

Copyright

by

Thomas Higdon

2008

The Dissertation Committee for Thomas Higdon
certifies that this is the approved version of the following dissertation:

**The Implementation of a Sonar Beamformer on the Cell
Broadband Engine**

Committee:

Brian L. Evans, Supervisor

Derek Chiou

**The Implementation of a Sonar Beamformer on the Cell
Broadband Engine**

by

Thomas Higdon, BS

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2008

The Implementation of a Sonar Beamformer on the Cell Broadband Engine

Thomas Higdon, MS
The University of Texas at Austin, 2008

Supervisor: Brian L. Evans

Modern, high-resolution sonar systems have prohibitive data processing requirements. Sonar beamforming, in particular, has computing requirements that challenge even the latest offerings from industry. Therefore, advanced hardware and optimized software are necessary to satisfy the requirements of this application. A sonar beamformer implementation for a heterogeneous nine-core architecture, the Cell Broadband Engine, is described. The implementation leverages existing algorithms and techniques in a concurrent fashion to effectively utilize the power of the platform. The implementation exhibits performance of 38 GFlops, 25% of peak performance. The existing prototype, developed on a Sony PlayStation 3, readily outperforms an efficient implementation on an Intel Xeon platform, with a significant savings in cost, power, and space.

Contents

Abstract	iv
Contents	v
List of Tables	viii
List of Figures	ix
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Summary	3
Chapter 2 Background	4
2.1 Sonar Signal Processing	4
2.1.1 Introduction to Beamforming	4
2.1.2 Filtering with the DFT	7
2.1.3 Circular Beamforming	8
2.2 The Cell Broadband Engine Platform	10
2.2.1 Cell Architecture	11
2.2.2 Precursors to the Cell	12

2.2.3	Communication Facilities	15
2.2.4	The PlayStation 3	16
Chapter 3	Beamformer Implementation	19
3.1	Implementation Overview	19
3.2	Distributed Matrix Transposition	23
3.2.1	Prior Work	23
3.2.2	Proposed Method	24
3.3	Synchronization	29
3.4	Double-buffering	32
3.5	DFT Filtering Considerations	34
3.6	Optimizations	39
3.6.1	Large TLB Pages	39
3.6.2	FFT Software	41
3.6.3	Efficient Memory Bank Utilization	43
Chapter 4	Implementation Results	46
4.1	Cell Processor Results	46
4.1.1	Speedup	47
4.1.2	Calculation versus Data Transfer Time	49
4.2	Comparison with an Intel-based Implementation	52
Chapter 5	Conclusion	55
	List of Acronyms	56
	Bibliography	57

List of Tables

2.1	A comparison of previous similar processors (where “Gops/s” is defined as “billions of integer operations per second” with a multiply-accumulate counting as two distinct operations)	13
2.2	A comparison of the sales figures of major game consoles in the period from each console’s respective launch to the end of July 2007 (the end of August in Japan). The PlayStation 3 was introduced in late November of 2006.	17
3.1	A comparison of the filter size with its corresponding optimum Fast Fourier Transform size and ratio of N/M	38
4.1	Illustration of performance and speedup by varying number of Synergistic Processing Elements	47
4.2	Illustration of performance and speedup by varying number of CPU cores utilized in the Intel-based beamformer implementation	52

List of Figures

2.1	Illustration of element delays required to localize a point source given a linear array of elements.	5
2.2	Simplified block diagram of a digital interpolation beamformer.	6
2.3	Illustration of element delays required to localize a point source given a circular array of elements.	9
3.1	Alternating sensors in the actual sonar array.	20
3.2	Naive proposed implementation of the beamformer.	21
3.3	Proposed implementation of the beamformer.	23
3.4	Inefficient matrix transpose.	26
3.5	Matrix transpose on the SPE (Synergistic Processing Element) using multiple rows.	26
3.6	Graph of the time required for a barrier for two different synchronization methods.	31
3.7	Double-buffering flow chart	32
3.8	Double-buffering illustration	33
3.9	Double-buffering illustration with writing	34
3.10	Comparison of huge pages vs. malloc	41

3.11	Difference in performance caused by efficient memory bank utilization	45
4.1	Speedup obtained for varying the number of Synergistic Processing Elements in the beamformer	48
4.2	Time required for calculation and data transfer sections for a varying number of Synergistic Processing Elements for Stage 1 of the implementation.	50
4.3	Time required for calculation and data transfer sections for a varying number of Synergistic Processing Elements for Stage 2 of the implementation.	50
4.4	Time required for calculation and data transfer sections for a varying number of Synergistic Processing Elements for Stage 3 of the implementation.	51
4.5	Time required for calculation and data transfer sections for a varying number of SPEs for all stages aggregated.	51
4.6	Speedup obtained for varying of CPU cores utilized in the Intel-based beamformer implementation	53

List of Acronyms

ALU Arithmetic Logic Unit

ASIC Application-Specific Integrated Circuit

CPU Central Processing Unit

DFT Discrete Fourier Transform

DMA Direct Memory Access

DSP Digital Signal Processor

EIB Element Interconnect Bus

FFT Fast Fourier Transform

FIR Finite Impulse Response

FSB Front Side Bus

HPC High-Performance Computing

IBM International Business Machines

LS Local Store

MFC Memory Flow Controller

MMU Memory Management Unit

MP Master Processor

MRA Maximum Response Angle

PP Parallel Processor

PPE Power Processing Element

PTE Page Table Entry

PS3 PlayStation 3

RAM Random Access Memory

RCF Reconfigurable Compute Fabric

SDK Software Development Kit

SIMD Single Instruction Multiple Data

SMM Synergistic Memory Management

SMT Simultaneous Multithreading

SPE Synergistic Processing Element

SSE Streaming SIMD Extensions

TI Texas Instruments

TLB Translation Lookaside Buffer

VMX Vector Multimedia eXtension

Chapter 1

Introduction

1.1 Motivation

The ever-increasing resolution of sonar systems in both the time and space dimensions means that more powerful computer systems are needed to process and interpret these increasing amounts of data. One sonar data-processing step, called beamforming, is particularly math-intensive, and is therefore often a bottleneck. Sonar system developers devote much time and expense to developing computer systems capable of executing this step quickly and efficiently. Important design variables include power consumption, cost, and rack space, although the main constraint is that a system must process sonar data in real-time.

The Cell Broadband Engine, the result of a joint development effort by IBM, Toshiba, and Sony, was introduced over a year ago. This CPU excels at floating point performance, with a theoretical peak of 200 GFlops. The Cell also has excellent memory bandwidth, around 25 GB/s. Both of these characteristics are very important for beamformers, which require good floating point performance and memory bandwidth.

This strength in floating point performance and memory throughput makes the Cell very suited for sonar beamforming applications. The Cell is a heterogeneous processor, meaning it is made up of multiple processing cores that are not necessarily the same. The Cell uses a Power Processing Element (PPE), which has a PowerPC-like instruction set, along with eight SPEs (Synergistic Processing Elements), which have a SIMD (Single Instruction Multiple Data) instruction set. Each SPE has no direct access to main memory, and must use its own local store for data and program storage, while using explicit instructions to trade data with the PPE. This architecture allows the Cell to realistically operate closer to its peak performance than traditional homogeneous architectures currently used in many sonar applications.

The intricate architecture of the Cell and relative newness of the programming tools available means that exploiting the full performance capability of the chip requires a low-level approach. Presented here is a beamforming algorithm that exploits the power of this processor, while conforming to the constraints of a given sonar system. The algorithm uses a variety of techniques to implement an efficient beamformer on the Cell. Careful data layout, efficient data movement among the processing elements, and synchronization among the elements will be discussed, along with methods for FFT calculation using available tools and a distributed matrix transposition.

The proposed implementation is capable of beamforming in real-time in a given sonar system using a single Cell processor, where the same operation requires four or more latest-generation Intel Core 2 Duo Xeon processors based on the x86-64 architecture. Fewer CPUs means fewer rack-mounted servers are required, meaning lower power, less maintenance, and less rack space. The ability to use a single CPU

for this application is a significant cost, space, and power savings.

1.2 Summary

Chapter 1 provides an introduction to this report. Chapter 2 of this report will discuss some of the background required to discuss this beamformer implementation. Section 2.1 will give the reader an introduction to sonar beamforming, discuss filtering using the discrete Fourier transform (DFT), and describe the circular beamforming algorithm used by this implementation. Section 2.2 discusses the Cell Broadband Engine, the platform which is used in this implementation. The architecture of the processor, previous similar architectures, and the actual development system, the PlayStation 3, will be discussed. Chapter 3 discusses the beamformer implementation that is the subject of this report. The implementation is first described at a high level, and then various facets are shown at a deeper level. The method of matrix transposition, synchronization constructs, filtering with the DFT, and performance optimizations are all discussed. Chapter 4 discusses the performance results of the implementation, and contrasts its performance on the Cell processor with a similar implementation on an Intel-based system. Chapter 5 concludes the report.

Chapter 2

Background

2.1 Sonar Signal Processing

2.1.1 Introduction to Beamforming

The objective of a submarine sonar system is to gain information about the area around a vessel. Underwater, visual information is very limited and electromagnetic waves at useful frequencies attenuate very quickly, so we are left only with acoustical means to achieve this goal.

Beamforming is a technique used by sonar systems to specify a direction of focus to an array of sensors, called hydrophones. In the simplest case for a linear array of sensors, this will simply correspond to a delay and weighting in each sensor such that a source, modeled as a plane wave, will appear to propagate to each sensor at the same time. The output of each sensor is then summed together. This is illustrated in Figure 2.1.

