UPTEC IT08 001
Examensarbete 30 hp
Januari 2008

The Cell BE as a Time Domain Correlator

for Radio Sensor Networks on the Ground and in Space

Martin Wåger

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

The Cell BE as a Time Domain Correlator for Radio Sensor Networks on the Ground and in Space

Martin Wåger

This report presents a time domain correlator (TDC) for the Cell Broadband Engine (Cell BE). The purpose of the report is to evaluate the use of the Cell BE for signal processing in radio sensor networks both on ground and in space. The TDC is implemented using a streaming algorithm that lowers the memory requirements and runs in real time. It is shown that the Cell BE is very suitable for the implemented algorithm and reaching 40% of the theoretical maximum performance in its current form. It is believed that after optimization the application will come very close to the maximum of 204,8 GFLOPS. In the evaluation, it is concluded that the latency reducing design and the high performance of the Cell BE makes it well suited for signal processing.

Handledare: Jan Bergman
Ämnesgranskare: Olivier Verscheure
Examinator: Anders Jansson
ISSN: 1401-5749, UPTEC IT08 001

Contents

1 Introduction **9**
1.1 Purpose 9
1.2 Cell Broadband Engine Processor 9
1.3 Time Domain Correlator 10
1.4 Goals 10

2 Signal processing in Radio Sensor Networks **11**
2.1 Radio Sensor Network 11
2.2 The three-channel digital RVFS 12
2.3 Signal processing 12
2.4 Correlation 12
2.4.1 Definition 13
2.5 Correlation using the Fourier transform 14
2.5.1 Speed of different FFT algorithms 14
2.5.2 Length of the FFT input 14

3 Cell Broadband Engine **15**
3.1 Cell Broadband Engine Architecture 15
3.1.1 Reasons for the CBEA design 15
3.1.1.1 Dealing with latency 15
3.1.1.2 The barrier 16
3.1.2 Breaching the barrier 16
3.2 The Cell Broadband Engine 16
3.2.1 PowerPC Processor Element 16
3.2.2 Synergistic Processor Element 17
3.2.2.1 Pipelines 17
3.2.2.2 Double precision floating numbers 17
3.2.2.3 Local Store 18
3.2.2.4 Memory Flow Controller 18
3.2.3 Element Interconnect Bus 18
3.2.4 Memory Interface Controller 18
3.3 Floating operation performance of the Cell BE 19
3.3.1 Real versus theoretical FLOP 19
3.4 Intrinsics 19
3.5 Direct Memory Access transfers 19
3.5.1 Alignment 20
3.5.2 Maximizing DMA transfer speed 20
3.6 Systems featuring the Cell BE 20

3.6.1 IBM BladeCenter QS20 . . . 20
3.6.2 IBM BladeCenter QS21 . . . 20
3.6.3 Roadrunner . . . 20
3.6.4 Sony PlayStation 3 . . . 21

4 Application design analysis **23**
4.1 Preparations before coding . . . 23
4.2 Implementing the correlation . . . 23
4.2.1 Streaming Correlation Algorithm . . . 24
4.2.2 Benefits and drawbacks of the SCA . . . 25
4.3 Analyzing the implementation . . . 25
4.3.1 Limitations arising from the computations . . . 25
4.4 Parallelization . . . 26
4.4.1 Partitioning the inner loop . . . 26
4.4.2 Partitioning the work within the inner loop . . . 26
4.4.3 Partitioning the outer loop . . . 27
4.4.4 A combination of the above . . . 27
4.5 Vector/SIMD parallelization . . . 27
4.6 Writing efficient SPE code . . . 27
4.6.1 Eliminating branches by function in-lining . . . 28
4.6.2 Eliminating branches by loop unrolling . . . 28
4.6.3 Branch hinting . . . 28
4.6.4 Instruction scheduling . . . 28
4.7 Incoming data . . . 28
4.8 Storage . . . 29
4.9 Analyzing the storage . . . 29
4.9.1 Total storage space needed . . . 30
4.9.2 Limitations arising from the data storage . . . 30
4.10 Data transfer in a parallel application . . . 31
4.10.1 Storage on the SPEs . . . 31
4.10.2 SPE data transfer requirements . . . 31
4.10.3 Reducing the needed transfer rate . . . 32
4.10.4 Hiding the data transfer . . . 32
4.11 Flexible implementation . . . 34
4.11.1 Kernels . . . 34

5 Application development stages **35**
5.1 Scalar implementation on a single processor . . . 35
5.2 Implementation using SIMD vectorization . . . 35
5.2.1 Vectorizing the data . . . 37
5.2.2 Vectorizing the code . . . 37
5.2.3 Performance testing . . . 37
5.2.4 Conclusions from the performance tests . . . 39
5.2.5 Problems with an implementation of the SCA . . . 39
5.3 Vectorized implementation on a single SPE . . . 40
5.3.1 Implementing the DMA transfers . . . 40
5.3.2 Converting PPE SIMD instructions to SPE . . . 40
5.4 Parallel implementation using all SPEs . . . 40
5.4.1 Calculating the distribution . . . 41
5.4.2 Parallel SPE code . . . 41

5.5 Comparison of the different implementations 42

6 Description of the final application **43**
6.1 Outline 43
6.1.1 Deviation from the definition 43
6.1.1.1 Ignoring max-lag start data 43
6.1.1.2 Ignoring the end data 43
6.2 Structure 43
6.3 Global values 45
6.4 PPE main 45
6.5 Data loader function 46
6.6 Correlator function 46
6.7 Data saver function 47
6.8 Structure file 47
6.9 Definition file 47
6.10 SPE main 47

7 Conclusions **49**
7.1 Source code 49
7.2 Performance 49
7.3 Possible improvements 49
7.4 Related work 49
7.5 Comparison with FFT applications 50
7.5.1 FFT method 50
7.5.2 Comparison 50
7.5.2.1 Memory usage comparison 51
7.5.3 Conclusion of the comparison 51
7.6 Programming the Cell BE 52
7.6.1 Memory 52
7.6.2 Parallelism 52
7.6.3 Optimization 52
7.7 Cell BE as a time domain correlator 53

Chapter 1

Introduction

1.1 Purpose

The main purpose of this report is to evaluate the Cell Broadband Engine Processor and investigate its use for signal processing in radio sensor networks both on the ground and in space. In order to do this in a realistic way and also to get a useful result we decided to create an application for correlation, a common task in signal processing. The design process is documented with emphasis on differences and difficulties, and the end result is evaluated here.

1.2 Cell Broadband Engine Processor

The Cell Broadband Engine Processor (Cell BE) is the first processor from a new exiting architecture, the Cell Broadband Engine Architecture (CBEA). The architecture was designed by a co-operation of IBM, Toshiba and Sony Computer Entertainment Incorporated (SCEI). The architecture is described in detail in [1]. It was originally intended to be part of SCEIs game console PlayStation 3 and aimed at game and multimedia applications but the architecture also looks very promising for scientific computation.

The architecture is designed to overcome three barriers that face modern processors, the memory, power and frequency barriers [2]. The CBEA overcomes these barriers by offloading the main processor to a number of smaller and simpler co-processors each with local memory, providing a constant access delay, and dedicated Direct Memory Access (DMA) logic.

The Cell BE have a theoretical peak performance of 256 GFLOPS[1] (single precision) and given suitable applications it can, unlike many other architectures perform close to this number. The Cell BE high "performance to watt" rating makes it an interesting choice for applications that has a limited power supply, limited cooling possibilities or both, for example as the main processor on-board a satellite in space.

[1]Giga (billion) Floating Operations per Second, see section 3.3

1.3 Time Domain Correlator

Correlation (see section 2.4) is a computationally heavy task, taking an order of N^2 calculations to correlate two signals of length N. Because of this it is common to transform the signals from time domain to frequency domain using a Fourier transform (see section 2.5). This transform is usually done with an implementation of the Fast Fourier Transform (FFT) algorithm reducing the calculations needed to the order of $N \log N$. However there are already several implementations of FFT applications for the Cell BE (for example [3, 4, 5]) and other groups are working on FFT based correlators (a so called FX-correlator[2]) for the Cell BE so it was decided that it would be more interesting to create a time domain correlator (TDC).

Since many signal processing tools often requires the result to be in the frequency domain the result from a TDC is usually Fourier transformed after wards forming XF-correlator[3] but here this step is omitted since one purpose of the application is to reduce the size of the result (see section 1.4). A TDC suffers from the exponentially increasing number of computations needed but have some benefits (see section 7.5). The implementation of a TDC it also much less complex and have a greater potential to fit the CEBA. Here we present an altered implementation named the Streaming Correlation Algorithm (see section 4.2.1) that enables the TDC to be run in real time using a limited amount of storage space.

1.4 Goals

In order to create a useful application some probable scenarios where thought up. One possible way to use the Cell BE could be as the on-board computer in a satellite. The satellite could be fitted with one or more antennas and several such satellites could be launched in a cluster. The cluster could then exchange and correlate the data gathered from the antennas in real-time. The result is much smaller then the raw data and could be sent down to Earth in a compact way. This is desirable since a large amount of antennas create much data and the connection between the satellite and Earth is limited.

Another way of using the Cell BE is on remote antenna clusters on Earth, performing the same task as the last example in order to reduce the data traffic from the clusters to the main recipient. These scenarios defines the following requirements on the application:

1. The application should be able to work with different antennas and different antenna configurations.
2. The application should be able to reduce the size of the result and the execution time by limiting the maximum lag (see section 2.4.1)
3. It should be possible to change the maximum lag at run time.
4. The application should use limited external resources.
5. The application should be designed for optimum computing speed.
6. The applications should allow other smaller tasks to be carried out at the same time.

[2]FX comes from Fourier transforming (F) and then multiplying (X)
[3]X for the correlator and F for the Fourier transform

Chapter 2

Signal processing in Radio Sensor Networks

2.1 Radio Sensor Network

A conventional radio telescope is usually either very large and expensive or not very sensitive. A cheaper alternative is to create a network of smaller antennas and then use signal processing on the antenna outputs to form a single large virtual antenna. An additional benefit is gained if the networked antennas are omnidirectional. Then several virtual antennas can be formed focused on different targets allowing several users to work on different tasks at the same time. The signal processing needed to accomplish this however is very computationally intense. Currently the calculations are often done in non-real time on saved data or in real-time using specialized hardware.

One example of a such a network is the Low Frequency Array (LOFAR) project. It consists of several remote sensor fields connected together using a large network. Phase 1 of the LOFAR project implements 45 sensor fields with a total of ~15000 different antennas. The antennas are processed locally and the total data output is in tens of Gibit/s[1]. The data is transferred to the LOFAR Central Processor, a BlueGene/L based supercomputer. The Central Processor process data in non-real time but LOFAR also features real-time processing. Its Compact Core antenna array, consisting of ~3000 antennas is correlated by the Wide Field Correlator consisting of a large amount of FPGA chips.

An alternative method to the centralized data processing is to spread all the computational power closer to each antenna. This reduces the data volume that needs to be sent as the processed data is much more compact. This is important on land, the planned extension to the LOFAR project will require a network capable of Tibit/s[2], but even more in space where the distance and equipment severely limits the bandwidth back to earth.

[1] Gi is short for gibi or gigabinary and is 2^{30}
[2] Ti is short for tebi or terabinary and is 2^{40}

2.2 The three-channel digital Radio Vector Field Sensor

This project uses the three-channel digital Radio Vector Field Sensors (RVFS)[9] as an example input. The sensor is based on the Information Dense Antenna (IDA) first described in [10]. The RVFS measures, in three dimensions, either the the electric field E using three orthogonal dipole antennas or the magnetic field B using three orthogonal loop antennas. The following text will use the electric field measurements but the calculations and results are the same for the magnetic field. The antenna measures the field on three axis (x,y,z) and samples the values at an interval that depends on its design. The output is a complex time series representing the electric field.

For every sample the RVFS sends six values, the real and imaginary parts (denoted In-phase and Quadrature phase respectively) of the three axis. These are grouped together as $E(n)$ where n denotes the sample index. One sample is thus

$$E(n) = \left[I_{x_n} + Q_{x_n}, I_{y_n} + Q_{y_n}, I_{x_n} + Q_{z_n}\right] \tag{2.1}$$

where $n = \text{sample rate} * t$ and every value I or Q is in the range of $\left[-2^{15}, 2^{15}\right]$ due to 16-bit sampling in the RVFS. This information is streamed as $E = [E_1, E_2, ...]$.

