
4
...
quadword
LS array

Figure 6.2: Workaround for transfer for smaller than quad-word chunks. It uses more than one DMA list. One per quad-word align e.g. for chars 16 DMA lists are needed.

the both output and status image neighbourhoods are transferred. We wanted to perform the transfer in scatter-gather manner through DMA lists but we have faced some problems. The DMA transfers and specially using DMA lists are designed for big amount of aligned data. When they are used for small amounts (smaller than 16bytes per list item) performance goes down because of unaligned data transfer. When smaller than 16 bytes (quad-word) data are being transferred every single item is automatically aligned to quad-word address within the local store buffer (see figure 6.1).

This increase required size of a buffer that is needed for the transfer. This can be partially solved by usage of multiple DMA lists (one for each quadword align). This is illustrated in the figure 6.2. For details see [10].

We have adopted this workaround and used it within the neighbourhood transfer. Because of the automatic local store quadword address aligning we had to use a translation table. This table maps position of actual neighbourhood members into

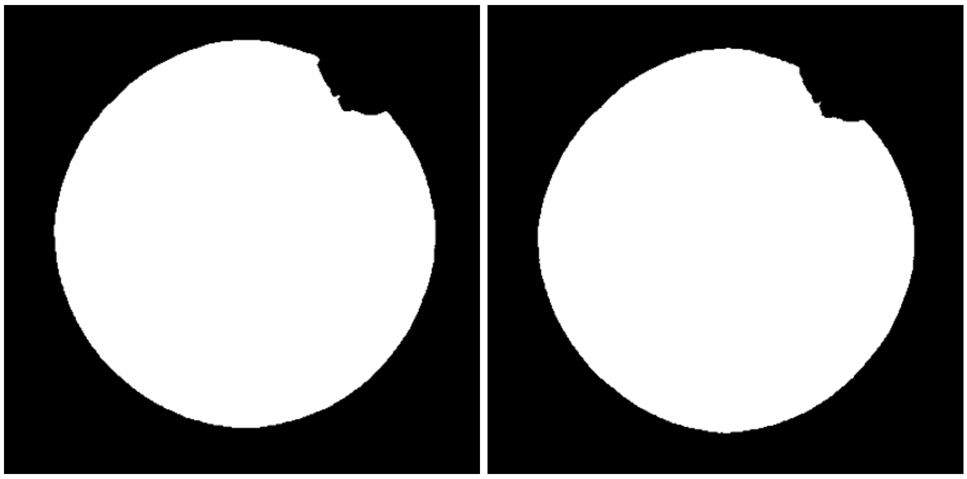


Figure 6.3: Comparison of one slice segmentation on different architectures. The left was computed on PC (i686), the right on Cell/B.E. Although the segmentation was run with the same parameters there is small difference between those two images. The Cell/B.E. level set has not reached as far as the PC one. It is because of duplicate nodes within layers.

position within the local store buffer. This is working solution but overhead is incredible and thus the solution is useless (see cellNeigbourhood.tcc for details).

Another problem is order of layer nodes resp. neighbourhoods processing. When there are two nodes within one layer that are next to each other processed subsequently changes made to the first processed neighbourhood would not appear to already preloaded next one. Therefore another merging was necessary to be performed. This coerced another changes to neighbourhood transfer and made the actual transfer unbearably expensive operation.

Another pitfall is the situation when sibling nodes are processed by two different SPEs. Adding nodes to layers is based on information from neighbours of the currently processed node. So it is possible that two different SPE inserts the same nodes subsequently into one layer because they process sibling nodes i.e. overlapping neighbourhoods. This makes no changes to output but means additional overhead due to processing multiplied nodes. Solution to this problem would require synchronization among the SPEs.

Additional improvements of neighbourhood transfers should be done to speed up the execution. This corresponds to data transfer optimization step of porting process (see paragraph 3.5). In our case this would mean radical simplification of neighbourhood transfer. This transfer should take only a few instructions to be effective. In the SDK examples transfers are performed by simple macros which is probably the most effective way. In our case simplification if the neighbourhood transfer would mean utilization of bigger image part transport within a single DMA transfer to avoid automatic local store alignment. This would mean a bigger local store array to store the image part. But processing of nodes that are associated with the part would be somehow gathered and thus the count of transfers would be lower. Processing of nodes should take into account their spatial information when inserted into a layer and thus to gather the node processing on the bigger neighbourhood. But such computation scheme would coerce complete redesign of the application and begining from scratch.

6.3 Code and design complexity

The Cell/B.E. programming means mainly programming for the SPEs because of their performance and count. Because of indirect memory access and need of usage of some multi-buffering memory transfer scheme the design is quite more complex over common processor. With another limitation which is the local store size is design of a Cell/B.E. application a challenge.

The Cell/B.E. is designed for intensive computation applications. For a programmer this means utilization of all SPEs and all their features at the maximum level. It is only possible for certain class of algorithms. Let's call them streaming parallel algorithm. But what is it? What is the definition? Work on our application has showed us what the streaming parallel algorithm is not. Therefore by negation of the features that slow down our program we could get a definition of the streaming parallel algorithm. It would look like the following:

1. Streaming nature

   Data of that program are uniform and can be processed in a small pieces (chunks) which are mutually independent and which can fit in the local store memory. This means that for a chunk processing only this chunk is necessary and not any part of the other chunks. When this chunk is once processed it is stored and is never retrieved for processing again.

   For our application this is not true. At least not for all the input data. Processing of a sparse field layer meets the streaming nature. But processing of parts of underlying images (neighbourhoods) associated with these nodes does not. The neighbourhoods are mutually dependent see 6.2.

2. Paralel nature

   Input data can be divided into parts which are again mutually independent and which can be sent to particular cores for processing. This avoids any mutual synchronizations among the SPEs.

   For our application is again not true. Work that is divided among the cores is not independent because of dependency of particular neighbourhoods. See 6.2 what causes problems.

When algorithm does not meet the streaming parallel definition then it should not be implemented or it must be changed. This means e.g. to use different data structures or to change the way the current data structures are used. In our case, change of the data structure to somehow gather node processing to specific image part would mean change of the algorithm we implemented. But then it would not be the original algorithm any more.

In contrast there are algorithms that fits the streaming parallel algorithm definition. The examples in the SDK that meet the definition. Except them e.g. thresholding meets the definition as well. It operates on an image that has uniform data - pixels. It apply simple condition on each pixel which result depends only on the processed pixel. Processing can be divided into chunks. These three features meets the streaming condition. Moreover the data can be divided into independent sets that can be processed by multiple cores (the parallel nature). Implementation of this algorithm can therefore result huge performance gain on the Cell/B.E. than conventional processors.