The conventional beamforming equation is given by [15]:

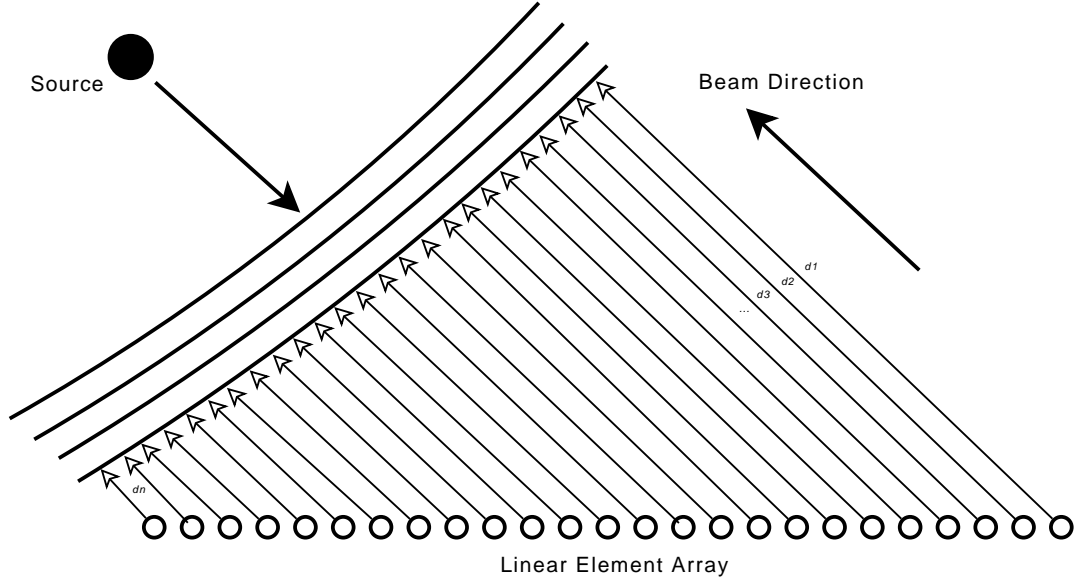


Figure 2.1: Illustration of element delays required to localize a point source given a linear array of elements.

$$b(t) = \sum_{n=1}^{N_E} a_n x_n(t - \tau_n) \quad (2.1)$$

N_E denotes the number of sensors, x_n denotes the sensor output of sensor n , a_n denotes the constant gain applied to the sensor output and τ_n denotes the time delay. The result of this equation is called the beam, and the angle to which the beam is focused is called the Maximum Response Angle (MRA). One can see that this beamforming operation can be modeled as a series of finite impulse response filters of each x_n .

To attack this problem digitally, the sensor output must be sampled. Specifically, the sensors must be sampled at a rate greater than the Nyquist frequency in order to allow accurate reconstruction later. Given a low-pass signal with a bandwidth of B_w , the Nyquist rate would be $2B_w$. This is a reasonable rate for

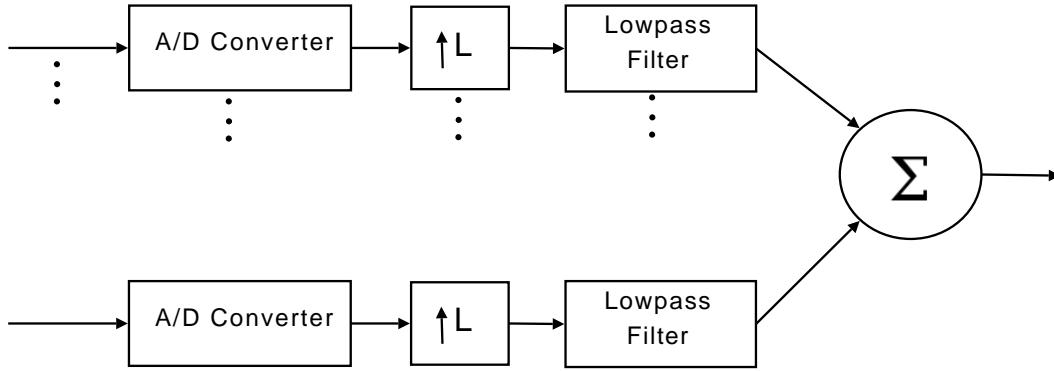


Figure 2.2: Simplified block diagram of a digital interpolation beamformer.

the A/D converters and connective cabling in typical sonar systems [13]. However, the time delays we wish to introduce in the beamforming process are often much smaller than $1/B_w$. In order to represent these delays, a much higher sampling rate is required. This requirement is the motivation for a technique called digital interpolation beamforming [15].

Figure 2.2 is a typical block diagram of a digital beamforming system. We wish to reduce the sampling rate of the signals entering the A/D converter, yet still allow the time delays to operate at the resolution required for beamforming. To do this, it is necessary to insert an upsampler followed by a low-pass filter between the multiplier and the time delay. The upsampler will resample the signal at the rate required by the beamformer time delays, and the low-pass filter will interpolate the signal. The low-pass filter will have a cutoff frequency of π/L , where L is the up-sampling factor needed to satisfy the delay resolution requirements of the beamformer [16].

2.1.2 Filtering with the DFT

Since the beamforming operation following the upsampler can be represented as an FIR filter, the possibility is presented of doing the calculations in the frequency domain instead of the time domain. Since Cooley and Tukey presented a fast way to compute the frequency domain representation of a digital signal [2], calculations of this sort are frequently done in the frequency domain for significant computational savings [25]. In our case, because we have a continuous stream of data, we must use a linear filtering method tailored to the DFT, such as overlap-save [16].

While linear filtering on a long signal in the time domain using a FIR filter is a trivial process, filtering the same type of signal using the DFT is a little more complicated.

The discrete convolution equation,

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k], \quad (2.2)$$

shows that in the time domain, each $y[n]$ is calculated from the weighted summation of the points in the time-domain filter. Therefore the calculation of each output point is independent, meaning the calculation of a arbitrarily long set of data points is simple.

Since a DFT operates only on discrete blocks of a signal, it is clearly impossible to calculate the filter output of a particular point independently. Using the DFT, a technique like overlap-save is required. Take M to be the length of the FIR filter, and L to be the length of an arbitrary segment of data, such that $L \gg M$. Also, let $N = L + M - 1$, where N is the DFT length of blocks to be processed. Each block that enters the algorithm consists of the last $M - 1$ points from the previous block concatenated with the next L points from the next block.

The beginning $M - 1$ points of the first block are simply zeroes. At each step, the N -point DFT of each block is computed and multiplied by the N -point DFT of the FIR filter padded with $L - 1$ zeroes. The inverse DFT is applied to this result, and the first $M - 1$ points of the output are discarded due to aliasing. The remaining L points are the time-domain result of the linear filtering operation. The L points from each successive step of the algorithm are concatenated together to form the final output of the filtering operation.

The reason that $L - 1$ points must be discarded each time is related to the fact that the DFT is not completely analogous to convolution in the time domain, as the actual Fourier transform is. A DFT operation in the frequency domain is actually equivalent to a circular convolution in the time domain, meaning that points at the beginning of the signal are corrupted with respect to the linear convolution. Since this corruption in circular convolution is actually due to an additive effect, this suggests another well-known DFT filtering technique called overlap-add. This method is arguably less efficient, and will not be described here.

2.1.3 Circular Beamforming

The previous discussion concerned a beamformer that used sensors arranged into a linear array. However, a circular array has some desirable characteristics. One is that we can form a beam in any direction, without the requirement of mechanical array movement (see Figure 2.3). Another is that the variation in beam width as the beam is rotated about the array is reduced [24]. As well, it turns out that the beamforming operation itself can be made a great deal more efficient.

In the previous section, the operations described provided the means to form a single beam given a set of shading coefficients and delay values. In a real sonar

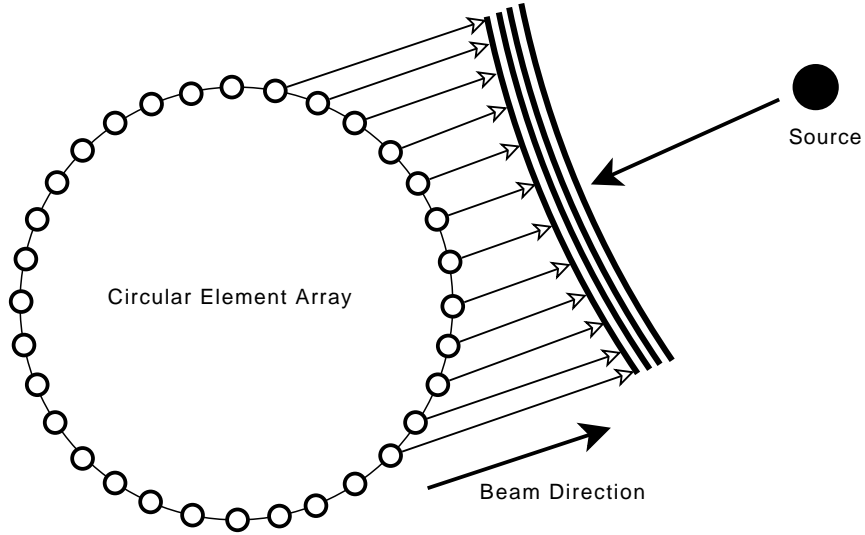


Figure 2.3: Illustration of element delays required to localize a point source given a circular array of elements.

system, however, a large number of beams is required. In order to be useful, a sonar system should be able to distinguish objects in a large number of directions, and preferably without mechanical movement, to decrease cost and mechanical complexity. The DFT provides a means to do this efficiently [18].

Farrier *et al.* observed that a circular beamforming operation can be likened to a two-dimensional DFT in time and space [5]. Let $\alpha_0[k] \cdots \alpha_{N-1}[k]$ be the sampled frequency domain representation of the weights required to steer a beam through the zero sensor, such that

$$Y_0[k] = \alpha_0[k]X_0[k] + \alpha_1[k]X_1\omega + \cdots + \alpha_{N-1}[k]X_{N-1}[k], \quad (2.3)$$

where $X_0 \cdots X_{N-1}$ are the frequency domain representation of the sensor outputs. Then it is easy to see that to steer the beam through sensor 1 requires that the weights are shifted by one element:

$$Y_1[k] = \alpha_{N-1}[k]X_0[k] + \alpha_0[k]X_1[k] + \cdots + \alpha_{N-2}[k]X_{N-1}[k] \quad (2.4)$$

Then

$$Y_l[k] = \sum_{j=0}^{N-1} X_j[k]\alpha_{j-l}[k], \quad (2.5)$$

where $\alpha_{-k} = \alpha_{N-k}$. One can see intuitively that this is a circular convolution operation, and as such, can be computed using the DFT. The final equation is

$$Y_l[k] = \sum_{m=0}^{N-1} X_m[k]b[k]e^{j2\pi ml/N}. \quad (2.6)$$

The full derivation is in [5].