The current implementation of the antenna communicates over a UDP/IP connection with a 10Mbit network interface so the maximum data output speed from the antenna is limited to a theoretical maximum of this speed. The current implementation of the antenna has a maximum output bandwidth of 82.1kHz giving a data rate of 82.1kHz * 3 * 2 * 16bit = 7.88Mibit/s[3] or 82.1k samples/s.

2.3 Signal processing

In order to form a virtual antenna from the sensor network we need to form the coherency density tensor. To do this we form matrix $EE^\dagger$, where † denotes the Hermitian conjugate[4], and calculate its time average, denoted by brackets as $\left\langle EE^\dagger \right\rangle_t$). The time average for one three-axis RVFS antenna can be rewritten as

$$\left\langle EE^\dagger \right\rangle_t = \begin{pmatrix} E_x \star E_x^* & E_x \star E_y^* & E_x \star E_z^* \\ E_y \star E_x^* & E_y \star E_y^* & E_y \star E_z^* \\ E_z \star E_x^* & E_z \star E_y^* & E_z \star E_z^* \end{pmatrix} \tag{2.2}$$

where E_n only contains the the n-axis values, $*$ denotes complex conjugate and $\star$ denotes correlation. This tensor is perhaps more well known in its frequency domain form where it is called the spectral density tensor.

2.4 Correlation

Correlation, either between two different signals called *cross-correlation* or the signal with itself called *auto-correlation*, is a comparison of two signals where one is shifted in time relative to the other. The function shows the degree to which the two signals are related at different times. Correlation is a well used function in many different

[3] Mi is short for mebi or megabinary and is 2^{20}

[4] The Hermitian conjugate is the transpose and complex conjugate together.

fields such as image processing, acoustics, and signal processing. Since it is used in several fields there are several different ways of describing the function and its use mathematically. Here the correlation will be given the following definition:

2.4.1 Definition

Two complex discrete time series from the antenna are denoted $x(n)$ and $y(n)$. The correlation between them is denoted $x(n) \star y(n)$. The operation results in a sequence denoted $R_{xy}(m)$ for $m = 0 \ldots \infty$ where m is the shifting index called *lag*. $R_{xy}(m)$ is defined as

$$R_{xy}(m) = \mathbb{E}(m)\left[x(n+m)y^*(n)\right] = \sum_{n=-\infty}^{\infty} x(n+m)y^*(n) \tag{2.3}$$

where $\mathbb{E}$ is the expected value operator and y^* is the complex conjugation of y[5] . Since we don't have infinite information about the signals $x(n)$ and $y(n)$ we make a finite approximation of the signals by multiplying them with a window function

$$w(n) = \begin{cases} 1 & -N \le n \le N \\ 0 & otherwise \end{cases} \tag{2.4}$$

and we get

$$\hat{R}_{xy}(m) = \mathbb{E}(m)\left[x(n+m)y^*(n)w(n)\right] = \sum_{n=-\infty}^{\infty} x(n+m)y^*(n)w(n) = \sum_{n=-N}^{N} x(n+m)y^*(n)$$

As $N \to \infty$ our approximation $\hat{R}_{xy}(m) \to R_{xy}(m)$. N also limits the maximum shifting we can do since for $(n+m) > N$ we have $x(n+m) = 0$ so N is the maximum lag (max-lag) we can calculate. The correlation is a Hermitian function

$$R_{xy}(-m) = R^*_{yx}(m) \tag{2.5}$$

so the final equation is

$$\hat{R}_{xy}(m) = \begin{cases} \sum_{n=0}^{N-m} x(n+m)y^*(n) & m \ge 0 \\ \hat{R}^*_{yx}(-m) & m < 0 \end{cases} \qquad m = 0 \ldots N \tag{2.6}$$

(where the sum is restricted to $N-m$ since the results after that is only zero). This can also be written in matrix form as

$$\hat{R}_{xy} = \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ \vdots \\ r_N \end{bmatrix} = \begin{bmatrix} x_0 & x_1 & \ldots & x_{N-1} & x_N \\ x_1 & x_2 & \ldots & x_N & 0 \\ x_2 & x_3 & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_N & 0 & \ldots & 0 & 0 \end{bmatrix} * \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{2.7}$$

The correlation of N samples takes an order of N^2 calculations . This can be reduced by limiting the range of m to a number M less then N. This will require only an order of $N * M$ calculations but will give a shorter max-lag.

[5]We use the conjugate since we are looking at the phase difference, if not used the average would be zero

2.5 Correlation using the Fourier transform

Instead of computing the correlation in the original time domain one can use the fact that the correlation transforms to a conjugated multiplication (taking an order of N calculations to perform) in the frequency domain. To convert the data to frequency domain we use the Fourier transform. For discrete data this transform is calculated by the Discreet Fourier Transform (DFT). This algorithm takes an order of N^2 steps to compute so using it gives no increase in calculation speed. Fortunately there is another version, the Fast Fourier Transform (FFT) that does this in an order of $N \log N$ steps[6]. The FFT can easily be inverted to transform the data back to time domain. This means that correlation of N samples takes an order of $2N \log N + N$ operations using a FFT.

There are many different FFT algorithms but the most used is the Cooley-Tukey (C-T) algorithm. The C-T algorithm works by decomposing the work in parts and computing the parts recursively. The decomposition can be made in different sizes but the most common is the $N/2$ or radix-2 variant (requiring N to be a power of 2). This means that a C-T based radix-2 FFT takes an order of $N \log_2 N$ steps to compute.

2.5.1 Comparing the speed of different FFT algorithms

Commonly the speed of an FFT implementation is measured in FLOPS. This makes it difficult to compare implementations as they might require a different number of FLOP for each step. The C-T algorithm takes on average 5 FLOP per step and since it is the most commonly used algorithm this number is often used when calculating the FLOPS for FFTs (even though the actual implementation takes more or less). It is also assumed that the FFT is in radix-2 (again this might not be the actual case) so the FLOPS count presented is based on $5N \log_2 N$ FLOP for N samples.

2.5.2 Length of the FFT input

If we are to correlate N samples using the FFT we would like to transform all the samples to the frequency domain, multiply them and transform back to time domain. However the FFT algorithm must have access to all the samples while calculating and for large N this becomes a problem since large memory usage usually means slow performance.

Preferably we want to stay in the cache, or for the SPEs in the local store (see section 3.2.2.3), thus the size of N for a fast implementation is limited to the number of samples that can be stored there. On many platforms we find that the speed of the FFT drops considerably for sample lengths above 8-32Ki[7] (depending on the cache size). On the Cell BE this restriction is less obvious and good speeds can be had even at length of 16Mi. The length of an FFT implementation is usually called points so an FFT with the input size of 1Ki would be called a 1024 point FFT.

[6] The base for the logarithm depends on the radix (see section 2.5).
[7] Ki is short for kibi or kilobinary and is 2^{10}

Chapter 3

Cell Broadband Engine

3.1 Cell Broadband Engine Architecture

The CBEA is an extension to the PowerPC architecture and aimed at distributed processing. The architecture does not specify an exact implementation, instead it allows a number of different configurations. It only requires that an implementation has at least one PowerPC Processor Element (PPE), at least one Synergistic Processor Element (SPE), one Internal Interrupt Controller (IIC) and an Element Interconnect Bus (EIB) connecting the units within the processor.

3.1.1 Reasons for the CBEA design

Until recently the main method to increase processor performance was to increase the clock frequency (cycles per second) of the processor. But the frequency can not be increased forever. Higher frequencies requires more power and this means more heat. It also means that there is less time for information (in the form of electrons) to move around as the speed of the information is limited to the propagation velocity.

This means that the area that can be reached within one cycle shrinks and that we either have to pack the circuitry closer (by shrinking the "wires") or suffer from increased latency. Since data storage takes lots of space on silicon there will always be a need to move data storage off the processor chip resulting in even further latency. A processor typically only have a few KiB in is registers, a few more in a close level (L1) cache, a few hundred in a higher level (L2) cache and the rest off the chip in main memory or on secondary storage.

3.1.1.1 Dealing with latency

Current processors suffers from a main memory latency of several hundreds (if not thousand) cycles. There are many methods for dealing with this latency developed over the years. Two examples are pipelines that can load new data at the same time as older data is processed and the cache hierarchy described above. Processors also have dedicated hardware that prefetch data into the caches before it is requested. This hardware looks ahead in the code and tries to guess what the code will require next. Branching disturbs this guessing so it also has to be predicted and so on. All this results in bloated architectures where the actual computing circuitry is a smaller part of the total processor.

3.1.1.2 The barrier

Even with the above mentioned methods the latency is the main barrier that prevents execution speed. Much time is spent on waiting for data because it is not available in close storage, perhaps due to a miss-prediction. The extra circuitry needed to alleviate the problem also increases the energy consumption of the processor and further increases the need to remove the heat this creates. This resulted in a barrier that future processor development could not easily break.

3.1.2 Breaching the barrier

The CBEA deals with this barrier with an uncommon design. It uses a conventional processor (the PPE) to handle the operating system (OS) and other maintenance tasks while providing specialized processors (the SPEs) for calculations. The SPEs are not fitted with a cache, instead it as a Memory Flow Controller (MFC). The MFC deals with data transfers to and from main memory independently leaving the SPE to handle other things. This hides the latency as computations can be carried out while new data loads. It also simplifies the architecture reducing power consumption and heat generation.

3.2 The Cell Broadband Engine

The Cell Broadband Engine Processor (Cell BE) is the first commercially available processor from the Cell Broadband Engine Architecture (CBEA). The current version of the Cell BE features one PPE and eight SPEs. An overview of its architecture is shown in figure 3.1. The processor is clocked at 3.2GHz.

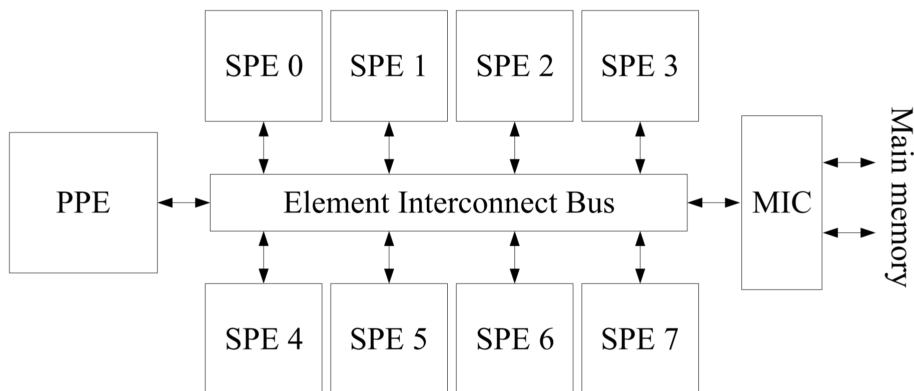

Figure 3.1: Overview of the Cell Broadband Engine

3.2.1 PowerPC Processor Element

The PPE contains a traditional PowerPC Processor Unit (PPU), two layers of cache and a memory controller (the Power Processor Storage Subsystem). A diagram of the PPE can be seen in figure 3.2. The PPU is a 64bit, dual-thread PowerPC fitted with an 32KiB L1 and a 512KiB L2 cache. As stated above the PPE is mainly intended to handle the OS, do maintenance and control the SPEs. The PPU can fetch four instructions at a

time, and issue two. Instructions can be somewhat reordered to improve performance but its pipeline is kept quite short in order to simplify its design. It features the AltiVec vector/SIMD[1] multimedia extensions and has 32-128bit vector registers.