Our work has showed importance of the initial consideration and the design phase. When there are more algorithms that are solving a desirable problem programmer should think carefully which one will be the best for porting to the Cell/B.E. First stage of the initial consideration should be a model of the application implementing chosen algorithm. Consideration what kind of data are processed. If they are uniform. If they are divisible into chunks. If the computing can be divided into independent parts i.e. what entity defines amount of work to be done. These are questions that lead to answer if the chosen algorithm is or is not the streaming parallel. If the answer is no, implementation of that algorithm is rather worthless and will result such a suboptimal program as the ours.

Another thing is the code complexity. Performing the optimisation porting cycle steps utilise all the Cell/B.E. features and thus leads to gain more performance. Usage of language intrinsics, different kind of special instruction of macros for variety of purposes, multi-buffering, etc. increases actual code complexity whereas decreases code readability. It is also time consuming and hard to perform.

There is still another question to be answered. Is the porting of an algorithm worth at all? Since only some special machines are equipped with the Cell/B.E. actual data have to be sent to the machine for computing and results have to be sent back. Therefore only complex algorithms where performance gain would be bigger than the time spent in transfer of actual data set are worth to port. The algorithm we have tried to implement is complex enough to be offloaded to compute on remote machine while the mentioned thresholding would not. As soon as common machines such as notebooks, desktops are equipped with the Cell/B.E. processor even a class of such simple image processing algorithms that is implemented in everyday-use software such as the thresholding or variety of masking or edge detections is worth to port.

Chapter 7

Conclusion

At the beginning we studied available literature to find out what is actually the Cell/B.E. and what benefits it brings. What special features it has and what they are good for. Then we have been trying to install SDK to start actual development process. During this phase we faced some obstacles such as bugs, incompatibilities among tools, the libraries that the tools use and even the operating system vs. SDK incompatibilities. Therefore we had to go through variety of forums and other sources to find the solutions. As a side effect we improved our Linux knowledge. Eventually we managed to install SDK and was able to start developing. Then we have been testing variety libraries, tools and examples to get familiar with the Cell/B.E. development.

We have chosen sparse field algorithm implementing level set based segmentation to port to Cell/B.E. platform. This is quite complex algorithm to test the platform's potential. We adopted ITK implementation of that algorithm. Therefore we had to study ITK tool kit and its internals. We have also incorporated the whole program into MedV4D framework. That means we have implemented new modules that allow using ITK and can offload some part of processing to remote machine.

Actual porting process started with cleenup the ITK code. Then profiling of existing application took place finding out hot spots of the code which can be in turn offloaded to SPE to take advantage of Cell/B.E.. But profiling results was quite unexpected so another redesign of application followed. In this new design almost whole original ITK code was offloaded to the SPE. A big code restructuring was necessary to allow us to perform actual computations on the SPE. We have repeatedly debugged same kind of errors due to corruptions of stack memory caused by DMA transfers. This has proved importance of usage memory checking tools. Finally we have been able to run the whole algorithm on SPE and to measure time need for computations.

The result of measurement showed that simple move the computation to the

SPEs is not sufficient because it does not utilize all the Cell/B.E. features. We have identified some bottlenecks of the application and discussed possible solutions. Implementation of these solutions would require whole application redesign using another data structure. These changes would actually mean change of the used algorithm.

Because of complexity of the chosen algorithm and the base implementation we spent so much time with uninterresting work and in many cases we have done the porting process in wrong way. This lead to silly mistakes and another time wastings in debugging bugs. But on the other hand the improvization was the reason of appearing of interresting ideas like the PC as the Cell/B.E. simulator.

Our work has proven that taking an implementaion of an algorithm and gradually change it to let it run on the SPE is not a good way of proting process. Instead it is necessary to think if the algorithm is well suited for porting. And if it is then implement it from scratch considering all the Cell/B.E. features utilization already from beginning. The gradually changing the original implementation with the idea of utilisation all the Cell/B.E. features as soon as it is able to run at least on the SPE is wrong. Then it could meand the whole application redesign like in our case.

We have also discussed differences between the programming for conventional processors and Cell/B.E. As well as question of actual algorithm complexity and worthiness of porting them to Cell/B.E.

The Cell/B.E. platform is very interesting for its variations of use scenarios and ability of program tuning and customization. We think the pallet of tools and features of the Cell/B.E. can make it interesting alternative to conventional processors whose lifetime is getting shorter due to limitations in manufacturing process. The Cell/B.E.'s great potential has already been proven but it is still waiting for wider spectrum of programmers.

If the process of the Cell/B.E. development starting became simpler we believe much more new programmers would start using and programming it. Nowadays there are plenty of information about the Cell/B.E. but they are somehow unsorted or out of date. The best information source are documents shipped along with the SDK. But they are targeted to contain all the information regardless the level of experience of the reader. That means when a programmer wants to start developing applications on the Cell/B.E. he/she would go trough a plenty of that information before he/she can start actual work. It is a pity there is total lack of information for PS3 users within SDK documentation. This is quite problem when a major part of beginners has a PS3 available. There is simply lack of some 'cookbook for beginners' with practical information and some howtos. We believe this work could be such a cookbook with such practical information that potentially may help to some other programmers who would like to start developing for the Cell/B.E.

Bibliography

[1] Mike Acton. Cell performance website. www.cellperformance.com, 2009. [Online; accessed 15-May-2009].

[2] Mike Acton. Understanding strict aliasing article. http://www.cellperformance.com/mike_acton/2006/06/understanding_strict_aliasing.html, 2009. [Online; accessed 22-May-2009].

[3] David Adalsteinsson and James A. Sethian. A fast level set method for propagating interfaces. *J. Comput. Phys.*, 118(2):269–277, 1995.

[4] Jonathan Bartlett. Programming high-performance applications on the cell be processor series. http://www.ibm.com/developerworks/views/power/libraryview.jsp?search_by=Programming+high-performance+applications&Submit.x=47&Submit.y=17&url=%2Fdeveloperworks%2Fviews%2Fpower%2Flibrary.jsp, 2009. [Online; accessed 22-April-2009].

[5] IBM corp. Programmer's guide. /opt/cell/sdk/docs/index.html.

[6] IBM corp. Cell Broadband Engine resource center website. www.ibm.com/developerworks/power/cell, 2008. [Online; accessed 12-December-2008].

[7] IBM corp. SDK documentation. www.ibm.com/developerworks/power, 2009. [Online; accessed 14-February-2009].

[8] Valgrind developer team. Valgrind memory check tools documentation. valgrind.org, 2009. [Online; accessed 20-March-2009].