Then for a circular array with equally spaced elements, a set of beams pointing outward from each element can be formed simply by computing an $M \times N$ DFT, where M is the number of sensors, and N is the number of points in time. This is a tremendous savings in computation, and a problem of this type can be efficiently calculated on today's microprocessors.

2.2 The Cell Broadband Engine Platform

The Cell Broadband Engine is a unique new microprocessor developed by a partnership between IBM, Sony, and Toshiba. It was designed from the ground up to have blindingly fast floating point performance as well as memory bandwidth, even at the expense of programming complexity. To further this goal, the development team decided to design it as a double-core architecture. They designed it with a PowerPC core, called the Power Processing Element (PPE), coupled with a set of

eight Synergistic Processing Elements (SPE), which are a novel architecture. The PPE and the SPEs communicate using the Element Interconnect Bus (EIB).

2.2.1 Cell Architecture

The PPE is simplified PowerPC core that is designed to serve as a manager of the SPEs. Its floating point performance is not particularly distinguished, although it does include a vector multimedia extensions (VMX) unit, a SIMD vector processing unit similar to Intel's SSE. It also has SMT, a facility that allows two threads to run independently with their own registers, although they must share execution units. The PPE, though, unlike the SPEs, is able to run modern operating systems, and therefore is suited to the task of delegating tasks to the SPEs, which serve as the workhorses of the architecture. IBM and Sony, in fact, have ported the Linux kernel to the architecture, as well as provided an extensive SDK free of charge to the public. The SDK even contains a full-fledged processor simulator. This effort has allowed the HPC (High Performance Computing) and scientific computing community to develop many important applications for the architecture.

The SPE is a fully SIMD architecture, with 128 128-bit registers, and the capability of performing four single-precision floating point operations in a single cycle. Most incarnations of the Cell run at 3.2 GHz, meaning that 8 SPEs executing four multiply-add instructions per second each are capable of reaching a peak performance of 204.8 GFlops.

The SPEs are not capable of accessing main memory, and therefore use a 256-kbyte LS for instruction and data storage. The SPEs use this LS in lieu of a hardware-controlled cache. Transfers between main memory and the LS are controlled only by software. Each SPE has a MFC that manages these DMAs. The

SPEs are capable of transferring data between their LSs as well. The peak bandwidth for transfers between LS and main memory, is 25.6 Gbytes/s, given a 3.2 GHz clock rate. This transfer rate is realistically attainable [11].

2.2.2 Precursors to the Cell

In 1994, TI introduced the TMS320C80, a double-core DSP with very good performance at the time of its release [23]. On the C80, the MP is responsible for task delegation among the four PPs. Although only the MP is capable of floating point operations, the integer unit on each PP is very capable. TI claims a peak of 500 million RISC-equivalent operations per second per PP at 50 MHz. This number is possible because of the ability of the architecture to accomplish one ALU operation, one multiply, and two memory accesses per instruction. The 32-bit ALU could be subdivided into four 8-bit units, with three inputs per unit, meaning the ALU by itself is capable of eight “RISC-equivalent” units, which is how the 500 million number above was determined. In reality, for most algorithms, coding to that level would be impossible. In the memory subsystem, the PPs have access to main memory through a DMA controller, through which software may transfer data to each PP’s 8 kB data RAM for processing.

The C80 and the Cell share many of the same design values. Both are made up of a number of powerful smaller processors that work together at the behest of a larger administrative core. Both are equipped with very high memory bandwidth for their time. The idea of a small, fast memory devoted to each processor element (the Cell’s LS and the C80’s data RAM) is an important idea shared between the two. One important difference is that the Cell is primarily targeted toward floating-point applications, while the C80 targets integer applications. The biggest reason for this

	TMS320C80	MRC6011	Cell Broadband Engine
Year Available	1994	2003	2006
Core Frequency	50 MHz	250 MHz	3.2GHz
Integer/FP	32-bit integer	8-bit integer	Single-precision FP
Local Store Capacity	8 kB	4 kB	256 kB
Peak ops/s	2 Gops/s	48 Gops/s	200 GFlops

Table 2.1: A comparison of previous similar processors (where “Gops/s” is defined as “billions of integer operations per second” with a multiply-accumulate counting as two distinct operations)

is probably that the transistor count required for a dedicated floating point unit for each processor using the technology of the day would have increased the chip’s area beyond practicality. Another difference is that the interconnection network between elements used on the C80 is the crossbar, while the Cell uses a ring. The crossbar is more efficient, but its area grows with N^2 , while a ring’s area grows only with N . The Cell designers chose a ring in order to save area[12], probably with an eye towards future expansion of the number of SPEs. Most of the remaining differences (more memory per chip, faster bus speeds) can be attributed to advances in semiconductor technology (see Table 2.1).

Another processor which shares some characteristics of the Cell’s design is the MRC6011, a chip that uses Freescale’s Reconfigurable Compute Fabric (RCF). The MRC6011 was aimed at being a lower-cost alternative to custom ASIC designs used primarily in cellular telephone base stations [6]. This chip consists of two RC (Reconfigurable Computing) modules, each of which contain three RCF cores. Again, this chip uses a DMA-style transfer mechanism to share data between cores. The MRC6011 uses a bus for DMA transfers, rather than a crossbar or ring, meaning there is likely more contention and less efficiency, although detailed specification of its memory system is not available.

While the C80 and MRC6011 were targeted at the applications primarily served by the high-performance DSP market [22], [6], the Cell was developed primarily for a gaming platform, the PlayStation 3 (PS3), with an eye toward the scientific and HPC markets. The Cell then, has a more general-purpose architecture, geared toward making a large variety of workloads fast, not just those of image and video processing. It has optimized support for both integer and single-precision floating point workloads.

The biggest difference, however, between the Cell and previous generations of technology with similar hardware design philosophies, is the availability of tools to ease the burden of programming. The C80's core-level parallelism is based on the use of a very extensive ALU combined with a dedicated multiplier and two address calculation units. This means that each instruction must be intimately aware of the hardware details of the processor in order for it to be efficient. In contrast, the Cell's SPEs' parallelism is based on well-known SIMD concepts that programmers have been using with great success for years. In addition, the SPEs' vector unit is based on the well-known AltiVec/VMX vector unit from the PowerPC architecture, easing the transition even more. It also appears that programming for both the MRC6011's and C80's elemental cores requires the use of assembly language. Programs for the Cell's SPEs can use C or C++ for the vast majority of programming, with little or no loss in optimization potential. The performance libraries provided by IBM also negate the need for low-level programming in some cases.

Another factor is that since the introduction of these other chips, that multi-core architectures in almost all arenas of computing have become ubiquitous. Then the ideas in distributed and parallel software architecture that are developed for general-purpose computing can be translated to work for the Cell. Although it

is improbable that the Cell will become as widespread as Intel's latest offering, the popularity afforded by the processor's association with Sony's PlayStation 3 means that researchers interested in its potential for other applications can be fairly confident that it will be supported for years to come.

2.2.3 Communication Facilities

The Cell provides a powerful intra-chip communication system for communicating among the PPE, SPEs and main memory called the Element Interconnect Bus (EIB). The EIB uses a ring architecture with four buses to connect between the PPE and the SPEs. There are two buses in each direction. A ring was used instead of a crossbar to save space on the chip [12].

The EIB operates at half the processor clock speed and can fetch 128 bytes per address request, meaning it has a maximum bandwidth of $128 \text{ bytes} \cdot 1.6 \text{ GHz} = 204.8 \text{ GB/s}$. Fetches from main memory are limited by the main memory bus, which operates at a maximum of 25.6 GB/s [11]. This also means that DMA transfers are most efficient when their size is a multiple of 128 bytes. For transfers smaller than 128 bytes, the full size is transferred, but only the relevant data is written, meaning that these smaller sizes are less efficient by their proportion to 128 bytes. Transfers of greater than 128 bytes must also be aligned on a 128-byte boundary.

Each SPE interfaces with the EIB using its Memory Flow Controller (MFC). Each MFC contains a queue of DMA requests. An SPE instruction can enqueue a DMA command on the MFC and allow the SPE to continue execution while the DMA commands in the queue execute independently. The MFC allows only sixteen requests in its queue simultaneously. To allow a series of transfers to or from disparate addresses, a DMA list is used. A DMA list command takes a pointer

to a list of addresses and sizes in an SPE's local store and performs the transfers serially. This allows a long sequence of DMA transfers to be done without having to block on placing a long series of requests onto the MFC.

Normally, the MFC does not strictly enforce ordering among the requests in its queue. This means that request A which was placed on the queue before request B could potentially execute after request B. Special fence and barrier flags can be added which will enforce ordering among the DMA commands in a specific tag group. The tag is a 5 bit number that is associated with each DMA request that organizes the requests into groups. A fenced command will not be performed until all of the other commands in the same tag group have been performed, although others in the same tag group may go before. A barrier command is similar, with the difference being that commands in the same tag group enqueued after may not be performed before the barrier command.

Another communication method between elements are the signal notification registers. These allow the exchange of 32 bits of data among SPEs with a low latency penalty. These can be used to pass around small amounts of data, addresses, or synchronization constructs. The MFC is used for commands that communicate between these registers, so the same sort of fence and barrier flags apply to these as well.