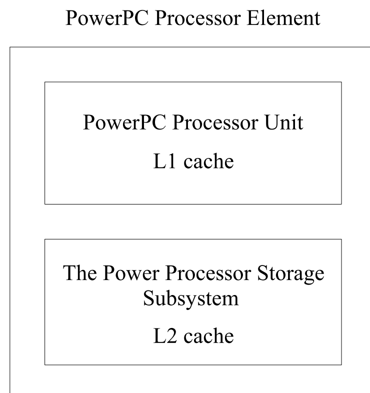

Figure 3.2: Diagram over the PPE

3.2.2 Synergistic Processor Element

The SPE contains the Synergistic Processor Unit (SPU) with a Local Store (LS) and a Memory Flow Controller (MFC) with a Direct Memory Access (DMA) controller. A diagram of the SPE can be seen in figure 3.3. The SPU is a 128bit RISC[2] processor specially designed for SIMD instructions (although it can handle scalar code). It can issue two instructions at once, one to each of its two pipelines (see below). It does not use the same set of instructions as the AltiVec on the PPE but they are similar. The SPU has 128-128bit vector registers but unlike the PPE it has no additional registers so it uses the vector registers for everything including storing scalars. The SPU has access to the 256KiB large LS instead of a cache but can not access the main memory directly. The SPU lacks advanced branch hinting and suffers quite much from missed branches. It makes up for this by allowing the programmer to place hints in the code as to what branch is most likely to be taken. But even with this it is best to avoid branches as much as possible. See section 4.6 for methods to do this.

3.2.2.1 Pipelines

The SPE has a dual pipeline. The pipelines are named even and odd and perform different tasks. The odd pipeline handles such things as load/store, branching and shuffles (see section 5.3 about the shuffle operation). The even pipeline handles floating and integer arithmetic, rotates, compares and more. For a complete list see [7].

3.2.2.2 Double precision floating numbers

The first version of the Cell BE does not feature full speed when computing double precision floating numbers. The pipeline latency is more than twice then the latency for single precision floats. On top of this it also prohibits dual issuing.

[1] Single Instruction Multiple Data, see section 4.5.
[2] Reduced Instruction Set Architecture

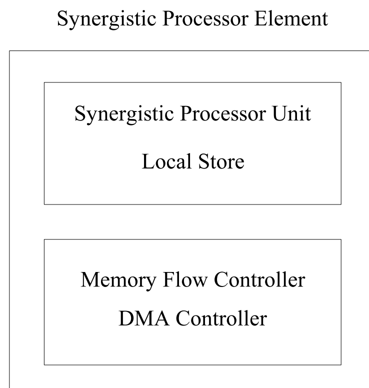

Figure 3.3: Diagram over the SPE

3.2.2.3 Local Store

The LS is the only memory that the SPE can access directly and it is used both for program and data storage. In order for the SPE to access the main memory a DMA transfer (see section 3.5 for more detail) has to be made by the MFC. The MFC then loads/saves the needed data to/from the LS. Since the LS it is not cached it has a constant, quite short, latency for load/stores.

3.2.2.4 Memory Flow Controller

The MFC contains the DMA controller and is responsible for the SPEs interface to the main memory via the EIB. The MFC has several channels that can be used for SPE-PPE, SPE-SPE and SPE-main memory communications. The MFC also contains three mailboxes, two outbound (from the SPE) and one inbound that are used to pass messages to the PPE. These can be used for example by the PPE to control the SPE program flow or by the SPE to signal task completion to the PPE.

3.2.3 Element Interconnect Bus

The EIB connects all elements in the Cell BE. It consists of four rings, two in each direction where the elements are daisy-chained in a circle. The EIB can transfer up to 204,8GiB/s if the transfers don't overlap. Overlaps can happen if the transfers are to/from the same place, if all transfers are in one direction or if the destination is six elements away (blocking two rings). It is important for applications requiring large data transfers that overlaps does not happen as they can reduce the transfer rate to about a third of the maximum (see [12] for an experiment).

3.2.4 Memory Interface Controller

The MIC is the interface between the EIB and main memory. It provides two channels and if both channels are used the theoretical maximum bandwidth is 25,6GiB/s. Normally this is a few GiB lower due to memory maintenance operations.

3.3 Floating operation performance of the Cell BE

Each SPE is capable of issuing one vector floating point operation per cycle (single precision). Given the processor frequency of 3.2GHz we get a theoretical maximum processing speed of 3,2*1*4=12,8 GFLOP per SPE.

The SPE also provide fused operations (for example the fused multiply and add doing c = a*b+c in one cycle). When they are exclusively used we get a maximum of 25,6 GFLOP. The PPE runs at the same speed but can issue two floating point operations per cycle giving it a theoretical maximum processing speed of 51,2 GFLOP.

In total a Cell BE with eight SPEs will have a theoretical maximum processing speed of 256 GFLOP. It should be noted however that the PPEs main task is to manage the OS and the SPEs so it is perhaps better to state only the SPEs total theoretical maximum processing speed of 204,8 GFLOP (for eight SPEs) as the systems maximum.

3.3.1 Real versus theoretical FLOP

Many manufacturers boast the FLOP speed of their processors but it usually does not mean much. It is hard or impossible on many architectures to reach 100% (or even 50%) of the stated number. On the Cell BE this is not the case. There are several examples (see [12] for one) of applications that run close to 100% efficiency on the SPEs.

3.4 Intrinsics

The architecture of the Cell BE requires that the programmer takes control of the memory system than other processors. Also SIMD operations requires more control compared to scalar operations since they group values in vectors and sometimes individual values must be computed on. The Cell BE provides an extension to the normal C and C++ languages with the SIMD and SPU instruction intrinsics. These intrinsics are used like function calls and substitutes one or more in-line assembly instructions. A full list of intrinsics are provided in [7].

3.5 Direct Memory Access transfers

As stated above all SPEs main memory access has to be done using DMA transfers. These transfers are requested on the SPE or the PPE and are handled by the associated MFC. The MFC manages the transfers independently of SPE. The SPE can send either single transfer commands that are issued immediately or a list of commands that are queued on the MFC. The MFC can then carry out the listed transfers out-of-order if it will increase the speed of the transfer. The reordering can be controlled from the SPE by issuing a barrier or a fence command. A barrier means that no transfers before or after the barrier can be moved to the opposite side. A fence means that a transfer issued before the fence might not be moved after the fence but transfers issued after the fence might be moved anywhere.

DMA transfers move one, two, four, eighth, 16 or a multiple of 16 bytes at a time with a maximum size of 16KB.

3.5.1 Alignment

Alignment refers to the address of the source and/or destination of a data transfer. In order for the transfer to be aligned to x bytes the memory address must be in a multiple of x. For an article describing this in detail see [8].

The MFC requires that DMA transfers are naturally aligned up to 16 bytes, for example a transfer of eight bytes must be aligned on an eight byte boundary. Transfers over 16 bytes need only be aligned on a 16 byte boundary. In addition to this the address of a DMA list must be aligned on an eight-byte boundary. If the transfer is not correctly aligned an interrupt is raised and the PPE have to change the address so that the transfer can be preformed. Something that takes lots of time.

3.5.2 Maximizing DMA transfer speed

Maximum performance of DMA transfers is achieved if the source and destination address is 128 byte aligned and the size of the transfer is a multiple of 128 bytes (provided no overlap happens, see section 3.2.3). The 128 byte alignment is important because if the addresses are not 128 byte aligned the data transfer speed is reduced to approximately half of the maximum.

3.6 Systems featuring the Cell BE

There are a number of commercially available systems that feature the Cell BE. Some examples of these are listed below.

3.6.1 IBM BladeCenter QS20

The first BladeCenter system from IBM to feature the Cell BE was the QS20. It features two Cell BE processors on a double wide server blade. It has 1GiB main memory and dual Gibit Ethernet connections together with up to four InfiniBand I/O links.

3.6.2 IBM BladeCenter QS21

QS21 is the second generation BladeCenter with Cell BE. It is of standard width allowing for up to fourteen blades in one chassis. It has 2GiB of main memory, dual Gibit Ethernet connections and twice the I/O rate of the QS20

3.6.3 Roadrunner

Roadrunner is the name for new supercomputer that IBM is building for the US Department of Energy in the Los Alamos National Laboratory in New Mexico [11]. It will use a hybrid design where one Opteron X64 processor from Advanced Micro Devices will be teamed up with two Cell BE. It will feature a redesigned version of the Cell BE with improved dual precision calculation speed. This version is also planned to be featured in the next version of the BladeCenter. The goal is to achieve one Peta FLOP of sustained LINPACK (a library for performing numerical linear algebra) calculating speed.

3.6.4 Sony PlayStation 3

The Sony PlayStation 3 contains one Cell BE processor running at 3.2GHz. It has 7 SPEs available (one is turned off in order to increase the production yield) and 256MiB of main memory. It has one Gibit Ethernet connection.

Sony has graciously allowed the PS3 users to install another OS side by side to its game OS. The Linux enabled PS3 will then have access to 6 SPEs (the 7th is reserved for the Game OS). There is a number of different Linux operating systems that works on the PS3 (Gentoo, Fedora and Yellowdog to mention a few) but the current Cell SDK (3.0) from IBM only supports Fedora 7.

Chapter 4

Application design analysis

4.1 Preparations before coding

It is a good practice to analyze the problem at hand before any coding starts. This is especially true for the Cell BE due to two factors, the parallelism that comes with the SPEs and the need for DMA that comes from the incoherent memory model. Both these factors need to be kept in mind the whole time as they are the main obstacles to overcome but also the main reasons the Cell BE is able to perform at its level.

There are a number of things to consider

- How to implement the correlation algorithm
- How to parallelize the algorithm
- Where from and how the incoming data arrives
- How to store the incoming data and the results
- How to ensure that the code is modular, that is able to use different antenna configurations and max-lag

We also need to analyze the limitations and possible approximations that these decisions impose. The following sections describe the design preparations in detail.

4.2 Implementing the correlation

Equation 2.6 works by calculating one sum for every m. An example of this is shown figure 4.1. A straight forward implementation would calculate each $\hat{R}_{xy}(m)$ in turn (row-wise in the figure). This works well if we have all the data at hand. In this case however we have a constant stream of incoming data and a limited storage space so the straight forward implementation does not work. Instead another algorithm has to be constructed.

Looking again at figure 4.1 we see that every new sample we receive that sample must be multiplied with all the previously received samples. The last three x samples received are color-coded in the figure to show where they are used. So instead of calculating each $\hat{R}_{xy}(m)$ in turn we see that we can calculate each new sample in a diagonal way (the colored band in the figure). Separately this is not helpful for us as it

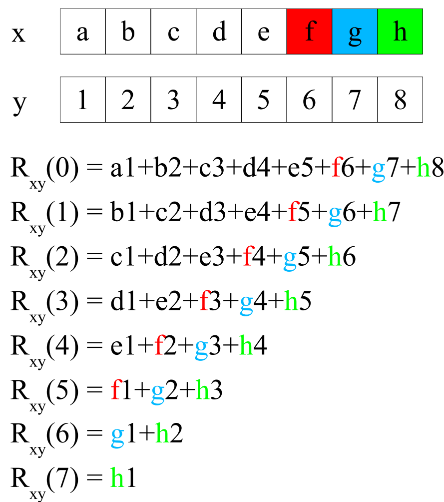

Figure 4.1: Correlation unrolled. The last three datums are colored to show where they end up

Program listing 4.1 Pseudo code for the SCA implementation

```
for all the data in the stream (n) do
  for all stored samples (m) do
    r_xy[m]+=x[n] * conjugate(y[n−m]);
    r_yx[m]+=y[n] * conjugate(x[n−m]);
  end
end
```

does not reduce the storage space needed but if we limit our max-lag and only correlate the lag $m = 0...M$ we limit the number of samples we need to store. This is due to the fact that the calculation of $\hat{R}_{xy}(m)$ only uses samples m steps back (the black numbers multiplied with the color coded data).

4.2.1 Streaming Correlation Algorithm

From this it can be seen that instead of having an outer loop over m and an inner loop over n we can do the opposite and see the incoming data stream as the outer loop over n and do an internal loop over m. This means that the calculation can be done for each new sample in real-time. An example of this change, named the streaming correlation algorithm (SCA), is shown in pseudo code in listing 4.1. In the algorithm the stored data is indexed from 0 to m and the stream is indexed from 0 to n. Equation 2.5 shows that negative values for R_{xy} can be obtained by conjugating R_{yx} so they do not need to be calculated.

4.2.2 Benefits and drawbacks with the streaming correlation algorithm

The greatest benefit from using the SCA is that we only need to store M samples in memory compared to the N samples needed for a straight implementation. The SCA is also computed in real time and given that the computer keeps up with the incoming data it eliminates the use of a large input buffer[1]. This means that we will save $2N - M$ times the data size of a sample of memory (given that the input buffer also is length N).