[9] Marc Droske, Bernhard Meyer, Martin Rumpf, and Carlo Schaller. An adaptive level set method for medical image segmentation. In *IPMI '01: Proceedings of the 17th International Conference on Information Processing in Medical Imaging*, pages 416–422, London, UK, 2001. Springer-Verlag.

[10] Cell/B.E. forum. DMA list issues thread. http://www.ibm.com/developerworks/forums/thread.jspa?threadID=143212, 2009. [Online; accessed 24-April-2009].

[11] Ross T. Whitaker Joshua E. Cates, Aaron E. Lefohn. Gist: An interactive, GPU-Based level set segmentation tool for 3D medical images. *School of Computing University of Utah Salt Lake City, UT 84112 USA*, 2004.

[12] Kitware. ITK website. `www.itk.org`, 2009. [Online; accessed 4-January-2009].

[13] Aaron E. Lefohn, Joe M. Kniss, Charles D. Hansen, and Ross T. Whitaker. A streaming narrow-band algorithm: interactive computation and visualization of level sets. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 243, New York, NY, USA, 2005. ACM.

[14] Fedora maintainer team. Fedora project site. `http://fedoraproject.org/`, 2008. [Online; accessed 22-December-2008].

[15] Stanley Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on hamilton-jacobi formulations. *J. Comput. Phys.*, 79(1):12–49, 1988.

[16] CGG team. project MedV4D. `http://cgg.mff.cuni.cz/trac/medv4d`, 2009. [Online; accessed 15-January-2009].

[17] OpenCV team. Optimization strategy article. `http://cell.fixstars.com/opencv/index.php/Optimization_Strategy`, 2009. [Online; accessed 21-May-2009].

[18] Ross T. Whitaker. A level-set approach to 3D reconstruction from range data. *Int. J. Comput. Vision*, 29(3):203–231, 1998.

[19] Wikipedia. Segmentation (image processing) — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Segmentation_(image_processing)`, 2009. [Online; accessed 22-April-2009].

[20] Terry S. Yoo. *Insight into Images: Principles and Practice for Segmentation, Registration, and Image Analysis*. AK Peters Ltd, 2004.

Appendix A

Images

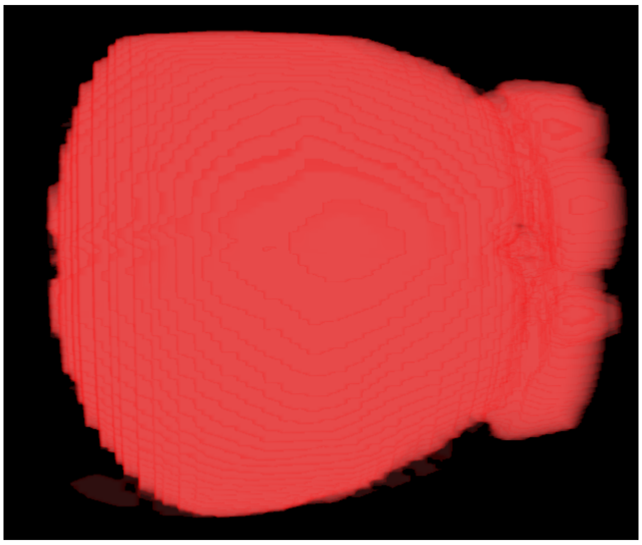


Figure A.1: 3D View of the segmented data set. VTK viewer used to visualize the data set. On the left part there is the parietal part of the skull. This part is quite well segmented as illustrate the figure A.2 The segmentation of the front skull part (on the right side) is poor due to leaking through the eye sockets (see the figure A.4).