2.2.4 The PlayStation 3

The primary purpose for the development of the Cell was as the microprocessor powering Sony's gaming platform, the PlayStation 3. Sony has succeeded in making the PS3 the most computationally powerful console yet, but has also provided software developers with an inexpensive platform on which to develop HPC and

Manufacturer/System	Sony PlayStation 3	Microsoft XBox 360	Nintendo Wii
Units Sold (millions)	3.7	8.9	9.0

Table 2.2: A comparison of the sales figures of major game consoles in the period from each console’s respective launch to the end of July 2007 (the end of August in Japan). The PlayStation 3 was introduced in late November of 2006.

scientific applications on the Cell. Other offerings from IBM and companies such as Mercury Computer Systems are in the \$10,000+ range and require expensive hardware, while one can obtain a PS3 gaming system at a local department store for a few hundred dollars. As mentioned above, availability of the Linux operating system and IBM’s SDK has put a fiscally viable Cell development platform into the hands of the masses.

The future of inexpensive Cell platforms, and perhaps Cell platforms in general, hinges to some degree on the success of the PlayStation 3. In current sales figures, the PS3 is far behind its competitors, Microsoft’s XBox 360, and Nintendo’s Wii. Table 2.2[20] illustrates the relative sales success of each competing console system as of July of 2007. It’s safe to say that the lion’s share of Cell processors being manufactured today are going into PS3s. To put this number in perspective, compare it with some of Intel’s recent offerings. Between when Intel’s quad-core chips were introduced in November of 2006 and the end of September of 2007, Intel shipped approximately 3 million units [21]. While the Cell has shipped more units in a comparable amount of time, the development costs of the Cell were high enough that it’s unlikely that its budget has been recouped.

The PS3 as a Cell development platform is not without its limitations. To increase yields on the Cell processors manufactured for the PS3, one of the eight SPEs is disabled. Also, due to the fact that the Linux operating system actually runs on

top of a hypervisor built in to the native PS3 Game OS, another SPE is dedicated to that, and is therefore unavailable for computational purposes. Some Linux kernel facilities that have been ported to the Cell, like OProfile, are unfortunately unavailable on the PS3 due to the inability to write to certain configuration registers. The inaccessibility of certain registers also precludes an optimization-minded application developer from altering them for possible performance enhancement. The PS3 does include a Gigabit Ethernet adapter, but those in need of higher bandwidth will find it difficult or impossible to retrofit another interface like Infiniband. An additional limitation is the inclusion of only 256MB of main memory, which could be a problem for applications needing large data sets. Also, graphics developers will be disappointed that the PS3's powerful graphics coprocessor, NVIDIA's RSX, is disabled when running in Linux.

Despite these limitations, the PS3 is an excellent initial development platform for the Cell. Those that wish to preview the Cell's computing prowess would do well to develop on Sony's gaming system, and then perhaps upgrade to a more powerful and suitable offering from IBM or Mercury when the application is mature.

The development for this implementation was done initially on IBM's Cell System Simulator. It was moved to the PlayStation 3 when the system became widely available for purchase.

Chapter 3

Beamformer Implementation

The Cell Broadband Engine provides in many ways an ideal system on which to implement a beamformer of the type described. This application requires large amounts of memory bandwidth, single-precision floating point computation speed, and is very parallelizable. The Cell is strong on all of these points.

3.1 Implementation Overview

The array for which this beamformer was implemented consists of a circular array of 192 sensors. Unfortunately, physical constraints dictate that the sensors cannot be spaced evenly around the array. Most sensors are separated by a distance $2L$, with the constraint that each sixth sensor and the following have a spacing of $3L$. This arrangement can be made to work with the circular beamforming technique described in Section 2.1.3 inserting zero samples in the spaces not occupied by sensors, as in Figure 3.1. This is a sort of upsampling operation. Then for each 6 sensors in the array, there will be 13 samples. This leads to $13 \cdot 192/6 = 416$ total samples in the circular array.

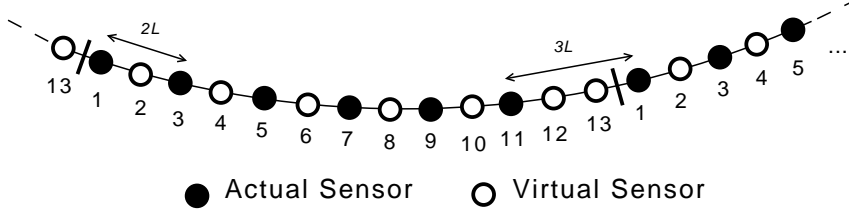


Figure 3.1: Alternating sensors in the actual sonar array.

For reasons that will be discussed in Section 3.5, we will use 8192 samples in the time dimension for the beamform calculation.

The basic steps of the circular beamforming algorithm are:

1. Do a 2D FFT on the input data points.
2. Multiply the result of Step 1 by the DFT of the filter coefficients.
3. Do an inverse 2D FFT on the result of Step 2.

Naively, one might expect to implement this on the Cell as in Figure 3.2. Unfortunately, to harness the full power of the Cell, we must subdivide these steps further for a number of reasons.

The input data array for this beamformer will be 416×8192 , where each point is a complex single-precision floating point number. Each point is therefore 8 bytes, with 4 bytes for each floating point value in the complex pair. Let B be the number of bytes in the input data array. Then

$$B = 416 \cdot 8192 \cdot 8 = 27262976 \tag{3.1}$$

The input array is 26 megabytes, which is far too large to fit on even the aggregate memory of the Cell's SPEs, which have only 256kB of memory each. This is an undesirable situation because it suggests that each SPE would have to pull data

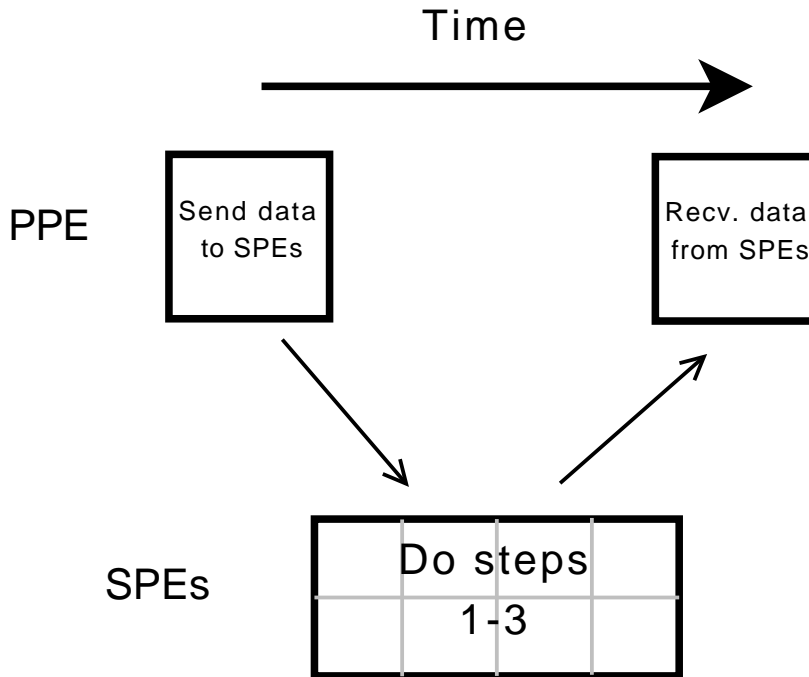


Figure 3.2: Naive proposed implementation of the beamformer.

from main memory during computation, which is a relatively expensive operation. Ideally, each SPE should be constantly executing computation instructions with no stalls waiting for memory operands to return.

So the data must be partitioned somehow in order to use the SPEs for this computation. Fortunately, two-dimensional FFTs are computed by computing a series of one-dimensional DFTs. To compute a two-dimensional DFT, one must first compute a one-dimensional DFT on each row of the array, and then compute a one-dimensional DFT on each column of the result. The row and column DFTs may be computed in any order.

This suggests that individual DFTs can be calculated on each SPE, meaning that no main memory accesses would be required during the computation of a particular DFT. On an SPE, when the computation for a particular DFT is done, it can

request another one until all of the computation for the 2D DFT is finished. The row and column DFTs must be separated because once a particular one-dimensional row DFT is calculated on an SPE, that SPE contains only one of the points necessary for the column DFT. Then each row DFT must be complete before any column DFTs can commence.

Then the algorithm now looks like this:

1. Do 1D row DFTs on the input data.
2. Do 1D column DFTs on the result of the previous step.
3. Multiply the result of Step 2 by the DFT of the filter coefficients.
4. Do inverse 1D column DFTs on the result of the previous step.
5. Do inverse 1D row DFTs on the result of the previous step.

If one looks at this from the viewpoint of implementation on the Cell, we see that Steps 2, 3 and 4 can be implemented together on an SPE without transmitting intermediate results to the PPE. The multiplication coefficients required for the data in each column can be transmitted to the SPE along with the data from Step 1, allowing the multiplication to proceed directly after the DFT computation. The inverse column DFT can then be calculated immediately after without consultation from the PPE. The implementation flow will look like Figure 3.3.

Applications such as this one that deal with computation on large data sets on a traditional microprocessor system are often very sensitive to the number of intermediate writes to main memory. Although the Cell is less sensitive to this consideration due to the use of double-buffering (see section 3.4), the above implementation minimizes the number of intermediate reads and writes to main memory

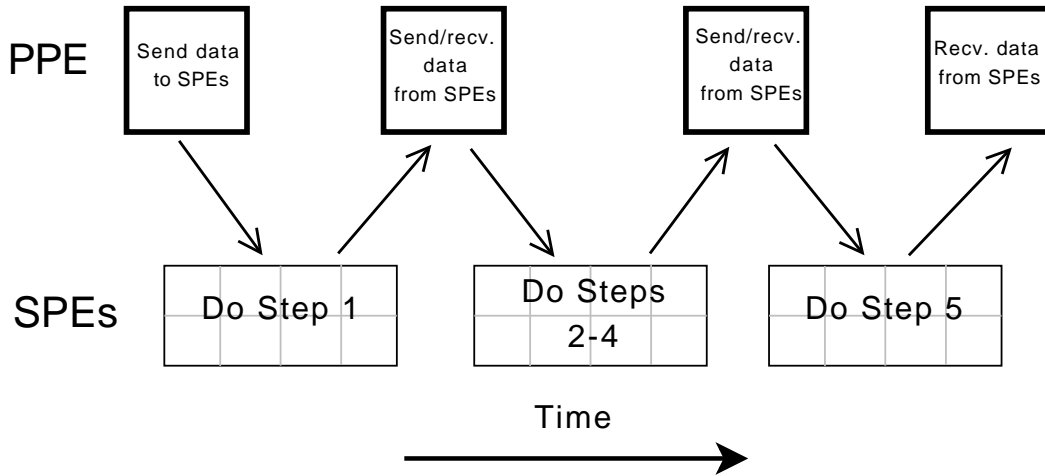


Figure 3.3: Proposed implementation of the beamformer.

at two reads and two writes. These operations are necessary because of the matrix transpositions that are required between the row DFTs and column DFT (i.e. between Steps 1 and 2 and between 3 and 4). The matrix transposition is fundamentally a data movement operation rather than a mathematical operation, like a DFT or a multiply. It cannot therefore be parallelized in the same way. The mechanics of the matrix transposition are described in the next section.