The most obvious drawback is of course the reduced max-lag this results in. However in order to reduce data output from the application this was the plan in any case. A more subtle but more important drawback of the SCA is that it is harder to unroll in an efficient way, something that will be shown in section 5.2.5. Also note that the SCA calculates backwards meaning that we use max-lag "old" data compared to the straight forward implementation that looks at future data. This implies that we have max-lag old samples stored before we can begin correlating.

4.3 Analyzing the implementation

An implementation of the SCA requires that each new incoming sample from the data stream must be multiplied with all the stored samples. This result is then added to the result storage containing the previous result. For every sample we get

$$\gamma = [\text{number of antennas}] * [\text{axis on each antenna}]$$

new complex values and the number of stored samples is max-lag. We need to do γ^2 complex multiplications for each stored sample (to correlate all antennas and axis) so in total we have to do $\gamma^2 \left[\text{max lag}\right]$ complex multiplications and the same number of additions (to store the result). We do four multiplications and two additions for every complex multiplication so for the whole process we have to do $4\gamma^2$ multiplications and $3\gamma^2$ additions giving a total of $12\gamma^2 \left[\text{max lag}\right]$ operations. Using the combined multiply and add function (see section 3.3) available on both the PPE and the SPE we need to do $4\gamma^2 \left[\text{max lag}\right]$ operations per incoming sample.

4.3.1 Limitations arising from the computations

As shown in section 3.3 each SPE has theoretical maximum processing power of 25,6 GFLOP using single precision. This will limit the max-lag that we can theoretically achieve and still be able to keep up with the data output of the antenna. An example with one three-axis RVFS antenna, a data output of 82,1k samples/s and using the above formula gives

$$\frac{4 * (1 * 3)^2 * \left[\text{max lag}\right] * 82,1 * 10^3}{10^9} < 25,6 \Rightarrow \left[\text{max lag}\right] \lesssim 8661$$

for each SPE used. Table 4.1 shows the theoretical maximum max-lag for different antenna configurations and different number of SPEs. Note that increasing the incoming data rate with a factor x decreases the maximum max-lag measured in time with a factor x^2.

[1] An input buffer is needed for the straight implementation in a real time environment to store incoming data while computing it.

	Number of SPEs			
Antenna configuration	1	6	8	16
One three-axis RVFS antenna at 82,1Ks/s	8661	51969	69292	138584
Two three-axis RVFS antennas at 82,1Ks/s	2165	12992	17323	34646
Three three-axis RVFS antennas at 82,1Ks/s	962	5774	7699	15398
One three-axis RVFS antenna at 821Ks/s	866	5196	6929	13858

Table 4.1: Theoretical maximum max-lag for different antenna configurations and different number of SPEs

How likely is it then to reach the theoretical maximum? If we had an unlimited register space on each SPE the problem would be trivial but we do not of course. However the architecture of the Cell BE allows us to hide much of the data latency amongst the calculations and as shown in the matrix multiplication example in [12] it is possible, with the use of buffered DMA access and optimized SPE code, to come very close to the theoretical maximum.

4.4 Parallelization

Parallelization can be done in many different ways. Looking at the pseudo-code in listing 4.1 we see that we have two loops that can be made parallel, the inner loop over m and the outer loop over n. We can also divide the work done in the inner loop. This gives us four options to consider:

1. Partitioning the inner loop
2. Partitioning the work within the inner loop
3. Partitioning the outer loop
4. A combination of the above

4.4.1 Partitioning the inner loop

Method one divides the inner loop (of max-lag length) over the number of available SPEs. The main advantage of this solution is that the stored data and result buffers are not shared amongst the SPEs so no mutual exclusion[2] (mutex) needs to be implemented and used. It is also simple to divide the work evenly for a different number of SPEs and different antenna combinations. The main drawback is that each loop on the SPEs need to compute $4\gamma^2$ multiply and add operations, something that quickly saturates the available registers and slows down the computations. See section 5.2.3 for a discussion on register use.

4.4.2 Partitioning the work within the inner loop

Method two divides the work in the inner loop. For example one SPE could correlate only one axis. This method alleviates problem with register use described above but

[2]Mutual exclusion provides a lock on a variable so that no one except the process holding the lock can access or change that variable.

suffers from the need of mutex. Something that can result in stalls as processes have to wait in turn to use a certain variable. It is also hard to make the partitioning for an unknown number of SPEs and the partitioning does not scale well with an increasing number of SPEs (as we might run out of work to share).

4.4.3 Partitioning the outer loop

Method three computes one or more new incoming samples on each SPE. This method gives the SPEs more work per new incoming data but it suffers both from the need mutex and the saturation of registers.

4.4.4 A combination of the above

There are many possible combinations of the three partitioning schemes mentioned above. One could for example partition both over the inner loop and over the work needed to alleviate both the problem with mutex and register saturation. Another example is a combination of method one and three which could reduce the number of data loads as much of the data is similar for each new incoming sample. This method does share stored data but not results and since the stored data does not change we do not need to have mutex.

4.5 Vector/SIMD parallelization

Perhaps the easiest way to get good performance on the Cell BE is to use the data-level parallelism that SIMD operations give. This is true in some part for the PPE but on the SPE SIMD operations are almost a must. In order to use SIMD operations the data operated on must be organized in sets. It is possible to form sets containing two, four, eight or sixteen values depending on the data type used.

Each set is 128 bytes long and called a vector. SIMD operations work on all values in the vector at once using the same operator for all. Correctly used this will speed up the code two to sixteen times (again depending on data type). There are two common ways of storing data in vectors, either in an array of structures (AOS) form or a structure of arrays (SOA) form.

An AOS stores data from different source in each component of the vector. For example a system with four co-ordinates would store x,y,z and w in one vector. The name AOS comes from the fact that a set of vectors can be represented as an array of structures where each structure contains the component values.

In a SOA only data from one source is stored in the vector. Using the example from the last section one vector would contain x values, one y values and so on. SOA can be represented as a structure of arrays.

4.6 Writing efficient SPE code

The SPE is very powerful but it requires careful programing. Since the Cell BE does not feature an advanced branch prediction it is important to eliminate branches. This can be done in several ways, for example by in-lining and unrolling code. For branches that can not be removed it is helpful to use branch hinting to instruct the processor which branch is most likely to be taken. The Cell BE also features a dual issue pipeline

(see 3.2.2.1) which means that two instructions can be scheduled simultaneously if arranged correctly

4.6.1 Eliminating branches by function in-lining

Using functions for parts of the code that needs to be performed more than once is a good way of simplifying the source code. However every time the function is used a branch in the main code is needed, and after the function is done it returns using another. One way of eliminating reducing such branches is to in-line in the function, giving the compiler an instruction to place the function code in the main code every time it is used. This eliminates the branch but increases the size of the compiled program. On the SPEs this might be a problem as the program share the LS with the data.

4.6.2 Eliminating branches by loop unrolling

Loops are very common in code but since every loop iteration requires a branch they might slow the application down. The number of loop iterations needed (and thus the number of branches) can be reduced by unrolling the loop. This is done by doing two or more loop tasks per loop. However one must eliminate all false dependencies. False dependencies occur when one line of code depends on another not because one value depends on the result of an earlier calculation but because they share a register. Removing false dependencies might requires the use of additional registers.

4.6.3 Branch hinting

A branch hint will allow the processor to load the instructions that results of the branch before the branch itself is calculated. This is especially important for unconditional branches as the SPU will always assume that a branch without a hint is not taken. The hinting might create extra stalls but there are better than the stalls that results in a branch miss.

4.6.4 Instruction scheduling

Since the two pipelines on the SPE can do different things at the same time programs can benefit from instruction scheduling. This needs to be done on assembly level as higher level functions often do several low level instructions at once. The idea with the scheduling is to keep both pipelines fed at the same time by reordering the code. This might not always be possible if there are dependencies that can not be removed or an imbalance between the number of instruction of the two types.

4.7 Incoming data

The RVFS communicates over UDP so we should use this protocol as data input. But the implemented application uses data stored on the local hard-drive as input instead. This is because we want to be able to run the application without interference of network delays, something that could affect test runs and give varying results. It is also useful to be able to create simpler test data to eliminate computing errors and to be able to reproduce test runs using the same data. Because of this the applications should be

designed in such a way that the data loading part could be exchanged to a UDP version without affecting the rest of the code.

4.8 Storage

An application based on the SCA stops using old samples after a certain point. This means that we should use a FIFO[3] buffer to store the incoming samples in. To avoid shuffling the buffer for every new sample we get, a circular buffer could be used. This means that a new sample overwrites the oldest going round in a circle. There is no support for such a structure in hardware so instead a circular buffer should be implemented using a standard array and made circular using index values that points to the different positions in the array. These indexes are then made circular with modular arithmetic. See figure 4.2 for an example of a circular buffer using pointers.

Modular arithmetic depends on the modulus operator, denoted $\bmod$. The operator returns the remainder of an integer division (e.g $5 \bmod 3 = 2$). If we have an array of length x and want to make it circular we index it with a variable n that always is in $n \bmod x$. Any increase or decrease of an index variable must be done inside a modulo operation, so for example increasing n with two is done as $(n+2) \bmod x$.

One important thing to know is the distance between two index values, a and b. This distance is normally given by $b-a$ but in modulo x it is done as $(x-a+b) \bmod x$. Using figure 4.2 for an example we see that the distance between the blue and the green arrow is $(16-8+15) \bmod 16 = 7$ and the distance between the blue and the red arrow is $(16-8+3) \bmod 16 = 11$. If the array is indexed often it is useful to keep the length of x to a power of two. This is because the modulo function can be replaced with the faster bit-wise AND operation, denoted $\wedge$, as $x \bmod 2^n \Leftrightarrow x \wedge (n-1)$.

The result from a correlation is an array of values with max-lag length. Since this is a fixed value the result can be stored in a standard array.

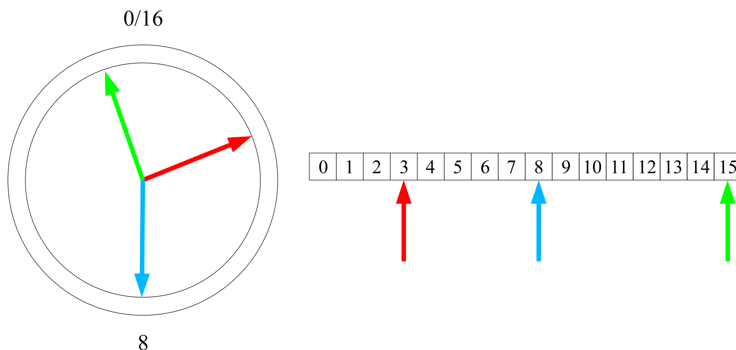

Figure 4.2: An example of an array made circular using pointers

4.9 Analyzing the storage

From each antenna comes a continuous stream of complex values (described in 2.1). Restating the definition from section 4.3 we get

[3]First In First Out

$$\gamma = [\text{number of antennas}] * [\text{axis on each antenna}]$$

new complex values for every new sample resulting in 2γ values (one for each I and Q). Using the SCA means that we need to store max-lag old samples.

The result from the computation also needs to be stored. We get γ^2 complex results per lag we calculate for a total of $2\gamma^2$ values (again one for each I and Q). If we use the SCA we will get max-lag number of results.

The fact that the results for each new incoming data are added to the previous result limits the maximum time the application can run as the stored results eventually exceeds the largest number possible to store. If the type used to store the results has a maximum value of [result max] and the incoming data has a maximum value of [data max] then we can run for a maximum of [result max] / [data max] samples. The addition can also be done with an averaging function eliminating this problem. For a discussion of the two methods see section 4.9.2.

4.9.1 Total storage space needed

The total storage needed for a given max-lag depends on the size of the data type we store the data in. We need to store

$$\alpha = 2\gamma \left[\text{max-lag}\right] [\text{s_data}]$$

bytes of old data (where [s_data] is the size of data type used in bytes) and

$$\beta = 2\gamma^2 \left[\text{max-lag}\right] [\text{s_data}]$$

bytes of result. In total we need

$$\delta = \alpha + \beta$$

bytes.