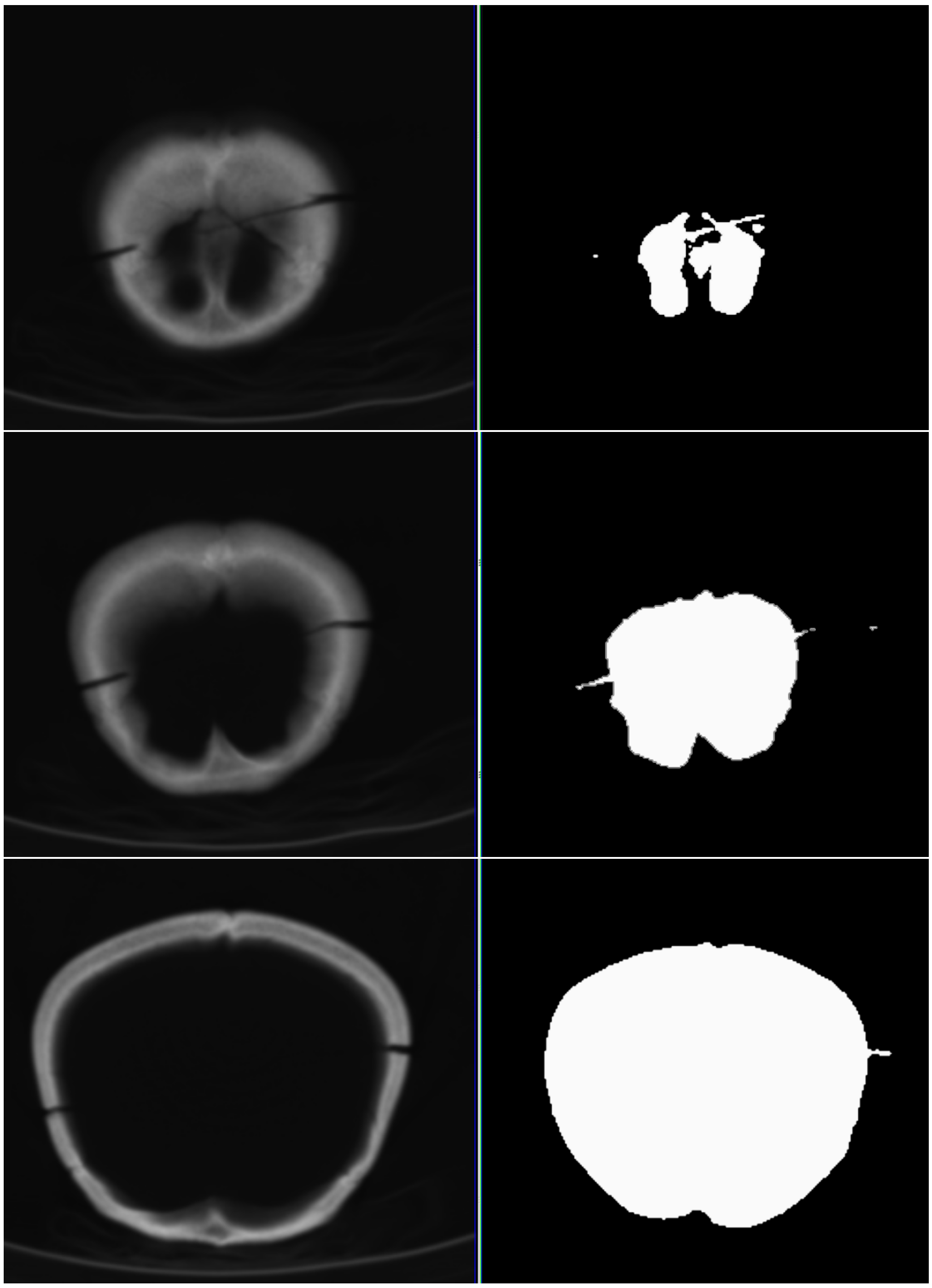


Figure A.2: Here is an illustration of the parental part of a skull segmentation. Cavities that should be segmented are small enough to let the level set reach all the edges. The segmentation has not completely leaked through the holes on both sides due to the curvature term. But if the holes were wider there would be leakings similar to the ones showed on the following figures.

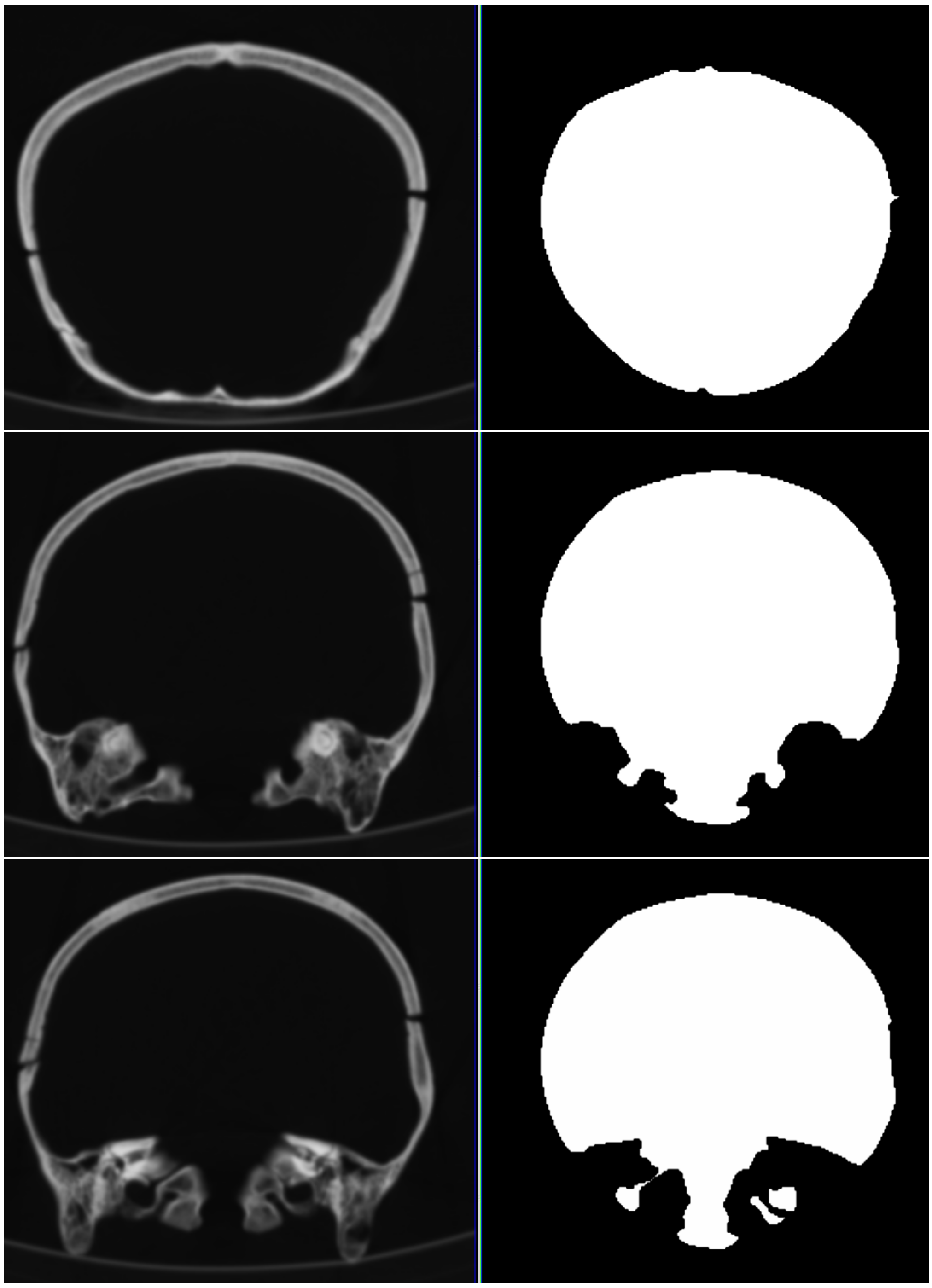


Figure A.3: This image set shows middle slices of the skull data set segmentation. Some edges are reached by the level set (on the right side of the skull mostly) while some are not. This is caused by insufficient count of algorithm iterations (500) or improper initial level set placement. It proves that result is highly initial parameters setting sensitive.