3.2 Distributed Matrix Transposition

3.2.1 Prior Work

There are a few examples in the literature of algorithms for fast matrix transposition. Eklundh's algorithm [4] is an example of one of these that overcomes the problem of transposing a matrix whose full extent cannot be contained within available high-speed storage. Eklundh's algorithm will perform an in-place matrix transpose of a square matrix of $2^N \times 2^N$ size. Ramapriyan [19] presented a generalization of

Eklundh's method for rectangular matrices of arbitrary dimension. The PRIM algorithm [8] is another example of this style of algorithm that the author claims outperforms Eklundh's method. These algorithms were designed for computer systems in which the main data set is stored on disk and operations are done in a higher speed RAM. One might liken this the slower main memory and higher speed LS of the Cell. One could in fact implement each of these on a Cell processor and take advantage of the parallelism inherent in these algorithms. However, one characteristic that all of these methods share is that they assume that the matrix is transposed in-place. However, on the Cell there is enough main memory to store at least two copies of our matrix in question. This means that more efficient means that transpose the matrix out-of-place can be found.

Choi et al. [1] described some matrix transpose algorithms for distributed networks of processors. While interesting, these algorithms also do not have much applicability to the Cell processor. These algorithms assume that there is not a higher-level memory to which the eventual result is to be transmitted. They assume a framework in which the input and output matrix data is interleaved between each processor's memory. In the case of the Cell, we require the input and output of the algorithm to be stored in the overarching main memory of the processor.

3.2.2 Proposed Method

As mentioned above, it's necessary for to perform a matrix transpose of the data at two distinct points in the algorithm. Two different approaches were considered to accomplish this task in an expedient manner.

An obvious approach to this problem would be to perform the matrix transposition on the PPE. This idea has a number of disadvantages. Foremost, within

the framework of the current method, this requires an extra read and an extra write to memory in addition to the reads and writes performed by the DMAs to and from the SPEs. This alone causes the speed of this operation to be prohibitively slow. On an Intel-based workstation of similar clock speed, an out-of-place matrix transposition of this size takes on the order of 10 milliseconds, a non-negligible amount of time in our application.

Since doing the transposition completely on the PPE is slow, the obvious alternative is to somehow do it on the SPEs. As discussed in Section 2.2.3, the Cell's DMA engine is more efficient when the size of each individual DMA is at least 128 contiguous bytes. Because each data point in this implementation is an 8-byte complex float, this makes it impractical to simply DMA every single point individually as shown in Figure 3.4, although this would be possible.

To make this more efficient, the data corresponding to a particular row in the destination should be aggregated together. This will allow the individual DMAs to be larger, and therefore faster and more efficient. One way to do this is to make sure that multiple rows of data are transferred to each SPE to be transformed. Then a matrix transpose can be done on each SPE. For instance, if each SPE has N rows of data, then each individual DMA can transfer $8N$ bytes at a time following each SPE's transpose. This operation is shown in Figure 3.5. In this case, though, one is limited by the number of rows that can be stored, transformed, and transposed on each SPE's limited LS size. A transpose on non-square matrices such as these is almost necessarily an out-of-place operation, meaning twice the amount of buffer space is needed to store the row data is required to perform the operation.

Another way to aggregate the data together would be to share data from other SPEs before transferring it to main memory. In this way, the amount of data

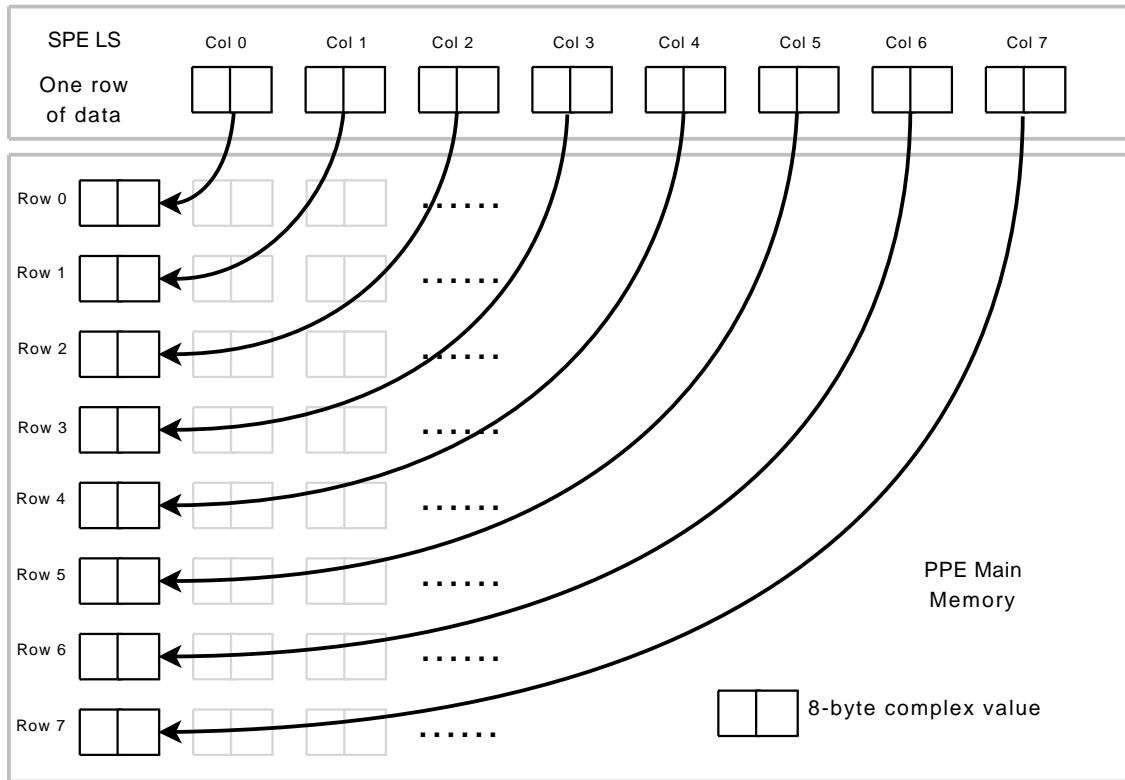


Figure 3.4: Inefficient matrix transpose.

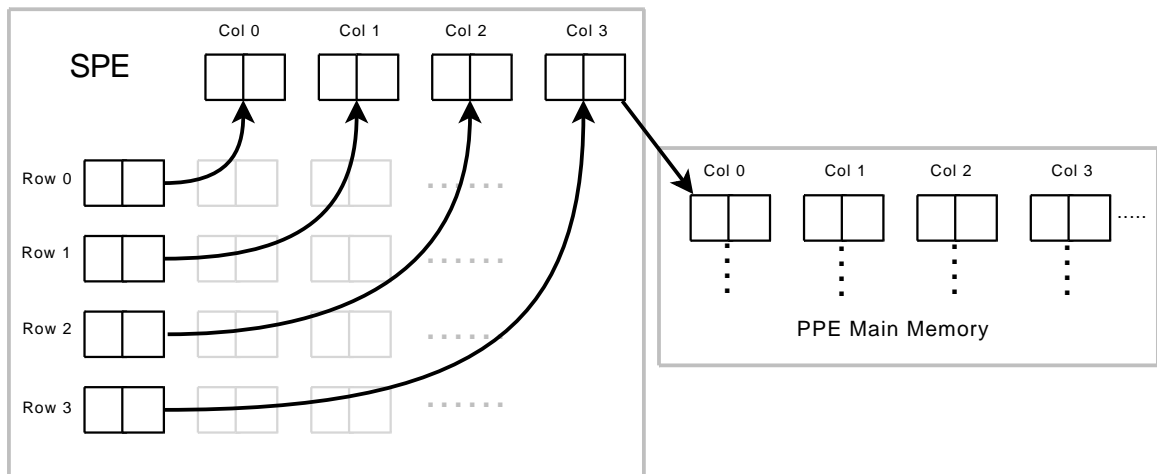


Figure 3.5: Matrix transpose on the SPE (Synergistic Processing Element) using multiple rows.

involved in an individual DMA to main memory can be maximized. In order for this to work, the data has to be distributed to the SPEs in contiguous discrete chunks, e.g. SPE 0 gets the first two rows, SPE 1 gets the second two rows, etc. If the data is distributed in this way, then after the matrix transpose on each SPE described previously, the first row of data on each SPE will correspond to a different part of the first row of data that will eventually go to the PPE. Each subsequent row on each SPE will also share this same characteristic. So then it is possible to exchange data among the SPEs so that each SPE is responsible for some fraction of the rows that will go to the PPE. In this way, the size of the individual DMAs into main memory is maximized, so the DMA data transfer speed is maximized.

This scheme has some disadvantages. First, an additional data read/write is obviously required, although this is not as great of a penalty on the SPEs' low-latency LSs as it would be in main memory. Another disadvantage is that some synchronization is required among the SPEs during each data exchange, which has some overhead as will be discussed in the next section. A third is that an additional buffer is required to allow the exchange transfer to take place while other computation is going on, potentially reducing the number of simultaneous rows of data that each SPE can handle simultaneously. The increased number of buffers also significantly increases the complexity of the double-buffering scheme required to ensure computation can take place during data transfers.