4.9.2 Limitations arising from the data storage

We have shown that the internal data representation is limited by

$$\delta \leq \left[\text{available main memory}\right]$$

The size of memory needed thus depends on the number of antennas, the number of axis on each antenna, the max-lag and the size of the type used to store the data. However the size of max-lag is also limited by computing speed (see section 4.3.1) so memory is normally not a problem. An example with one three-axis RVFS antenna, a max-lag of 70000 and using the float data type (of size 4B) gives a total storage space size of

$$\left(1 * 3 * 2 + (1 * 3)^2 * 2\right) * 70000 * 4 \approx 6{,}4\text{MiB}$$

We have also shown that the maximum time possible to calculate (often referred to as the window), is limited by the largest value allowed in the data type used and the maximum size of the incoming data. There are a number of ways to deal with this:

- A straight summation using a integer data type will result in a correct result but with a very limited window

- A straight summation using a floating point data type will result in a loss of precision as the result grows. It will also limit window but much less then an integer.

- A averaging summation will result in a loss of detail (either loosing small values or cutting large ones). It does not pose any limitations on the window however.

4.10 Data transfer in a parallel application

So far we have only considered PPE data storage. If we are to utilize the Cell BE potential we need to use the SPEs. This in turn means that we need to transfer and store data in the LS of each SPE.

4.10.1 Storage on the SPEs

The SPEs will run a program that is similar to the PPE variant and needs similar storage. The LS is limited to 256KiB however and that space also needs to be shared with the program code. Each SPE can thus only take part of the total stored data. Because both the stored data and the result are of the same length each SPE can fit

$$\psi = \left\lfloor \frac{\text{[Available LS]}}{\left(2\gamma + 2\gamma^2\right)\text{[s_data]}} \right\rfloor$$

samples from the total (where $\lfloor x \rfloor$ is the floor function rounding down). For example if 200KiB of LS space is allocated we could fit 2133 of the max-lag samples with one three-axis RVFS antenna or 248 with three antennas if we use a 4B data type.

4.10.2 SPE data transfer requirements

The data transfer rate from the incoming data device to the main memory is not high because it is limited to the data output of the antennas. But when we start working on the SPEs the data transfer rate increases since we need to transfer both the stored data and the old results from the main memory to the LS of the SPEs and then transfer the new results back. These transfers must be done once for every incoming sample so we need to transfer $\alpha + \beta$B of data to the LS and βB of data back giving a required transfer rate of $(\alpha + 2\beta) *$ [incoming sample speed]B/s. This can also be written as

$$\left(2\gamma + 4\gamma^2\right)\left[\text{max-lag}\right]\text{[s_data]}\left[\text{sample speed}\right]\text{B/s}$$

We also need to transfer the work sample (the data being correlated) for an extra $2\gamma\,$[s_data] [sample speed] but that is a negligible amount. An example using one three-axis RVFS antenna with an output of 82,1k samples/s, a max-lag of 70000 and a data type size of 4B would require a transfer rate of

$$((6+36)) * 70000 + 6) * 4 * 82100 \approx 899\text{GiB/s}$$

This is more then 35 times the theoretical maximum transfer rate of the MIC.

4.10.3 Reducing the needed transfer rate

Looking at the data sent to each SPE we see that it is almost the same for each transfer. The data is only shifted once with one new sample added and one old removed. The result data is not the same for each transfer but the result from several samples is accumulated. This means that if we send the SPE some additional stored data (overhead) in each transfer we can calculate several samples for each transfer, adding the different results together before they are sent back. This will reduce both the incoming and outgoing transfer to the SPEs. The overhead of each transfer is illustrated in figure 4.3.

In addition to the extra overhead we also need to send the same amount of extra (future) work data to compute with. This means that LS storage space is decreased by the size of the overhead times two. With overhead the LS can fit

$$\psi' = \left\lfloor \frac{[\text{Available LS}] - 4\gamma\,[\text{calc./transfer}]\,[\text{s_data}]}{\left(2\gamma + 2\gamma^2\right)[\text{s_data}]} \right\rfloor$$

samples from the total max-lag (were [calc./transfer] is the number of times we calculate per transfer). The total overhead (including the extra work) sent in samples is calculated as

$$\epsilon = 2\left\lceil \frac{\left[\text{max-lag}\right]}{\psi'} \right\rceil [\text{calc./transfer}]$$

where the first term (within the ceiling function) is the number of times we need to send data to the SPEs and the rest is the overhead per such transfer. Note that there is no overhead in the results transferred back to the main memory.

Using this method the new transfer rate becomes

$$\frac{\left(\left(2\gamma + 4\gamma^2\right)\left[\text{max-lag}\right] + 2\gamma\epsilon\right)[\text{s_data}]\left[\text{sample speed}\right]}{[\text{calc./transfer}]}\text{B/s}$$

As an example we can take one three-axis RVFS antenna and correlate to a max-lag of 70000 with 200KiB of LS space and 250 extra samples all using a data type size of 4B. This gives $\psi' = 2008$ and a total transfer overhead of 17500 samples. This means that we must send 70000 stored data plus the overhead and the result but only for every 250th incoming sample. This gives a transfer rate of

$$(42 * 70000 + 6 * 17500) * 4 * 82100/250 \approx 3{,}73\text{GiB/s}$$

a reduction of almost 242 times and a much more manageable number. Figure 4.4 shows the reduction for different number of calculations per transfer. The decrease is similar for other antenna combinations and max-lag.

This method to reduce data flow is the same as the second partitioning method mentioned in section 4.4.4. As stated there it does not put any other requirements on the SPE parallelization but it does restrict the max-lag we can calculate to a multiple of the overhead size.

4.10.4 Hiding the data transfer

In order to run the correlation undisturbed from memory access the DMA calls must be hidden. This is possible due to the MFC and its ability to work asynchronous from the

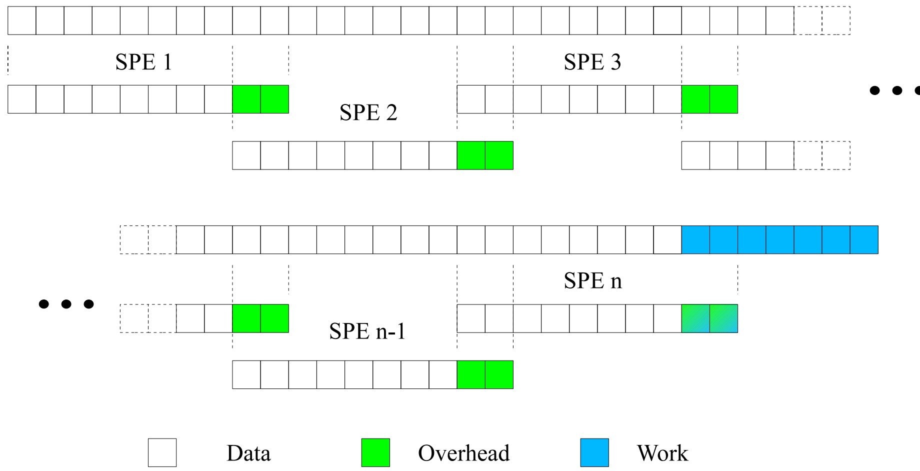

Figure 4.3: Picture showing how data is allocated to each SPE. The green area is the overlap.

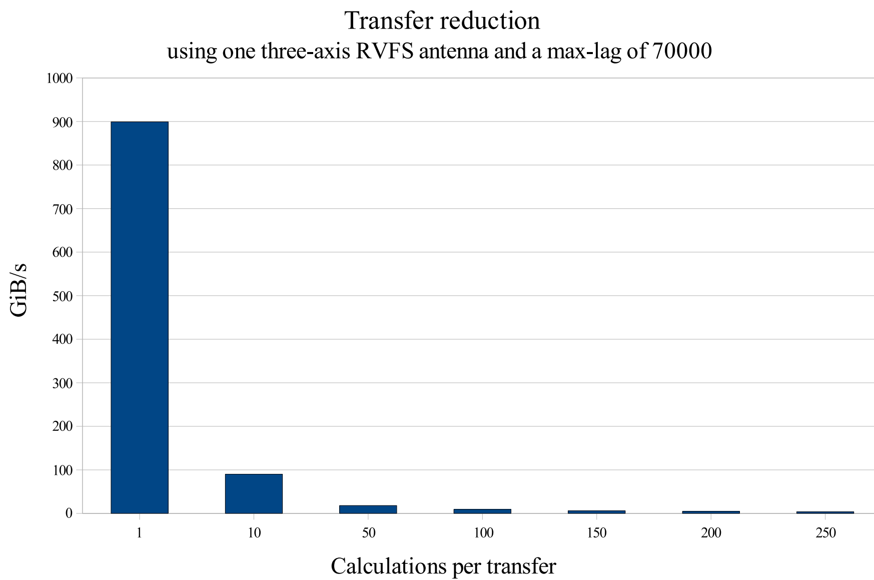

Figure 4.4: Diagram over transfer rate reduction

SPE and the dual issue pipeline that allows the calculations to be scheduled together with the MFC commands. This in turn means that the SPE buffer must be layered so that data can be loaded into one layer and computations can be made with data from another layer.

4.11 Flexible implementation

As different applications have different antenna configurations it is useful if the design could easily be adapted to work with both different antennas and different numbers of axis on the antennas. The code will be written in C and this somewhat limits the modularity but using a general structure to save a sample means that most parts of the code that do not deal directly with the individual parts of data can work with all different configurations (one structure can then contain all the data from several axis and several antennas).

It should be noted that since this implementation reads data from the hard-drive (as explained in 4.7) using a binary file format (for speed) it is best that the test data is created on the same platform as the application using it. This is because a structure saved in binary form is written bit for bit and there is no guarantee in C that a structure will have the same internal data representation on different platforms.

4.11.1 Kernels

As for the parts of the code that directly deals with the data (i.e the computational part) it becomes much harder to stay flexible. This is because the code needs to be fine tuned to the specific setup. Instead several different so called computational kernels can be created, each kernel designed for a particular setup. The kernels can then be linked at compilation (or even at run time) to work for different configurations. These kernels do the majority of the computation and they need to be hand tuned. For this report only one such kernel (for one three-axis RVFS antenna) is built but the results presented here can be used as a basis for creating other ones.

Chapter 5

Application development stages

Programming the Cell BE can be a complex task and it is hard to create efficient code straight ahead. This is especially true for a developer new to the Cell BE. Instead it can be helpful to develop the application in steps, adding new features one at a time as described in [13]. Starting on the PPE gives a feel for the application and helps understanding the data flow. It is not necessary to vectorize on the PPE but many of the instructions used for the AltiVec PPE vectorization are similar on the SPE and it is perhaps easier to do debugging on the PPE. The correlator application was developed in the following steps:

1. Scalar code implementation on the PPE.
2. Vectorized implementation on the PPE.
3. Vectorized implementation on the SPE.
4. Parallel implementation using all SPEs.

For each step the code was tested with different max-lag against test data of a fixed length. The main differences and results are shown in the following sections and the speed improvements are presented last.

5.1 Scalar implementation on a single processor

The scalar implementation for a single processor is quite straightforward. The code looks similar to and closely follows the pseudo code in listing 4.1. The incoming data is stored in a buffer of length *max-lag* and results are stored in arrays of the same length. The main loop of the implementation can be seen in listing 5.1. This is the slowest implementation and is only included as a reference. It can, like the next example, be optimized by unrolling the code but no time was spent doing so.

5.2 Implementation on a single processor using SIMD vectorization

The code in section 5.1 can be vectorized using the AltiVec SIMD functionality available on the PPE. This might seem unnecessary but since the SPEs only operate on

Program listing 5.1 Scalar implemenation

```
#define x_n_Q    (data+i_work)->x.Q
#define x_n_I    (data+i_work)->x.I
#define x_nm_Q   (data+temp)->x.Q
...
#define z_nm_I   (data+temp)->z.I

while( dataLeftToLoad )
{
    /* Correlate one sample */
    for(i=0;i<maxlag;i=i+1)                                    10
    {
        temp =(dataLength+i_work-i) % dataLength;
        /* rxx */
        (r_xx+i)->I += x_n_I*x_nm_I+x_n_Q*x_nm_Q;
        (r_xx+i)->Q += x_n_Q*x_nm_I-x_n_I*x_nm_Q;
        ...
        (For all r__)
        ...
        /* rzz */
        (r_zz+i)->I += z_n_I*z_nm_I+z_n_Q*z_nm_Q;              20
        (r_zz+i)->Q += z_n_Q*z_nm_I-z_n_I*z_nm_Q;
    }
}
```

Program listing 5.2 Extracting and shifting in AltiVec code

```
/* Splat the second value in x(n) */
n_I_0 = vec_splat(x_n_I_t,1);
n_Q_0 = vec_splat(x_n_Q_t,1);
/* Shift x(m) one value left (using concatenation with next vector) */
t_I_0 = vec_sld(x_m_I_1,x_m_I_0,SIZEOFVECUNIT);
t_Q_0 = vec_sld(x_m_Q_1,x_m_Q_0,SIZEOFVECUNIT);
/* Perform complex multiplication: x(n) * conjugate(x(m)) */
x_r_I_0 = vec_madd (n_I_0,t_I_0,x_r_I_0);
x_r_I_0 = vec_madd (n_Q_0,t_Q_0,x_r_I_0);
x_r_Q_0 = vec_madd (n_Q_0,t_I_0,x_r_Q_0);                      10
x_r_Q_0 = vec_nmsub(n_I_0,t_Q_0,x_r_Q_0);
```

vectors (emulating scalar operations if needed) and the SPE vector functions closely match the AltiVec functions the work spent to convert the PPE code to AltiVec will be regained later when we move to the SPEs. Two things needs to be done in order to use the SIMD functionality. First we need to vectorize the data and then convert the code to vector operations.