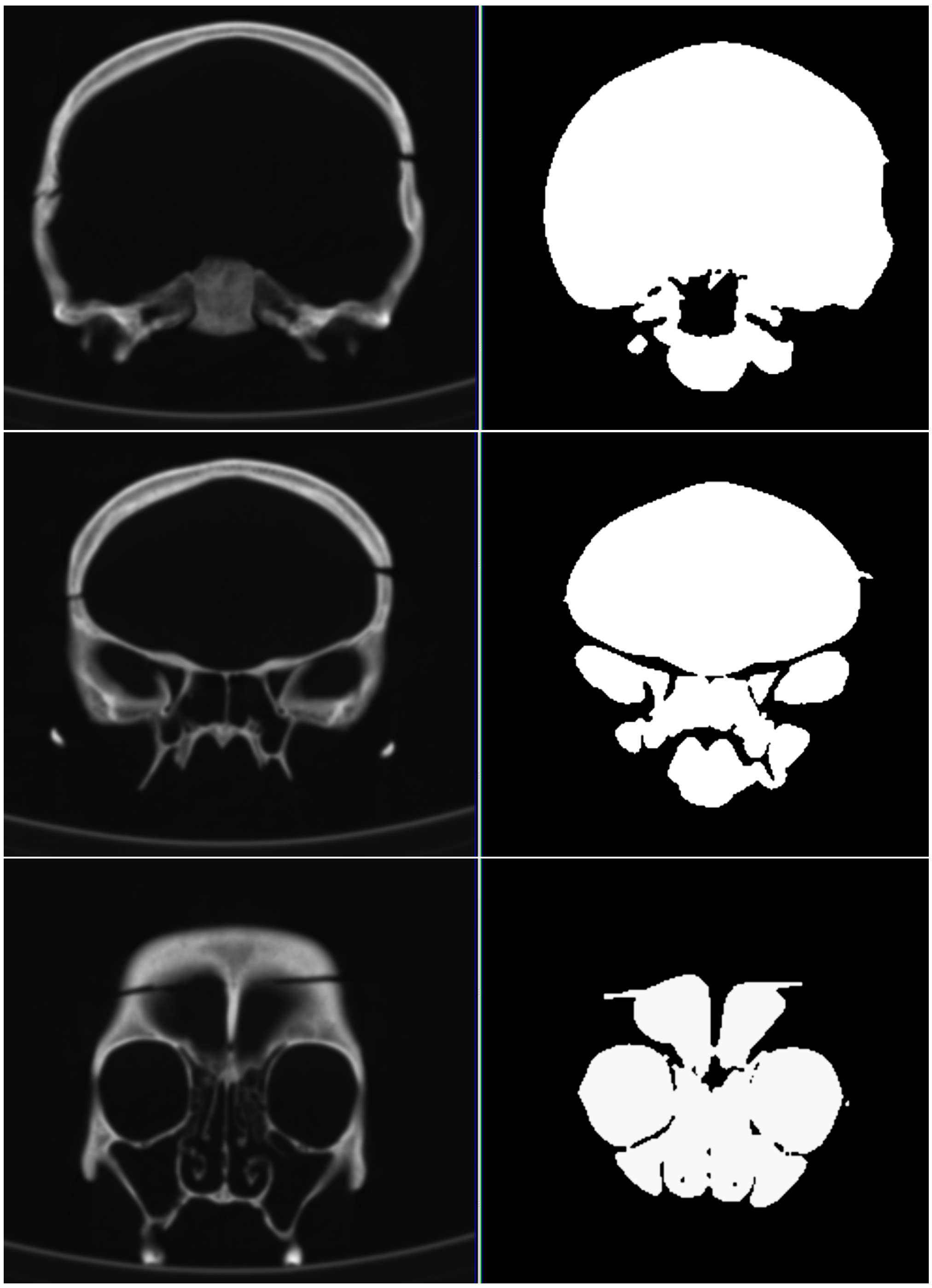


Figure A.4: These three images illustrate the leaking of level set through throat hole, cavities near nose and eye sockets.

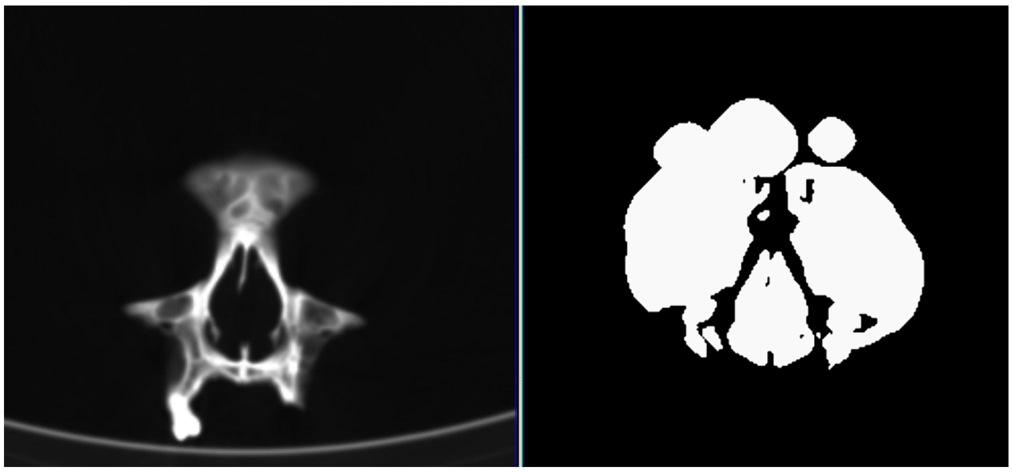


Figure A.5: This slice is really poorly segmented due the leaking of the level set through the previous slices (see the figure A.4).