While the initial version of the implementation used the SPE data exchange method for the matrix transposition, it was found that the disadvantages were too prohibitive. A derivative method was devised, motivated by the fact that a completely transposed result is not required during intermediate steps of the algorithm, i.e. between Steps 1 and 2 or 4 and 5.

The transposition can then be done in two interchangeable steps. The first step is similar to the first part of the row transposition method discussed above. Let the original matrix be $M \times N$ elements. Let r be the number of rows of the matrix that will fit in the memory of each SPE. Then each SPE will have an $r \times N$ matrix in its LS at the beginning. After the transpose, the matrix will be $N \times r$. Each SPE will then transfer its matrix to main memory in row-major order, meaning N arrays of length r will be stored serially in memory from each SPE. Once all of the data has been transferred to main memory, the SPEs will transfer each row of r length individually, using a stride length of rN . When M/r rows at this stride are concatenated together in the SPE, they make up a length M row of the transposed matrix of dimension $N \times M$.

It is important to note that the two steps of this algorithm can be interchanged. Instead of the first step being to transpose the data in the SPE, the data can instead be transferred to main memory in the inverse of the way the data was transferred from memory in the second step. In other words, divide the data into packets of length r , and then transfer each to memory with a stride of rN . An rN -sized length of data can then be pulled from memory contiguously by the next set of SPEs and be transposed to yield a row of data of length M .

The interchangeability of these steps is important when considering the dimensions of the data that we require. In this beamformer implementation, a 416×8192 data matrix will be transposed forward and backward. In the first part of the algorithm, the SPEs will have a row of 8192 data points, which is 64 kB. It is impossible to fit another row of data onto the SPE's LS given the current buffering strategy (described in Section 3.4). Then it is important to be able to transfer this particular set of data contiguously and then use the higher value of r allowed by

the smaller rows of 416 points when transferring the data from main memory to the SPEs in the next step. When transposing the data back, the algorithm is performed in the opposite direction for the same reason.

3.3 Synchronization

Synchronization is an important part of most distributed applications, and this one is no exception. In our case, some synchronization is required between each part of the algorithm that transfers data to and from the PPE's main memory, so between parts 1 and 2 and 5 and 6. The reason for this need is that a particular SPE operates independently of other SPEs, and therefore has no way of knowing when the other SPEs have finished their jobs. This lack of knowledge necessitates a way for each SPE to wait for the other SPEs at a given point in the code. This sort of synchronization construct is called a barrier.

Considered were two different methods of barrier synchronization. The first was a model in which one node serves as a leader, with every other node denoted as being a follower. Each follower node signals the leader when it has reached the barrier point, and then waits. When the leader reaches the barrier point, it waits until each one of the followers has signaled it, signals each follower, and then proceeds. When each follower receives this signal from the leader, it proceeds past the barrier as well. This is implemented on the Cell's SPEs using the signal notification registers and mailbox facilities described in Section 2.2.3. The signal notification register on the leader SPE is configured to be in OR mode, meaning that a 32-bit value written by an SPE will be ORed with the current value of the register. By assigning a particular bit in the register to each of the follower SPEs, the leader SPE can be aware of which and how many SPEs have signaled it at any given time.

When the leader SPE reaches the barrier point, it simply polls its signal notification register until the number of bits set equals the number of follower SPEs. It then sends a mailbox message to each of the follower SPEs, and then proceeds. The follower SPEs block on the reception of a mailbox message after they have signaled the leader SPE using the signal notification register facility.

The second, more egalitarian method considered was one in which each node signals every other node that it has reached the barrier point. After signaling, each node waits until it has received a signal from every other node. This method was implemented on the Cell using the signal notification registers in exactly the same way as described above.

The first method has the advantage of requiring fewer messages to be exchanged. Given N SPEs, the leader-follower method will require $N - 1$ messages when each follower signals the leader, and then $N - 1$ more messages when the leader signals each follower, leading to $2(N - 1)$ messages total. On the other hand, the egalitarian method requires N SPEs to send out $N - 1$ messages, meaning $N(N - 1)$ messages are needed.

Figure 3.6 illustrates that as the number of SPEs involved in the barrier operation increases, the time needed increases somewhat linearly, at least for numbers of SPEs this small. The data for this graph was measured by looping each barrier type 10,000 times consecutively and averaging to get the time for a single barrier. From the graph, it is easy to see the expected additional delay caused by the leader follower method. From the graph, one could extrapolate the signal notification latency from the difference in time required between the two types of barriers. In the case of 2 SPEs, the latency is approximately 150 ns, or 480 cycles at 3.2 GHz.

Although this graph only measure up to the number of SPEs present on the

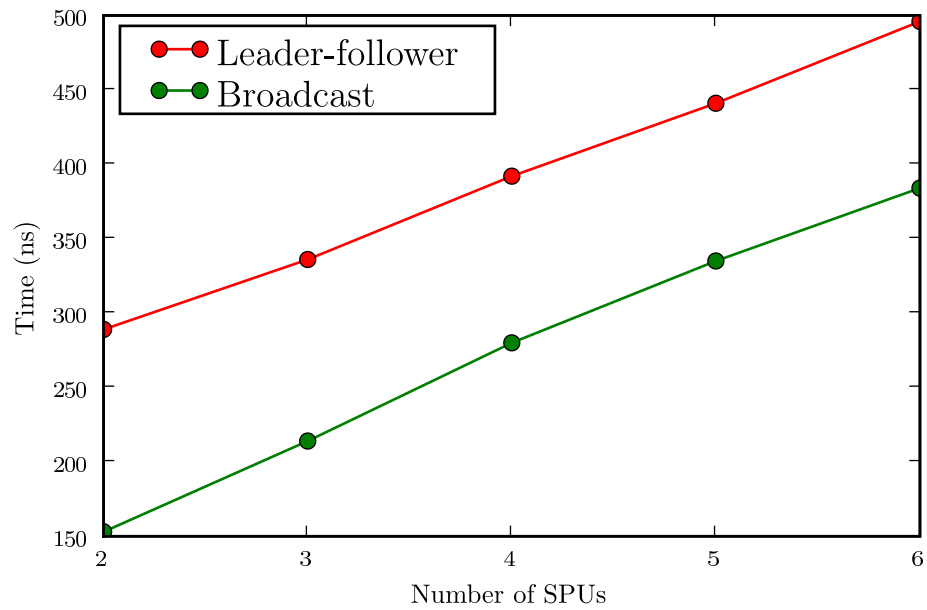


Figure 3.6: Graph of the time required for a barrier for two different synchronization methods.

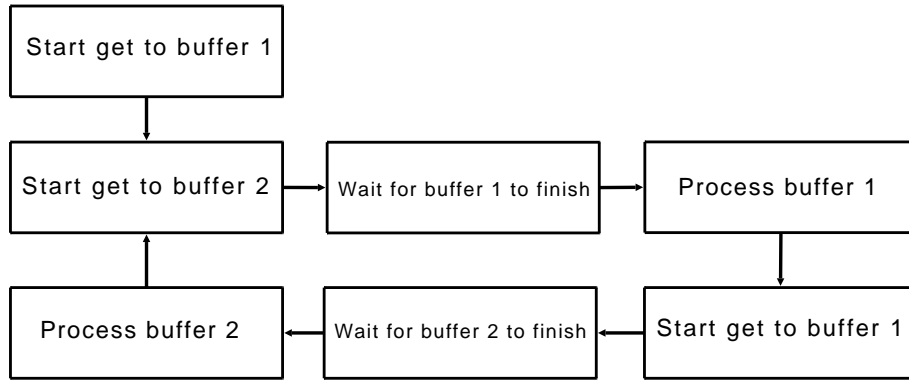


Figure 3.7: Double-buffering flow chart

PS3, this trend would probably continue up to the eight SPEs present on a normal Cell processor. In future iterations of the chip with more SPEs, one might expect to see the broadcast barrier times increasing faster than the leader-follower model, due to the much greater required number of messages. For 6 SPEs, the leader-follower method requires $2 \cdot 5 = 10$ 4-byte messages, while the broadcast method needs $6 \cdot 5 = 30$ messages.

3.4 Double-buffering

Double-buffering is simply processing data in one buffer while simultaneously transmitting data in other buffers. For data-intensive applications like this one, it's very important to utilize this technique in order to maximize memory bandwidth and computation performance. An application will ideally spend zero time blocking on I/O required to process the next batch of data, and the use of double-buffering is one technique needed to minimize this time.

Figure 3.7 illustrates the flow of input and processing of a double-buffering application on the SPE. First, a DMA get is initiated to transfer the appropriate

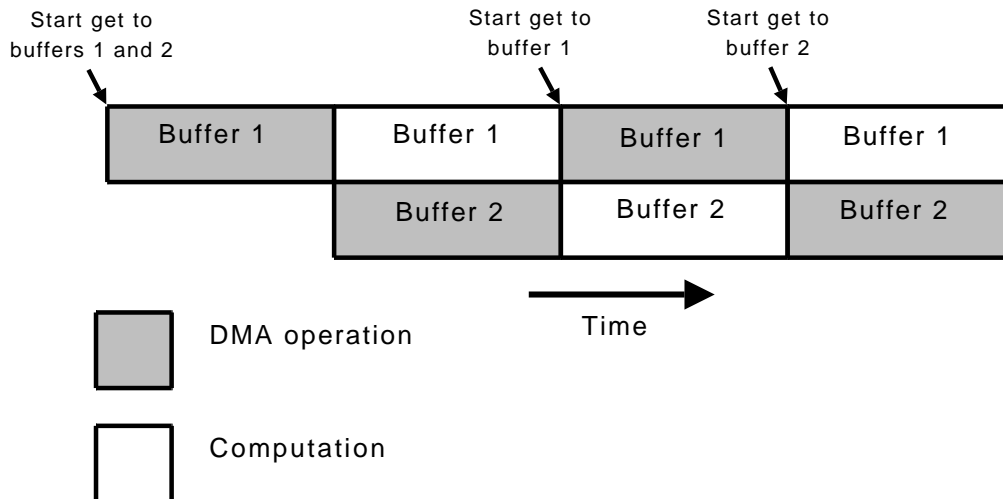


Figure 3.8: Double-buffering illustration

data into both buffers 1 and 2. Execution is then suspended until the DMA get into buffer 1 has finished. This is accomplished using an MFC completion command on the tag group used by the buffer 1 **mfc_get** command (see Section 2.2.3 for details). When completion of the buffer 1 DMA is assured, processing of buffer 1 can commence. When this processing is complete, another **mfc_get** is initiated to buffer 1 to replace the data just processed. Execution is then suspended while waiting for the **mfc_get** to buffer 2 to finish, at which point the data in buffer 2 is executed, and a **mfc_get** initiated to buffer 2 again. The cycle begins again.