5.2.1 Vectorizing the data

As described in section 4.5 there are two main ways to vectorize the data. We can either use the AOS form, and get vectors that contains data from one sample, or use the SOA form and get vectors that contain several samples.

One of the goals with the application is that it should work with different antenna specifications. Using AOS form would mean that one vector could contain data from different antennas and/or have components left empty. This leads to more complex implementation and empty components gives a severe computation speed loss. Instead the SOA form is used even though this means that max-lag and data length needs to be multiples of four or the code becomes complicated.

5.2.2 Vectorizing the code

Normally the vectorization of an array multiplication is straightforward. Correlation is different however since every value, even the ones inside each vector, needs to be multiplied with every other value (compared to the the normal case where we only multiply one array with another). This requires us to "splat" the vector into four new vectors. To splat a vector means that we create a new vector containing in all cells only one of the values from the old vector. The other operand to the multiplication then needs to be shifted. This shift is done by concatenating the second operand with its next vector, shifting the result and cutting out the needed values. This operation is shown in detail in figure 5.1 (where the cutting is marked in gray) and a short example in C code is shown in listing 5.2.

5.2.3 Performance testing

Testing the SIMD implementation shows that the performance is not limited by the number of calculations done but instead on the time spent loading data and saving the result. This problem is also described in [14] and there the problem is alleviated by reordering and unrolling the code in order to improve data locality. This means that the overhead from data load/save is reduced as more data is loaded/saved each time. In the example given in [14] the author calculates up to 52 samples per load/save using 31 of the 32 AltiVec registers provided on a PowerPC. This results in a code performing 2.6 times faster then the original SIMD code (and 10-14 times faster then the scalar code), using version 2.95.4 of gcc[1].

A test on the PPE with a slightly rewritten code, as the original one uses another operating system, shows that for the 4.1.1 versions of gcc the results are a bit different. The result of the test runs are shown in table 5.1. The code was tested both on the PS3 and a QS20 and compiled using both the O2 level of optimization, as used in the original example, and with 03, which also uses in-lining and loop unrolling. All four tests gave very similar results, O2 and O3 optimizations where identical.

[1]The GNU Compiler Collection, `http://gcc.gnu.org`

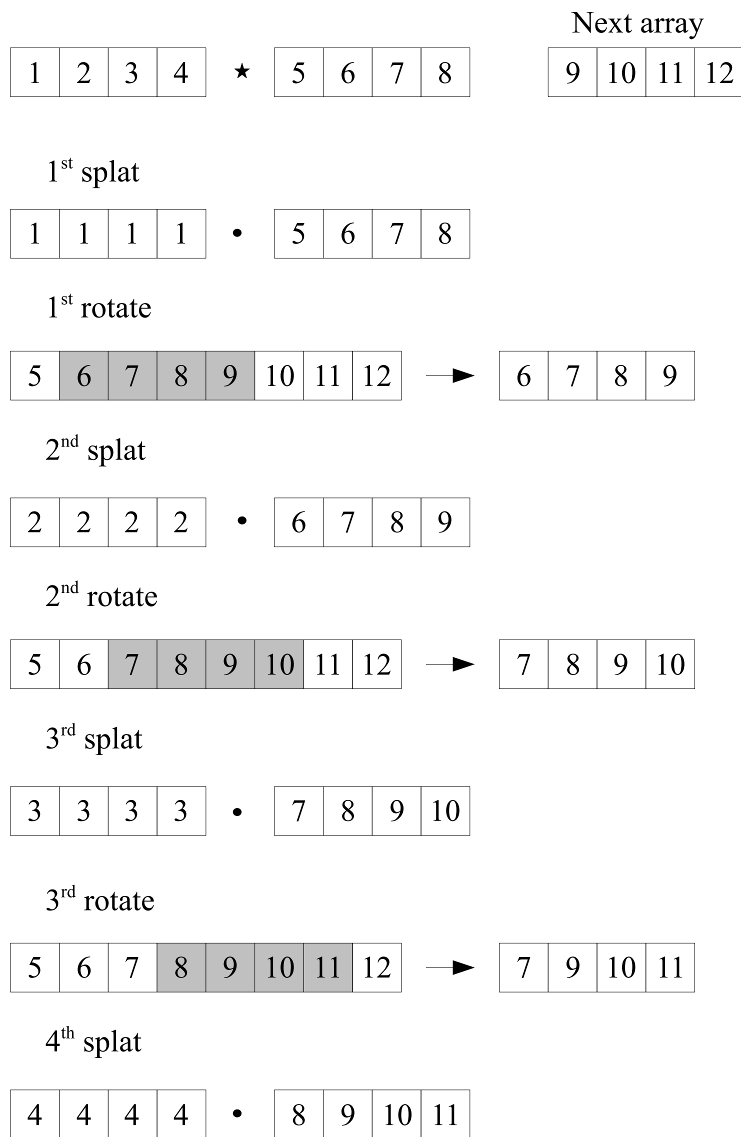

Figure 5.1: One vector being correlated with another using splat and rotate operations. The gray area is the part cut out from the concatenated vectors.

It can be seen that the code does behave like the code described in the article but that the cut-off point comes much earlier. The peak performance was reached at 4x36 (using 23 of the 32 registers) instead of 4x52. One likely cause for this is that the compiler optimization on the PPE uses more registers probably in order to reorganize the code to better utilize the pipeline and the dual issuing available on the PPE. This shows that too much unrolling without paying attention to the assembly instruction ordering can be bad even if there are registers available.

	performance in Ms/s for N=1024	
q	PS3 O2 and O3	QS20 O2 and O3
scalar	0,31	0,31
optimized scalar	1,31	1,32
4x4	1,03	1,03
4x8	2,04	2,05
4x12	3,05	3,06
4x16	3,75	3,76
4x20	4,32	4,33
4x24	4,99	5,00
4x32	5,63	5,62
4x36	5,80	5,75
4x40	3,14	3,15
4x44	2,34	2,35
4x48	1,84	1,85
4x52	1,31	1,31
4x56	1,55	1,55

Table 5.1: Test runs of the FIR filter application described in [14]. Performance is average of ten runs.

5.2.4 Conclusions from the performance tests

Unfortunately the code in the example above only performs a single real correlation but the application we are implementing requires several complex ones. As shown in section 4.3 we need to do γ^2 complex multiplications for every sample. Even at only one three-axis antenna we have nine complex multiplications. The full SIMD implementation will use 28 registers before unrolling. This is already more than used in the fastest example above and thus most likely also above the cut-off point. It is unlikely that simple unrolling can be made on the PPE and no effort was made to do so. A processor with more registers could prove beneficial, something seen in section 4.6.

5.2.5 Problems with an implementation of the SCA

The performance test above shows that it is beneficial to do as many load/saves at one time as possible. An implementation of the straight correlation can compute the result for a fixed lag on the whole data buffer before the result needs to be saved. This means that the algorithm can do a number of loads at the start and then a number of saves at the end of the loop. An implementation of the SCA however computes only one

sample (the most recent) at all different lags and must thus save for every computation. This complicates the optimization of the code.

5.3 Vectorized implementation on a single SPE

Converting the code to run on one SPE requires two changes. First the data needs to be brought into the LS and the results back out and then the PPE SIMD code must be converted into SPE SIMD instructions.

5.3.1 Implementing the DMA transfers

The actual data transfer from the main memory to the LS takes place on the SPE side using DMA calls, as described in section 3.5. DMA transfer is optimal when data is loaded in 16KiB chunks from an EA address with 128-byte alignment to a LS address of the same alignment and DMA transfers creates interrupts if it is not aligned to the minimum requirement.

We need to be able to access the data and result arrays at any index and this means that each sample should be stored on an address that is a multiple of 128 and must at least be stored in an address that is a multiple of 16. To ensure this and to get optimum transfer speed the structures for sample and result are padded so that they are of a multiple of 128 bytes. This together with an aligned initial allocation assures that we always can have maximum DMA performance. This changes the calculations in section 4.10, but only with a multiple of [padding]/[original size].

As seen in section 4.10.3 we need to calculate several samples per data transfer in order to reduce the transfer rate. The sample being worked on is thus copied to the SPE along with a number of additional work data (this is actually future data for the current loop ant it is achieved by buffering the incoming stream in a work buffer). The number of extra work data is the same as the number of times we calculate per loop.

In addition to the overhead we must also send one extra sample for the shifting (see section 5.2.2) of the last vector. This sample is not correlated and does not affect the size of the result.

5.3.2 Converting PPE SIMD instructions to SPE

Converting the vectorized code from PPE to SPE is not difficult. There are only two operations that can not be directly translated, the *vec_splat* and the *vec_sld.* Both operations are replaced on the SPE with the more general *spu_shuffle*. The *spu_shuffle* takes two vectors and a bit-mask as operands and the result is a new vector with elements from the two vectors depending on the bitmap. It can thus be used both for "splatting" and rotating. An example of bit-masks can be seen in listing 5.3. Each of the 16 hexadecimal values specifies the origin of the data that goes in that part. A leading 0x0 indicates the first vector, a 0x1 the second. Every vector is regarded as having 16 parts each part 8 bytes long.

5.4 Parallel implementation using all SPEs

In order to achieve good results on the Cell BE we need to use the all SPEs and this requires parallelization. In order to do this well the work should be evenly distributed

Program listing 5.3 Example of *spu_shuffle* bit masks.

```
/* Bit masks for rotation and spat */
#define SPLAT_FIRST     (vec_uchar16)
  { 0x00, 0x01, 0x02, 0x03, 0x00, 0x01, 0x02, 0x03, \
    0x00, 0x01, 0x02, 0x03, 0x00, 0x01, 0x02, 0x03 }
#define SPLAT_SECOND (vec_uchar16)
  { 0x04, 0x05, 0x06, 0x07, 0x04, 0x05, 0x06, 0x07, \
    0x04, 0x05, 0x06, 0x07, 0x04, 0x05, 0x06, 0x07 }
#define SPLAT_THIRD    (vec_uchar16)
  { 0x08, 0x09, 0x0a, 0x0b, 0x08, 0x09, 0x0a, 0x0b, \
    0x08, 0x09, 0x0a, 0x0b, 0x08, 0x09, 0x0a, 0x0b }                10
#define SPLAT_FOURTH (vec_uchar16)
  { 0x0c, 0x0d, 0x0e, 0x0f, 0x0c, 0x0d, 0x0e, 0x0f, \
    0x0c, 0x0d, 0x0e, 0x0f, 0x0c, 0x0d, 0x0e, 0x0f }
#define ROT_RIGHT_ONE (vec_uchar16)
  { 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, \
    0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13 }
#define ROT_RIGHT_TWO (vec_uchar16)
  { 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, \
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17 }
#define ROT_RIGHT_THREE (vec_uchar16)                                   20
  { 0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13, \
    0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b }
```

over the available SPEs so that they can all work at the same time. It is also important that the distribution is not made in such a way that it blocks any SPE from working (more then necessary). As discussed in section 4.4 there are several ways to parallelize correlation. The method we choose is the one based on the partitioning of the main loop.

5.4.1 Calculating the distribution

The calculations for the data distribution is done on the PPE. It is based on the max-lag that the user gives at run time. First we calculate the number of times the SPE data buffer is completely filled. Then we calculate the length of the last, partly full, load. Both these calculations includes the overhead discussed in section 4.10.