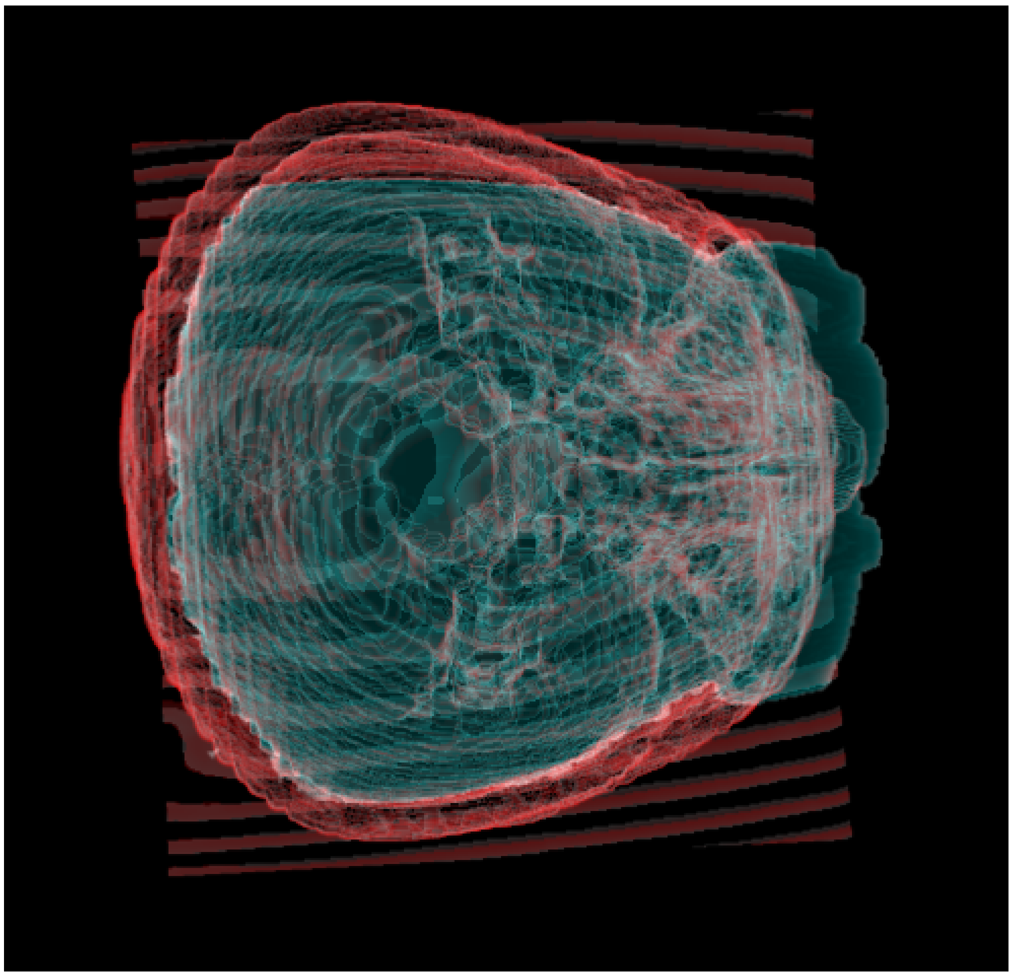


Figure A.6: The original skull data set was blended (the red background) with the result of segmentation (the bluegreen overlay).

Appendix B

Tools setup

This part is a cookbook how to install all the necessary content to be able to begin development for the Cell/B.E. It also covers some bugs fixing as well as useful hints how to use the partial SDK content. The instruction concerning the SDK installation was based on installation on a i686 machine. But shall be the same for the other architectures since it operates on the same operating system and repository versions of third party software.

The first step is installation of Fedora operating system. For details see fedora official site [14].

B.1 SDK installation

As first you have to do is to download actual SDK. Go to `http://www-128.ibm.com/developerworks/power/cell/downloads.html?S_TACT=105AGX16&S_CMP=LP`. You should download following files:

1. SDK 3.1 Installer cell-install-3.1.0-0.0.noarch.rpm (11MB), contains install script and other stuff for SDK installation.
2. SDK 3.1 Developer package ISO image for Fedora 9 CellSDK-Devel-Fedora_3.1.0.0.0.iso (434MB), contains rpm packages that actual SDK is composed from (SDK packages)
3. SDK 3.1 Extras package ISO image for Fedora 9 CellSDK-Extras-Fedora_3.1.0.0.0.iso (34MB), contains some extra packages for Fedora

Download it wherever you want (even though in documentation is /tmp/cellsdkiso). Lets call the folder ISODIR. First you shall stop the YUM updater daemon.

```
/etc/init.d/yum-updatesd stop
```

If it outputs: ”bash: /etc/init.d/yum-updatesd: No such file or directory”, you do not have any YUM updater daemon installed so you can skip this step. Now issue following command to install required utilities for SDK installation

```
yum install rsync sed tcl wget
```

Now install the downloaded installation rpm.

```
rpm -ivh ISODIR/cell-install-3.1.0-0.0.noarch.rpm
```

After this step you have new stuff in `/opt/cell` installed. There is SDK installation script (cellsdk) located as well. It is wrapper for YUM that manages the SDK packages. Run it with parameter –help to see the usage. So next step is to run it.

```
/opt/cell/cellsdk --iso ISODIR -a install
```

Parameter –iso tells to use the downloaded ISOs and path where they can be found for mounting onto a loop-back device. Parameter -a disables agreeing licenses. Otherwise you have to write some ’yes’ to agree. Process begins with setting local YUM repositories pointing to the ISOs. Then all default packages are installed with all their dependencies. To check result of the installation issue

```
/opt/cell/cellsdk verify
```

Now we have SDK installed. Lets continue with installation of IDE. It consists again of packages. Now install yumex that provides graphical interface to YUM to simplify processing packages. It let you simply check packages that you want to install.

```
yum install yumex
```

To install CellIDE run yumex, go to Group View→Development→Cell Development Tools. Check *cellide*, that is actual IDE (Eclipse with cell concerning stuff) and *ibm-java2-i386-jre*, that is Java Runtime Environment, JRE needed for running of the IDE. Click ’Process Queue’. Note: you should have the ISOs mounted onto a loop-back device. Otherwise you get ’Error Downloading Packages’ after clicking ’Process Queue’. Therefore you have to mount ISOs whenever you want to install package concerning the SDK

```
/opt/cell/cellsdk --iso ISODIR mount
```

After the installation you have two new folders. `/opt/cell/ide` that contains the IDE and `/opt/ibm/java2-i386-50` where the JRE resides. To run the ide you have to specify folder where the JRE is (through -vm param).

```
/opt/cell/ide/eclipse/eclipse -vm /opt/ibm/java2-i386-50/jre/bin/
```

B.1.1 Bug fixing

If you start the IDE and it crashes with unhandled exception it is probably caused by xulrunner library. It is usually installed with Firefox3. There is following workaround:

1. download an older version of the xulrunner

   e.g. from: `http://releases.mozilla.org/pub/mozilla.org/xulrunner/releases/1.8.1.3/contrib/linux-i686/xulrunner-1.8.1.3.en-US.linux-i686-20080128.tar.gz`

2. untar to an accessible directory

   Lets call it XULDIR.

3. edit the

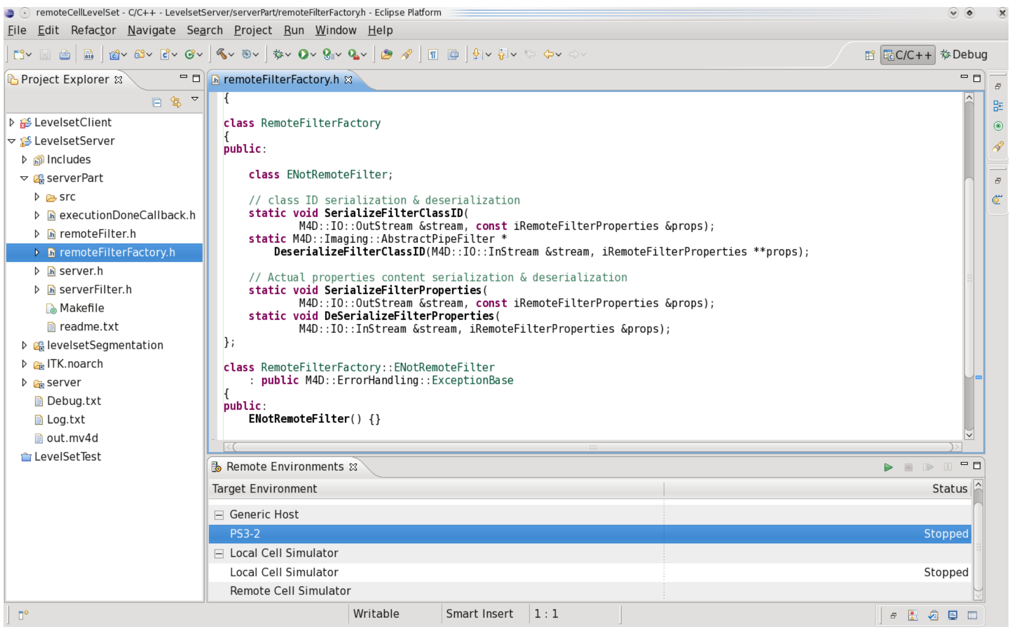
   `/opt/cell/ide/eclipse/eclipse.ini` file as follows:

   ```
   ...
   -vmargs
   -Dorg.eclipse.swt.browser.XULRunnerPath=XULDIR
   ...
   ```

Now you should start the IDE without the crash. Screenshot of a common eclipse view is on the figure B.1.