In order for this scheme to be optimal, the amount of time required for transferring the data should be less than the time required to process the data. In this case, the waits in Figure 3.7 would be zero, and the entirety of CPU time would be spent on actually processing data, which would maximize performance.

It's necessary to introduce another wrinkle into this problem. In the case of the beamformer implementation, the data needs to be transmitted back to the PPE after processing on the SPEs, and the above method does not account for that. This

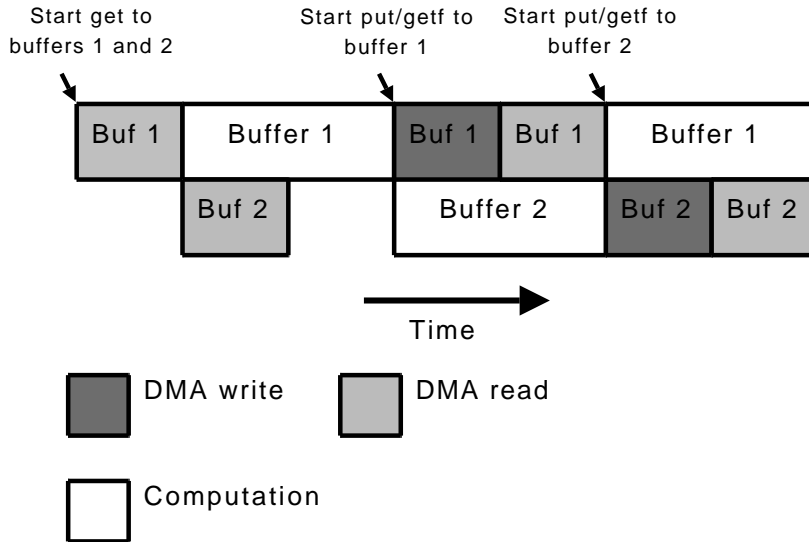


Figure 3.9: Double-buffering illustration with writing

is solved by issuing an `mfc_put` followed by an `mfc_getf` instead of the `mfc_get` only. The `mfc_put` and `mfc_getf` must share the same tag group. This will cause the two DMA operations to be performed serially, with the `mfc_put` completing before the `mfc_getf` can begin. In this case, it's necessary for the sum of the time it takes for one `mfc_getf` and one `mfc_put` to be less than the amount of time it takes for the computation to be done on a single buffer in order to allow optimal performance.

3.5 DFT Filtering Considerations

By adjusting the length of the DFT operations used in the overlap-save computation described in Section 2.1.2, it is possible to minimize the number of arithmetic operations required to calculate the final result. The DFT length is decided by taking into account the two different factors that determine its length: the effect on the overlap-save computational efficiency, and the efficiency of FFT-style subdivision.

These will each be explained in turn.

At each stage of the overlap-save algorithm, the efficiency E is determined by dividing the number of points calculated by the number of points discarded. Then for DFT length N and filter length M ,

$$E = \frac{N - (M - 1)}{N} \quad (3.2)$$

For instance, for an overlap-save operation where the DFT length N is 1024 points, and the filter length M is 32 points, the efficiency will be $1024 - (32 - 1)/1024 = 0.97$. This efficiency describes the fraction of wasted operations at each stage of the overlap-save algorithm. Because of the nature of the FFT it is impossible to simply eliminate the calculation of these particular points. It's easy to see that for a given filter length, the longer the DFT is, the more efficient each stage of the calculation will be.

The efficiency at each stage does not tell the whole story, however. If one is allowed to arbitrarily choose the length of an FFT calculation, a series of shorter FFTs will have fewer operations than a single long one. In mathematical terms, given that an FFT has $5N \log_2(N)$ operations, y x -point FFTs will always require fewer mathematical operations than an xy -point FFT. It's easy to see this by setting

$$f = \frac{5xy \log_2(x)}{5xy \log_2(xy)} = \frac{\log_2(x)}{\log_2(xy)} = \log_x(xy) \quad (3.3)$$

where f is the fraction more computations required by y x -point FFTs over one xy -point FFT. Since we assume $y > 1$, then $f < 1$, meaning the condition required is true. This means that shorter FFTs will yield fewer computations proportionally than longer FFTs.

So while this metric is negatively correlated with FFT length, the stage efficiency metric is positively correlated with FFT length. This means that there is some value of FFT length that will yield the fewest number of computations for a given filter length and signal length.

This value can be determined by deriving an expression for the number of mathematical operations required for a given FFT length, and then minimizing this expression. To do this, let X be the number of points in the signal we wish to filter, N be the FFT length, and M be the filter length. Also let the sequence be segmented into blocks of L points. L will satisfy the equality $N = L + M - 1$ or $L = N - M + 1$. L will also correspond to the number of “good” points that will be gotten from each FFT, i.e. the points that are not discarded. It can be seen that

$$\frac{X}{L} = \frac{X}{N - M + 1} \quad (3.4)$$

FFTs will be required to filter an X -point signal. It can also be seen that this will require

$$\frac{X}{N - M + 1}(10N \log_2(N) + 6N) \quad (3.5)$$

operations, given that each of the two FFTs requires $5N \log_2(N)$ operations, and the complex multiplies require $6N$ operations. Dividing by X to yield operations per point rather than total operations, the quantity is then

$$\frac{10N \log_2(N) + 6N}{N - M + 1} \quad (3.6)$$

We desire

$$\hat{N} = \operatorname{argmin}_N \left\{ \frac{10N \log_2(N) + 6N}{N - M + 1} \right\}. \quad (3.7)$$

That is, we wish to find \hat{N} , the value of N that yields the smallest value for the function for a given value of M (the filter length). Then in order to find \hat{N} , we can differentiate the expression in the curly braces and then set the result to zero. This will yield the critical points of that function. Differentiating the expression, setting it to zero, and simplifying yields

$$-(M - 1)(\ln(2))N + 10 \cdot 2^x = (M - 1)(10 + 6 \ln(2)) \quad (3.8)$$

Here, we have an equation in the form

$$aN + b \log_2(N) = c, \quad (3.9)$$

where $a = \frac{10}{\ln(2)}$, $b = -10(M - 1)$, and $c = (M - 1)(\frac{10}{\ln(2)} + 6)$. This equation can only be solved in terms of the Lambert W function [3]. The Lambert W function is the inverse function to $y = xe^x$, so $x = W(y)e^{W(y)}$. In order to solve Equation 3.9, it needs to be in the form of xe^x . Through more manipulation, Equation 3.9 yields

$$-N \ln(2^{-a/b}) e^{-N \ln(2^{-a/b})} = -2^{c/b} \ln(2^{-a/b}). \quad (3.10)$$

Then

$$\hat{N} = \frac{-W(-2^{c/b} \ln(2^{-a/b}))}{\ln(2^{-a/b})}. \quad (3.11)$$

Filter Size (M)	Optimal FFT Size (N)	N/M
4	12	1.10
8	35	2.90
16	88	4.34
32	210	5.53
64	478	6.55
128	1065	7.47
256	2339	8.32
512	5084	9.14
1024	10965	9.93
2048	23501	10.71
4096	50114	11.48
8192	106409	12.23
16384	225106	12.99
32768	474671	14.49
65536	998063	15.23

Table 3.1: A comparison of the filter size with its corresponding optimum Fast Fourier Transform size and ratio of N/M

The W function has two different real branches, W_0 and W_{-1} , and therefore the equation has two real solutions. For this equation, the W_{-1} branch yields the large positive values of N that are desired. To find the optimum value for $N = 2^x$, x an integer, one must evaluate Equation 3.6 at the adjacent values of $2^x = N$.

One can see that the common rule of thumb of 4:1 for $N:M$ is valid for smaller filter sizes, but as M increases, so does the optimal ratio.

In reality, one would choose the value of N that yields the best performance in the particular system. On systems with cache-based memory schemes performing FFT calculations, one will often see a rapid drop-off in performance when the FFT size increases beyond the size of the largest fast cache. So these calculations will likely only affect the choice of FFT size within the size of the cache on these systems. On the Cell, the limitation is in the size of the local store of each SPE. With a 256

KB LS size and given double-buffering considerations, the largest practical FFT size is 8192 points.

3.6 Optimizations

Although the details of the implementation of the beamformer have already been discussed, there are a few optimization details that have not yet been addressed. These details, while not integral to the implementation, have nevertheless been important to maximizing its performance.

3.6.1 Large TLB Pages

The Translation Lookaside Buffer (TLB) is memory that caches virtual memory translations. The TLB can greatly speed up memory accesses by storing recently used virtual to physical address translations in a higher speed memory. Nominally, virtual addresses will reference an entry in the page table, which is stored in main memory. These Page Table Entries (PTEs) contain a physical address which is then used to reference the area of physical memory that contains the data that was requested using the virtual address. This area of physical memory is called a page. The default size of a page in the PowerPC architecture is 4096 bytes. This suggests that for modern systems, there will be a large number of PTEs. For example, on a PS3, there is 256 MB of main memory. Given 8 bytes per PTE, that means that there will be $8 \cdot 2^{28} / 4096 = 2^{19} = 512$ KB of PTEs, which is the size of the entire L2 cache of the Cell. It's therefore advantageous to cache entries that are frequently used.