If, for example, each SPE can hold 400 samples, plus overhead and work, in the LS data buffer and we have 8 SPEs calculating with a max-lag of 4500 we will do one full load ($400 * 8 = 3200$ samples) and one partly full load of $1300/8 = 162,5$ on each SPE. Since it is wasteful to do uneven workloads on the SPEs we increase the max-lag so that the partly full load length always is a number evenly divisible with the number of SPEs. In the previous example the max-lag is increased to 4504 and the last partly full load will be 163 instead. The code for this is shown in listing 5.4.

5.4.2 Parallel SPE code

Since the distribution of work is done on the PPE and the work requires no mutex while working the parallel SPE code is the same as code for a single SPE.

Program listing 5.4 Data partitioning for the SPEs

```
/* maxlag can't be 0 */
    if (maxlag==0)
        maxlag = 1;
    /* Calculate the number of times we need to send data
     * to the SPEs for every new sample */
    fullSpeDataLoops = maxlag /
        ((SPE_DATA_BUFFER - SAMPLES_CALCULATED_PER_LOOP)
        * speThreads);
    /* How much data remains after the full loops */
    partSpeDataLength = maxlag - fullSpeDataLoops *                 10
        (SPE_DATA_BUFFER - SAMPLES_CALCULATED_PER_LOOP)
        * speThreads;
    /* If this data is not an even multiple of the number of SPEs */
    if (partSpeDataLength % speThreads)
        /* Increase the maxlag */
        maxlag +=speThreads-partSpeDataLength % speThreads;
    /* How much data remains after the full loops
     * with the new maxlag*/
    partSpeDataLength = (maxlag - fullSpeDataLoops *
        (SPE_DATA_BUFFER - SAMPLES_CALCULATED_PER_LOOP)             20
        * speThreads)/speThreads;
```

5.5 Comparison of the different implementations

Table 5.2 shows the results from the different implementations. The speed is given in GFLOPS based on the number of calculations needed in theory (see section 4.3.1) and compared to the theoretical maximum (t-max). The reason for the falling percentage of maximum performance is mostly due to the lack of buffered loads on the SPEs.

Implementations	GFLOP	% of t-max
Scalar PPE	0,15	0
Vector PPE	1,46	3
1 SPE	9,16	36
6 SPEs	57,74	38
8 SPEs	76,99	38

Table 5.2: Results from different implementations compared to the theoretical maximum (t-max)

Chapter 6

Description of the final application

6.1 Outline

The application created is named *The Cell BE time domain correlator*. It calculates the correlation of the different antenna signals using equation 2.6. The end result is the matrix defined in equation 2.2 although the individual cells are in different files.

The correlator application creates several threads to be able to perform multiple tasks simultaneously. One thread is created for each available SPE, one to manage the data loading and one to control the SPE execution. The application structure is outlined in figure 6.1.

6.1.1 Deviation from the definition

The application diverges from the results from equation 2.6 in two ways:

6.1.1.1 Ignoring max-lag start data

Because of the way that the SCA works we need max-lag old samples stored (see section 4.2.2). This means that we ignore the first max-lag samples in the data file and start correlating on sample max-lag+1. This is actually beneficial since it means that the results from two consecutive correlations can be added together to get the result of a longer time window.

6.1.1.2 Ignoring the end data

Because the application correlates data in chunks of max-lag (because the limitations in transfer rates) the application only correlates data with a length that is a multiple of the number of samples calculated per loop. Any reminding data is ignored.

6.2 Structure

The implementation of the application is divided in several parts to simplify and structure the development. The PPE side of the code has the following parts:

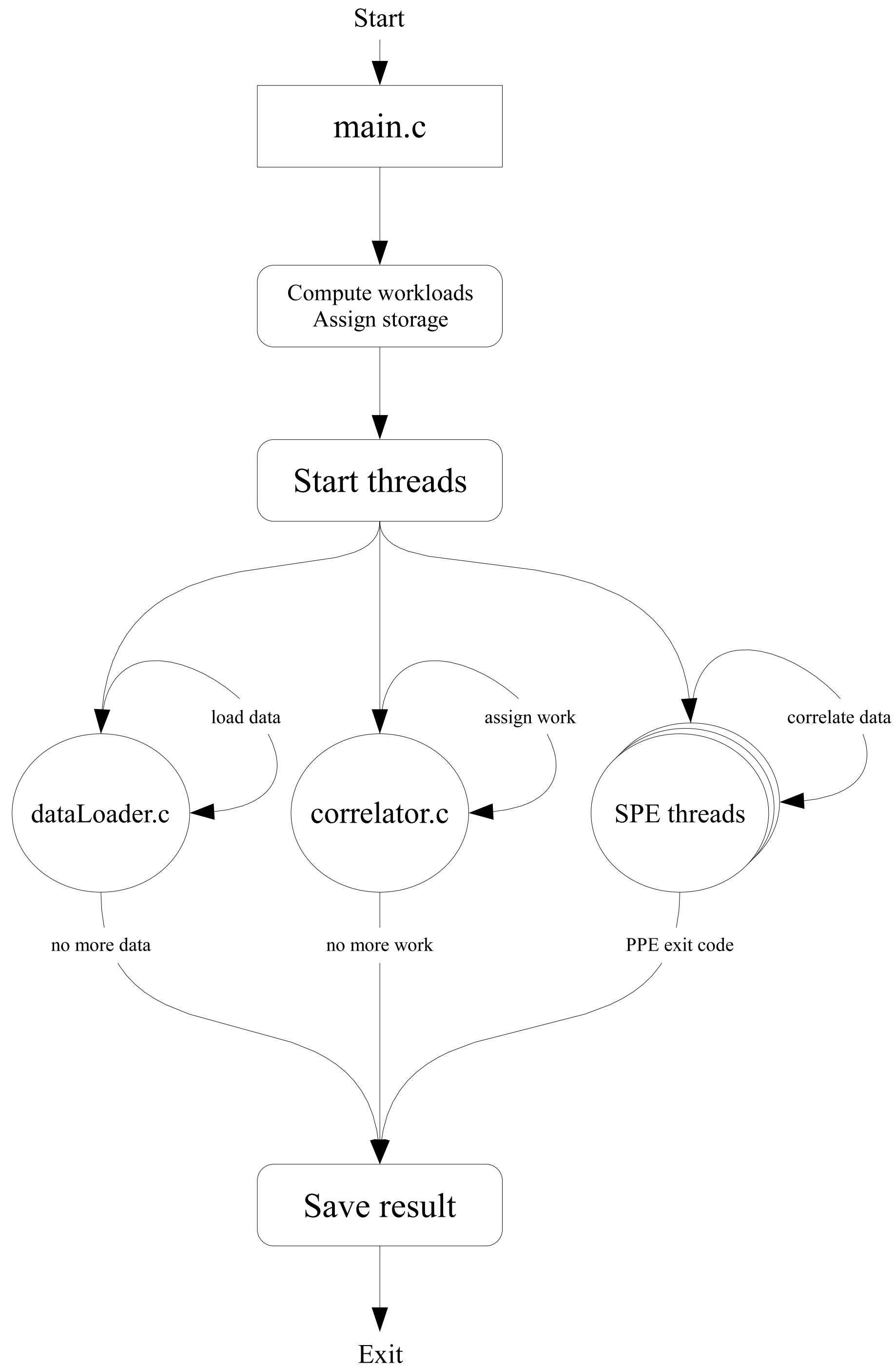

Figure 6.1: Program flow

1. Main: Containing setup, allocation, thread starting and results gathering
2. Data loader: Provides the data, in this case from the hard-drive
3. Data saver: Saves the result to hard-drive
4. Correlator: Manages the SPE execution
5. Structures: Contains the structures for different data types
6. Definitions: Contains definitions that can be changed by the user prior to compilation to change the behavior of the application

The SPE side of the code only have one part. The following sections describes the parts in more detail.

Both the PPE and the SPE programs also have support functions for timing. The SPE timing and the timebase function is taken from the open source IBM SDK examples and the PPE timing is based on the *cpucycles* application by D. J. Bernstein [15].

6.3 Global values

Defined globally, that is accessible to all functions, are the circular data buffer and the result arrays. There are three global index values that points to the data buffer dividing it into three parts.

- The first part is used to store the samples needed for the correlation and is of max-lag length. The index *i_data* corresponds to the index of the first sample in this part.
- The second part contains the uncorrelated samples waiting to be correlated indexed by *i_work*. The length of this part must be at least the same as the number of samples calculated in each SPE loop but a larger number allows the loads to be buffered.
- The last part is used for the loading of new samples and is indexed by *i_free*. The longer this part is the more samples are loaded for each load operation. A larger number reduces the overhead / sample and speeds up the loading.

As an example of this we can use figure 4.2. The read arrow in the figure can be *i_data*, the blue arrow the *i_work* and the green arrow *i_free*. We would then have five samples of stored data, seven samples of future work and four unused positions to load data in.

Also defined globally is the *dataLeftToLoad* integer which is non-zero while there is data to load (it has to be declared volatile so it is constantly checked and not cached) and other integers that are used to pass data to externally defined functions.

6.4 PPE main

The main contains the start point for the application and the global variable definitions. It starts by checking that the structures are correctly padded, a requirement for the indexing (see section 5.3.1) and then retrieves the user given max-lag. This value is the changed to fit the workload for the SPEs (as described in section 5.4.1).

After the max-lag has been determined the data buffer length is calculated. The length of the data buffer must be at least the size of the max-lag plus the load and work buffers. This length must be a power of two to reduce the operations needed to calculate the modulo (see section 4.8) so the next highest power of two values is taken.

The function then dynamically assigns storage for the circular data buffer and the result arrays aligning the start address on a 128B boundary (see section 3.5). Then the SPE context is created and the SPE program is loaded. The threads, one for each SPE, one for the data loader and one to manage the correlation, are started. A barrier is used to wait for all started threads to finish and after that the result is saved and all resources are freed.

6.5 Data loader function

The current implementation of the data loader uses files on the hard-drive (see section 4.7). The function first checks that there is enough space allocated in the buffer to support the defined load size. Then the data file is opened and the function enters a loop which runs until there is no more data in the file. Every time the function starts a new loop it checks if there is enough space in the buffers for a load. If there is not, which is usually the case, the function puts its thread to sleep for a defined time. If there is enough space the function checks if it can make a full load or if it has to split the loads in two parts to wrap the buffer. This is because the *fread*[1] function used for the loads does not support circular buffers.

All the time the amount of data returned from the *fread* is checked against the requested length. If less data is returned the function checks if the end of the file (EOF) is reached. If it is the function changes the *dataLeftToLoad* integer to zero, updates the *i_work* index with the new data and exits the thread. If EOF is not reached the function assumes that an error has occurred and sets the *dataLeftToLoad* integer to zero and exits with an error.

6.6 Correlator function

The correlator function starts up by checking that there is enough initial data, both max-lag old data and enough work data for a loop, to start the calculations (see section 6.1.1). It puts its thread to sleep until there is, periodically checking back. When enough data is loaded the function enters its loop. The loop first checks to see if there is enough data to correlate the defined number of samples. Otherwise it will put its thread to sleep and wait for the data loader. If there is enough data the function starts assigning work to the SPEs.

The function uses the distribution calculated in the PPE and sends four messages to each SPE containing the address of the data to load, the length of data to load, the address of the work (same for all SPEs) and the offset in the result arrays that corresponds to the data used in the calculation. The function then blocks until all SPEs are finished. If several transfers per sample loop is needed this process is repeated.

After all SPEs are done the function updates the *i_data* and the *i_work* index to correspond with the new positions and then returns to the top of the loop. This goes on until the *dataLeftToLoad* integer is set to zero and there is not enough data in the work buffer for a new calculation (the remaining data is not correlated, see section

[1]Included with the C stdio.c library

6.1.1). The function then sends the predefined exit signal to the SPEs and terminates its thread.

6.7 Data saver function

The data saver is a simple function that saves data from a general array to the hard-disk using binary format.

6.8 Structure file

The structure file defines the layout and size of the structures used in the application. As stated in section 5.3.1 the size of the data type used for the data buffer and result array must be a multiple of 128B. This might add overhead in the transfers but it should still be faster then a lesser alignment if the overhead is not excessive.