B.2 IBM Full-System Simulator

The last part of development environment is the IBM Full-System Simulator (systemsim). It is not part of the SDK so you have to download it separately. Visit `http://www.alphaworks.ibm.com/tech/cellsystemsim/download` and download rpm with the simulator appropriate to the platform you are currently using. Be sure to download fedora 9 version of the simulator (cell-3.1-8.f9.*). Then install it.



```
{

class RemoteFilterFactory
{
public:

    class ENotRemoteFilter;

    // class ID serialization & deserialization
    static void SerializeFilterClassID(
            M4D::IO::OutStream &stream, const iRemoteFilterProperties &props);
    static M4D::Imaging::AbstractPipeFilter *
        DeserializeFilterClassID(M4D::IO::InStream &stream, iRemoteFilterProperties **props);

    // Actual properties content serialization & deserialization
    static void SerializeFilterProperties(
            M4D::IO::OutStream &stream, const iRemoteFilterProperties &props);
    static void DeSerializeFilterProperties(
            M4D::IO::InStream &stream, iRemoteFilterProperties &props);
};

class RemoteFilterFactory::ENotRemoteFilter
    : public M4D::ErrorHandling::ExceptionBase
{
public:
    ENotRemoteFilter() {}
```

Target Environment	Status
Generic Host	
PS3-2	Stopped
Local Cell Simulator	
Local Cell Simulator	Stopped
Remote Cell Simulator	

Figure B.1: Sreenshot with opened source, project explorer and remote environments windows.

```
rpm -ivh ISODIR/systemsim-cell-3.1-8.f9.i386.rpm
```

Maybe some dependencies will be missing. So you have to install it. In our case it was libBLT24 and libtk8.5.

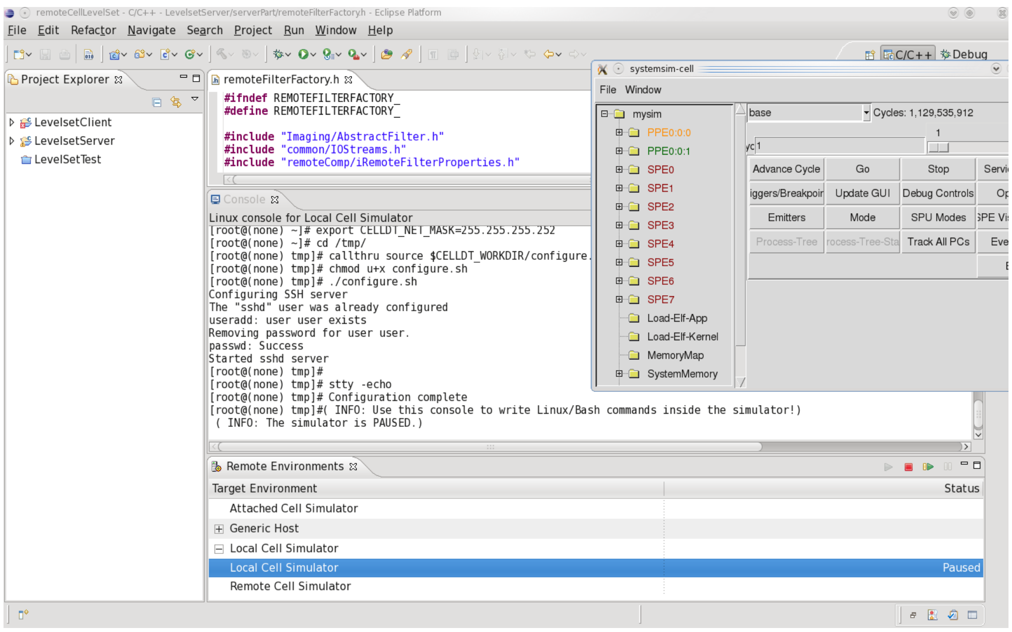
```
yum install blt tk
```

Now you have simulator installed. But it has nothing to simulate. Image with image of simulated Fedora 9 system is needed (sysroot image). It is among SDK rpms so install it using yumex (Cell Development Tools→sysroot_image). Now all necessary stuff is installed. You could start the IDE and start development. But there are some bugs to fix yet.

B.2.1 Bug fixing

One issue (stack issue) is with tcl (scripting language that is used for configuration of the systemsim). There is bug with stack size checking that causes cycling of tcl code. To workaround this problem use *ulimit* command that changes default environment of Linux programs and disables the stack size checkings.

```
ulimit -s unlimited
```



```
#ifndef REMOTEFILTERFACTORY_
#define REMOTEFILTERFACTORY_

#include "Imaging/AbstractFilter.h"
#include "common/IOStreams.h"
#include "remoteComp/iRemoteFilterProperties.h"
```

Linux console for Local Cell Simulator

```
[root@(none) ~]# export CELLDT_NET_MASK=255.255.255.252
[root@(none) ~]# cd /tmp/
[root@(none) tmp]# callthru source $CELLDT_WORKDIR/configure.
[root@(none) tmp]# chmod u+x configure.sh
[root@(none) tmp]# ./configure.sh
Configuring SSH server
The "sshd" user was already configured
useradd: user user exists
Removing password for user user.
passwd: Success
Started sshd server
[root@(none) tmp]#
[root@(none) tmp]# stty -echo
[root@(none) tmp]# Configuration complete
[root@(none) tmp]#( INFO: Use this console to write Linux/Bash commands inside the simulator!)
 ( INFO: The simulator is PAUSED.)
```

Target Environment	Status
Attached Cell Simulator	
⊞ Generic Host	
⊟ Local Cell Simulator	
Local Cell Simulator	Paused
Remote Cell Simulator	

Figure B.2: Screenshot of Eclipse cellide after simulator startup. Remote environments window contain Local Cell Simulator item that is in paused state. Above this window is simulator console with few last outputs. On the upper left side is part of simulator control window.