6.9 Definition file

In order to create a flexible application many variables are defined in the definition file. The file contains definition flags to enable debugging, timing and profiling of the code. It also specifies the buffer sizes, the number of samples correlated per loop in the correlator function (and thus on the SPEs) and sleep times for the threads. It contains file paths for the data loader and saver and defines the maximum SPEs that the application can use. It also defines the value used to stop the SPE calculations.

6.10 SPE main

The SPE program is in one file and consists of two parts both running in the same infinite loop. Before the loop the start up arguments are loaded from main memory and then the SPE waits for the first command from the PPE. This command is made up of four values. The index to the stored data, the length of data to load, the index to the work data and the offset in the result array.

The loop starts by issuing DMA commands to load the old result, the stored data and the work[2]. When this is done the correlation starts. The correlation is quite long but many parts are similar, so to avoid errors and help in the development the similar pars are defined with macros that the compiler replaces with the actual code at compile time. The correlation is done in two loops, one internal over the SPE data buffer and one external over the amount of work sent. After the correlation is done the result is sent back and the SPE waits for the next command.

The loop continues until the PPE sends the predefined exit number instead of a work address. This will make the SPE exit the loop and finish the program.

[2]This DMA should be made buffered and done with DMA lists in future versions

Chapter 7

Conclusions

7.1 Source code

The source code for the application is registered under the open source BSD license and is free to use and redistribute according to the license. It is available at
`http://tdccellbe.cvs.sourceforge.net/tdccellbe/tdc/`

7.2 Performance

The performance of the final version of the application was shown in section 5.5 and is close to 40% of the theoretical maximum. The application uses a limited amount of data due to the way TDC was implemented (see section 4.2.1) and the PPE is running on only 10-16% of its maximum leaving the rest to other applications.

7.3 Possible improvements

The performance stated above differs from the theoretical maximum stated in section 4.3.1 by 60% Part of this is because the unbuffered DMA and part is because the SPE code is not optimized. It can be assumed that is possible to reach the maximum given enough time using the methods stated in section 4.6, mainly using assembly level unrolling and scheduling. This assumption is based on similar problems being solved with near maximum performance (as stated in section 4.3.1).

7.4 Related work

We are not aware of any other TDCs for the Cell BE. In the frequency domain there is a project to build an FFT based correlator for the Cell BE ongoing at the Metsähovi Radio Observatory. It will be based on the DiFX Very Long Baseline Interferometry correlator but the design is not as general as the correlator we are building so comparisons are not easy.

7.5 Comparison with FFT applications

Since no other TDC exists we instead evaluate the performance of our implementation by comparing it to the performance of FFT applications using the method described in section 7.5.1 below. However it is hard to make a straightforward comparison as an FFT does things differently then our implementation of the TDC. Our application is streaming giving it an unlimited data length (until the application stops) but a limited max-lag. It also calculates the correlation based on an unlimited signal (see section 6.1.1). An FFT based implementation will be limited in data length (see section 2.5.2) but will always calculate the maximum lag for that data length (that is $M=N$). It also correlates under the assumption that the data is finite.

There are possibly methods for Fourier transforming large data with a limited FFT (perhaps with a version of the overlap and add method) but in order to keep these comparisons simple we assume, to the FFTs advantage, that we are dealing with a limited data length. We also assume in most cases that the max-lag calculated is the maximum N instead of M.

7.5.1 FFT method

To do a correlation on the antenna signals in the Fourier domain we have carry out the following steps:

1. Transform the complex antenna channels (one per axis and antenna) to the frequency domain
2. Multiply the converted channels with the complex conjugate of all the other channels
3. Inverse transform the result of the multiplication back to time domain

We assume that a FFT correlation of a signal of length N requires that the signal is padded with zeros to twice its length order to avoid cyclic convolution. This means that in order to correlate the data from one three-axis RVFS antenna we would need (using the formula from section 2.5.1) $3*5*2N\log_2 2N$ FLOP for the frequency transforms, $9*4*2N$ FLOP for the multiplications[1] and then $9*5*2N\log_2 2N$ FLOP for the inverse transform. This gives a total of $120N\log_2 2N+72N$ FLOP, the $72N$ multiplications are assumed to be done at 200GFLOP. Using this formula we deuce the number of samples calculated per second.

7.5.2 Comparison

The speed of the TDC used in this comparison is 200 GFLOP instead of the current speed of the application. We feel that it is better to use this value as it represents a possible speed of the Cell BE better. We have chosen three different FFTs and use the data length that they give the best results for. The FFT implementations are

1. FFTC described in [4] running at 18,6 GFLOPS on 8ki input and 21 GFLOPS on 16Ki input
2. FFTW described at [3] running at 22,5 GFLOPS on 128Ki input

[1] Complex multiplications using the fused multiply and add

3. The large FFT described in [5] running at 46,8 GFLOPS on 16Mi input

Table 7.1 shows the sample rate for different FFT implementations and input lengths on using the formula from above. The value is compared with the sample rate for one TDC with a max-lag of N and one TDC with a max-lag of 70000 (TDC-L) where applicable.

	N	M sample/s	TDC M samples/s	TDC-L M samples/s
FFTC	4096	11,87	1,36	-
FFTC	8192	12,44	0,68	-
FFTW	65536	10,99	0,08	-
FFTW	524288	8,31	0,01	0,08
LargeFFT	8388608	16,16	~0	0,08

Table 7.1: Comparison of different FFT implementations against the TDC using maximum max-lag and a TDC with a max-lag of 70000 (TDC-L)

7.5.2.1 Memory usage comparison

Looking at table 7.1 one might wonder why there is any need to use FFTs of shorter lengths, this is because the table does not show the memory requirements for each implementation. If we again use one three-axis RVFS antenna a FFT application needs to store 12 complex arrays of length 2N (the 2 comes from the zero padding) in main memory. The TDCs storage space is described in section 4.9 and is limited by the max-lag.

Table 7.2 shows the memory requirements for different FFT lengths compared with the memory requirements for TDC using maximum max-lag (TDC) and an implementation of the SCA using a max-lag of 70000 (TDC/SCA-L). It can be seen the the 16Mi point FFT requires a lot of memory. In fact it exceeds the total memory possible on both the PS3 and the QS20, only allowing it to be used on the QS21 (one can of course save the data to disc but then the execution speed becomes limited by the I/O transfer speed). The TDC-L on the other hand requires a constant and very small amount of memory.

2N	FFT	TDC	TDC/SCA-L
4096	0,75MiB	0,38MiB	-
8192	1,5MiB	0,75MiB	-
65536	12MiB	6MiB	-
524288	96MiB	48MiB	6,41MiB
8388608	1536MiB	768MiB	6,41MiB

Table 7.2: Comparison of memory requirements for different FFT input lengths against the memory requirements for the TDC using maximum max-lag and a an implementation of the SCA with a max-lag of 70000 (TDC/SCA-L)

7.5.3 Conclusion of the comparison

It is obvious (and not surprising) that the correlation using FFTs are faster. The purpose of this work is not to make the fastest correlator but to try an TDC. It it still interesting

to see that with a limited max-lag the TDC/SCA is capable of correlating an unlimited signal in real time using only a limited amount of storage space. This is not easy to do with an FFT. It should also be noted that the FFTs add rounding errors in their calculations something that the time domain correlator does not (apart from what was mentioned in section 4.9.2).

7.6 Programming the Cell BE

Starting to program for the Cell BE feels like a daunting task. The processor is different from what one is used to and new methods needs to be learned. After getting more familiar with the processor this fades away more and more and it is obvious that what before seemed hard or impossible is doable without to much effort. There are three things that gives the most problems for a beginner.

7.6.1 Memory

First is the way the SPEs handle memory, making the programmer responsible for cache management, something that many are unaware of or ignores. The design for an application on the Cell BE should be aware of this design the code so that it suits the memory model. For an experienced high performance programmer this is not uncommon ground and the added control over the memory latency is a great tool. The memory system on the Cell BE can allow an applications to run as if each SPE had the whole main memory at cache like latency, something that can give very good performance. The only limitation to this is the memory transfer rate which, although very high, is not unlimited.

7.6.2 Parallelism

The second problem is to turn the mind from scalar to parallel thinking. This is a task that face almost everyone struggling for performance now days as most processors features two or more processor cores. On the Cell BE this is take a step further and running an application on only one core is not a good option. There are many ways to make an application parallel so usually this is not a big problem. The aim should be to create an application that scales well with an increasing SPE count and does not waste computation power having idle SPEs.

7.6.3 Optimization

Because the Cell BE can hide much of the memory latency optimization of the code can be more important then on other architectures. The way instructions are issued to the pipeline also means that extra care should be spent on instruction scheduling. Add to this the lack of advanced branch prediction and it is clear that much performance can be gained on the Cell BE from code optimization. Although compilers can do much they are currently not at level with a good programmer. It should be noted that the Cell BE does not perform badly on unoptimized code, it is only that it will preform so much better when optimized.

7.7 Cell BE as a time domain correlator

We have shown in here that the Cell BE is very suitable as a time domain correlator allowing for a near maximum performance. The time domain correlation is still slower then the frequency domain version even when they are running on much less FLOPS but it is still capable of performing its task. Using the SCA the implementation only uses a small amount of main memory, something that is expensive in space.

Acknowledgments

I would like to thank my supervisor, Dr Jan Bergman, and my assistant supervisor, Mr Lars Daldorff, both at the Swedish Institute of Space Physics (IRF) in Uppsala for their help and support. I also would like to thank Dr Olivier Verscheure at the IBM T.J. Watson Research Center in Hawthorne, New York, for his scientific review of my work, Dr Erling Weibust and Mr Niklas Dahl at IBM Sweden, for providing me with access to an IBM BladeCenter QS20, and Professor Bo Thidé and the staff of IRFU for answering questions and providing support. Finally I would like to thank my wife Hedvig Holmberg for proofreading my work and my seminar opponent Mr Magnus Broberg for pointing out improvements and clarifications. The project has been financially supported by IBM through two Shared University Research grants.

Bibliography

[1] Cell Broadband Engine Architecture, version 1.01, IBM Technology Group Library, 03 October 2006

[2] J. A. Kahle et al: “Introduction to Cell Multiprocessor”, IBM J. Res. & Dev. Vol. 49 No. 4/5 July/September 2005

[3] FFTW: Fastest Fourier Transform in the West library, see http://www.fftw.org/

[4] D.A. Bader, V. Agarwal: “FFTC: Fastest Fourier Transform on the IBM Cell Broadband Engine”, 14th IEEE International Conference on High Performance Computing (HiPC 2007), Goa, India, 18-21 December 2007.

[5] A. Chow, G. Fossum and D. Brokenshire: “A programming example: Large FFT on the Cell Broadband Engine”, IBM Technology Group Library, 2005.

[6] For LOFAR project information see http://www.lofar.org/

[7] Cell Broadband Engine Programming Handbook, version 1.1, IBM Technology Group Library, 24 April 2007

[8] J. Rentzsch: “Data alignment: Straighten up and fly right”, IBM developerWorks Library, 08 February 2005, http://www.ibm.com/developerworks/library/pa-dalign/

[9] R. Karlsson, W. Puccio, J. Bergman, T. D. Carozzi, and B. Thide: “Three-channel digital radio vector field sensor: description and demonstration”, Proc of the Nordic Shortwave Conference HF-04, Fårö, Sweden, 10-12 August 2004

[10] J. Bergman, T. D. Carozzi, and R. Karlsson: “Present and future applications of information dense antenne”, Proc. of the Nordic Shortwave Conference HF 04, Fårö, Sweden, 10–12 August 2004.

[11] For the Roadrunner project see http://www.lanl.gov/roadrunner/

[12] T. Chen, R. Raghavan, J. Dale and E. Iwata: “Cell Broadband Engine Architecture and its first implementation”, IBM J. Res. & Dev. Vol. 51 No. 5 September 2007

[13] Cell Broadband Engine Programming Tutorial, version 3.00, IBM Technology Group Library, 2007

[14] G. Kraszewski: “Fast FIR Filters for SIMD Processors With Limited Memory Bandwidth”, XI Symposium AES "New Trends in Audio and Video", Bialystok, Poland, 20–22 September 2006

[15] The eBats cpucycles benchmark is located at http://www.ecrypt.eu.org/ebats/cpucycles.html