The last is to fix actual tcl script that manages loading the sysroot_image (21% issue - loading of the sysroot_image freezes on 21% so is not started and thus unusable). It is caused by wrong triggers that are triggerd when some text is output to the simulator console during the sysroot_image loading. There are probably triggers that wait for text from a previous version of SDK that is never output in the current version. That is why the loading freezes on 21%. To fix it you have to edit `/opt/cell/ide/eclipse/plugins/com.ibm.celldt.simulator.profile.default_3.1.0.200809010950/simulator_init.tcl` file. Replace the "Welcome to Fedora Core" string with "Welcome to Fedora" and "INIT: Entering runlevel: 2" with "Starting login process".

It is useful to create starting script that solve the stack issue and add systemsim directory to PATH (needed for running).

```
ulimit -s unlimited
PATH=/opt/ibm/systemsim-cell/bin:\$PATH
/opt/cell/ide/eclipse/eclipse -vm /opt/ibm/java2-i386-50/jre/bin
```

Startup of the simulator shall finish to paused state after application of the fixes. Screenshot of started simulator is on the figure B.2.

B.2.2 Installation of libraries into sysroot_image

Because sysroot_image is provided as an image of installed Fedora 9 without any Cell/B.E. libraries so next step is to install them into sysroot_image.

```
/opt/cell/cellsdk_sync_simulator install
```

This shell script installs all rpms for ppc and ppc64 platforms that finds in `/tmp/cellsdk/rpms`. By default these rpms are copied into `/tmp/cellsdk/rpms` during the install process. If they are not still there (or in installed subdirectory) you have to copy them by hand from ISOs (note: ISOs has to be mounted).

```
cp \
/opt/cell/yum-repos/CellSDK-Devel-Fedora/rpms/*.{ppc,ppc64}.rpm\
/tmp/cellsdk/rpms
```

B.2.3 Copying content into sysroot_image

Sysroot_image is a common binary image that can mounted and thus some additional content can be copied into. This is useful when extra third party libraries that are not part of the default image need to be used. In my case that was e.g. boost libraries. To mount the sysroot_image issue:

```
mount -o loop /opt/ibm/systemsim-cell/images/cell/sysroot_disk\
<your mount point>
```

And then copy whatever you want.

B.2.4 Simulator hints

You can ssh to running simulator. It is better to use real bash that the console within IDE. You have all the bash advantages like command and path completion available in contrast to 'text mode' of the IDE console.

Sometimes root user is needed for an operation performed in the simulator. Its password should be disabled. It can be done when sysroot_image is mounted. Under host machine root account the `<sysroot_image_mount_point>/etc/passwd` file should be edited. The first line is the root's so deletion of '*' character from the second field (after the second ':' character) will disable the root's password. Note that this action must be performed when the simulator is not running otherwise the changes will be overwritten by the simulator.

B.3 Using examples

Examples are installed in `/opt/cell/sdk/src` as tarballs. So you have to untar each you want to use. It is good to start with the examples and tutorial sources. Each folder has its own makefile that manages the makefiles in its sub folders. So you can call only the top level one to build all projects in the sub folders or any from the sub folders to build the particular projects.

It is convenient to use the sample sources in CellIDE where you can build it as well and create run/debug configuration for running within a cell environment. To use the example code (for example `/opt/cell/sdk/src/tutorial/simple`) create new c++ makefile project. Click right button on it to get into properties. C/C++ general tab → Paths and Symbols → Source location. Here you have to add the folder with the sources (`/opt/cell/sdk/src/tutorial/simple`) by 'create / link folder' button →advanced → link to folder in filesystem. Now you have two folders in list. The first one is the original, created during project creation and the other newly linked folder with the source. You can delete the original one since you are not going to use it. Next is necessary to set up 'Build directory' to tell the IDE where shall search for the makefile. It is C/C++ Build tab. Use 'Workspace' button to specify the folder because it will use workspace_loc variable and thus independent on exact location on filesystem.

B.4 Tuning memory usage on PS3

PS3 has only 256MB RAM memory and even not the whole is visible for operating system. This is very small amount for operating system and programs together. When install fedora system with default state and boot it up, the amount of remaining free memory is about 10MB. It is insufficient for either debugging and compilation. So some of the resources has to be switched off. Our PS3s are accessed remotely via ssh so there is no need for X server. So this is the first thing you can turn off. This is performed by change of default runlevel from 5 to 3. Run level setting is in `/etc/inittab` file. So change in line

```
"id:5:initdefault:"
```

the 5 to 3.

Another resource are services. Here you have to consider if the service you want to turn off is really unnecessary. In Our PS3 NeworkManger and Wireless supplicant was turn off. NOTE: when you turn off the network manager, you have to turn on the network service otherwise the networking will not run properly. For service management within ssh console the `/usr/sbin/ntsysv` manager is quite useful. After disabling all unnecessary services we got about 130MB of free space.

Yellow dog distributions goes even beyond. They can access another 256MB in PS3 locked for graphics. Special device is created and the graphic memory is then used as a swap partition. For details see `http://us.fixstars.com/products/ydl/`.

B.5 Performance tools

Information about packages concerning performance tools can be obtained by:

```
yum groupinfo "Cell Performance Tools"
```

If they are not already installed issue following command to install them.

```
yum groupinstall "Cell Performance Tools"
```

B.6 Visual Performance Analyser - VPA

Another useful tool is VPA. It is not part of SDK so it should be downloaded separately. Visit `http://www.alphaworks.ibm.com/tech/vpa` for details. After installing (actually unpacking) the downloaded file similar fix to the one in eclipse.ini file (see paragraph B.1.1) should be done to run the VPA correctly.

B.7 Cell/B.E. cross compilation

When cross-compiling a Cell/B.E. program which uses lot of third party libraries it is good idea to share some folders on actual Cell/B.E. machine. It avoids necessity of copying different architecture stuff to the machine where the cross-compilation is performed as we did. In our case it was boost and ITK libraries and we performed cross-compilation for Cell/B.E. on a i686 machine. We wanted to use repository versions of the Cell/B.E. resp. ppc64 libraries on our i686 machine. So the first thing we tried was to install appropriate packages of different architecture. But we have not found a way how to do it. Every try has failed due to architecture mismatch. So we believe that mounting remote Cell/B.E. machine shared folders is the only way how to use repository content directly and avoid copying the different architecture content.

Appendix C

Content of the DVD

There is an index.htm that could be used as starting point. It points to this thesis as well as to install all additional third party content required by test applications. There are also instructions how to compile and use attached code and data.

The index file also contains user documentations of all the test applications.