A traditional CPU uses a Memory Management Unit (MMU) to do these translations. In fact, this is what the Cell's PPE does. The SPEs use a similar facility

called a Synergistic Memory Management (SMM) unit to do their translations. Each SPE has its own SMM unit, each of which has its own TLB. Each of these TLBs can cache the translations for 256 different PTEs. 256 entries in the TLB means that each SPE is capable of addressing $256 \cdot 4096 = 1048576 = 1 \text{ MB}$ unique bytes before there is a capacity miss in the TLB. This indicates that if the outermost loop of an algorithm must address more than 256 unique pages (automatically true if addressing more than 1 MB), then the TLB will be unable to cache all of the virtual address translations across each execution of the loop. Ideally in high-performance applications, once one has “warmed up” the TLB by accessing all of the memory locations where data is to be accessed, then each memory access hits in the TLB, allowing optimal memory subsystem performance. If each execution of even the outermost loop causes a TLB miss, then this will not be the case, and performance will suffer.

This is a common problem with high-performance applications that deal with large data sets. Luckily, modern architectures have provided a solution. On the Cell, it is possible to configure pages to be as large as 16 MB, rather than 4 KB. This facility is provided by the Linux kernel. At boot time, the user specifies a number of large pages to the kernel, and the kernel then reserves an area of contiguous physical memory with a size equivalent to the number of large pages specified. These pages can then be accessed by a user with appropriate privileges by using `mmap()` on a special file exposed by the kernel.

This is a special-purpose system, with a memory-use profile that is known in advance, so many of the drawbacks to using large pages are negated [14]. The large pages are allocated in advance by the user, so the operating system does not need to do this allocation automatically or intelligently. There is no need for fragmentation

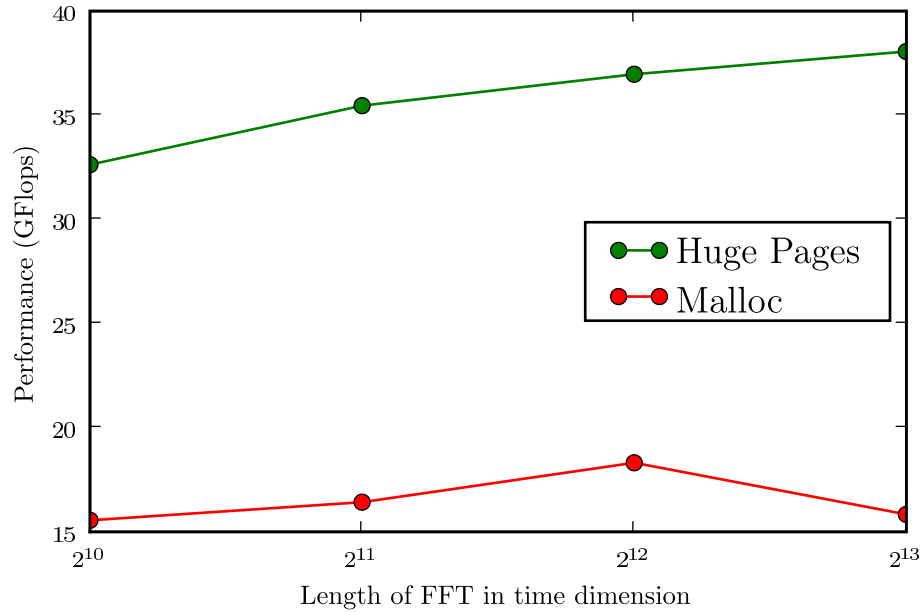


Figure 3.10: Comparison of huge pages vs. malloc

control, because the large pages will be used as specified for the duration of the application's run.

Figure 3.10 compares the performance of the implementation using large pages compared with 4 KB pages.

3.6.2 FFT Software

Because this implementation spends the majority of its time calculating the results of FFTs on the SPEs, it's important to have FFT software that can operate as close to the peak performance of the SPEs as possible. This requirement is complicated by the fact that one of the FFT lengths needed, 416, has a relatively large prime factor (13). The libraries that came with the early versions of IBM's Cell SDK included an optimized FFT implementation. Unfortunately, this library is only able

to calculate FFTs whose lengths are powers of 2. Later versions of the SDK include a library that is able to calculate arbitrary FFT lengths, but again unfortunately only efficiently for FFT lengths with smaller prime factors (3, 5, and 7).

There are some researchers that have made available fast code for the FFT. The most well-known of these projects is probably FFTW [7]. FFTW does fast FFTs by stitching together a set of “codelets” in various ways. Each codelet corresponds to a particular part of the data path for a given transform, and each is pre-generated automatically by code written in a higher-level language. For a given transform size on a given computer, FFTW first generates a “plan,” which specifies which codelets are to be used. This plan is generated by testing the performance of different combinations of codelets and searching for the combination with the highest performance on that machine. Although planning can take a significant amount of time, plans can be stored ahead of time to allow the best code to be used in a given algorithm.

Although the bulk of the work for FFTW was done by Frigo and Johnson at MIT, IBM has donated a port of the software to the Cell. The port, unfortunately, was not suitable for inclusion in this beamformer implementation. This is because the FFTW port assumes that the transform will be called from the PPE on data stored in memory. The beamformer implementation assumes that the transform will be called from the SPEs on data in the local store.

Another group that has taken a more general view of this problem is SPIRAL [17]. The SPIRAL software has not limited itself to generating code for FFTs, and has instead focused on code for general DSP transforms. Also, instead of composing together pre-generated codelets, SPIRAL generates the code for entire transforms. The code is generated using similar methodology, by testing the performance of

prospective transform code with different characteristics and using the fastest, although the search algorithms for SPIRAL are more sophisticated.

In this implementation, code used for FFT calculation is a combination of code from the open-source IBM libraries and the SPIRAL project. The IBM library code is used in the case that the FFT size is a power of two, and the SPIRAL project has generously provided code that is used in the other case.

3.6.3 Efficient Memory Bank Utilization

In order to access memory efficiently, it's necessary to take into account the memory layout. The Cell uses a memory banking system that interleaves memory at strides of 128-bytes across sixteen different memory banks [9]. A memory architecture like this favors large contiguous accesses. If accesses of less than 128 bytes at strides of $128 \cdot 16 = 2048$ bytes, or multiples thereof are made, then access will be inefficient. This is because these sorts of accesses will only be presented to one physical memory bank, whose bandwidth will be less than if accesses are distributed across all of the banks.

This sort of discontinuous access pattern has the potential of occurring at one particular point in the Cell beamformer implementation, during the matrix transpose. When the DMAs involved in the matrix transpose occur, they are at a large memory stride that depends on the number of columns that are processed in each step of the matrix transpose. The maximum number of these in this particular implementation is 8. Given each sample is two 4-byte single-precision floating point numbers, this means that every individual DMA transfer is $8 \cdot 2 \cdot 4 = 64$ bytes. Since the typical row size is 8192 elements, this also means that these transfers are done at a stride of $8 \cdot 8192 = 65536$ bytes. Since 65536 is a multiple of 2048 and these

transfers are relatively small, they easily fit the profile described above.

In order to solve this problem and allow efficient memory access, a change to the data layout was required. When Part 1 of the algorithm does the row FFTs on the input data, instead of sending the data back to main memory as before, a 128-byte buffer between each row is inserted. This means that the first element of each row will be $65536 + 128$ bytes apart, instead of 65536. This will cause the DMA transfers in the next part to access each memory bank in turn, instead of only accessing a single one repetitively. The data is sent back to main memory in the same fashion, and when the last row inverse FFTs are done, the data is sent back to main memory without the buffer, causing the output format to remain the same as it was before.

This change is easy to implement because it does not require any change in the input or output format of the data. It only requires a minor change in the address calculation of DMA transfers. The difference in performance is fairly dramatic, and is illustrated in Figure 3.11.

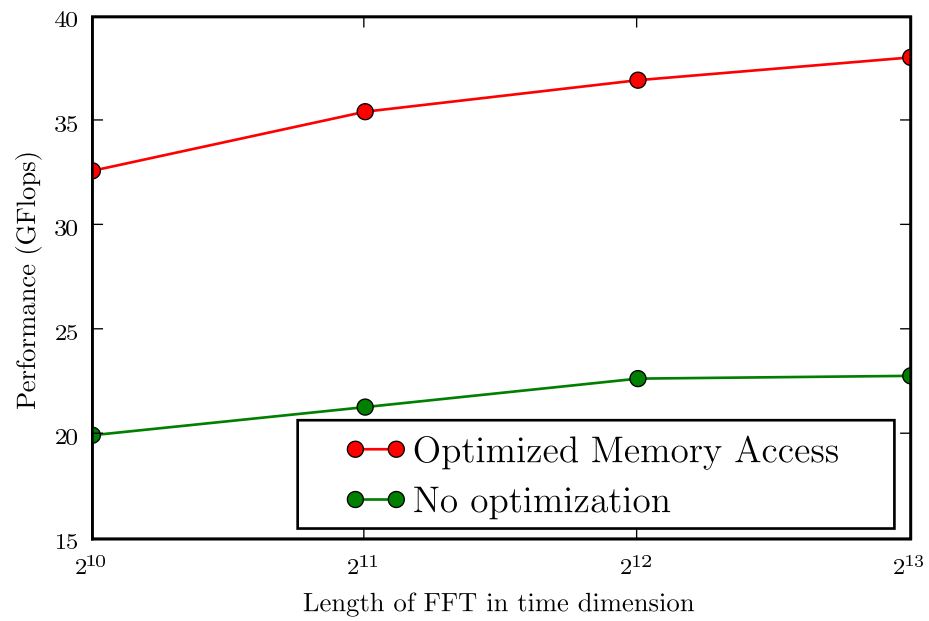


Figure 3.11: Difference in performance caused by efficient memory bank utilization

Chapter 4

Implementation Results

In this chapter, the overall performance of the Cell processor beamforming implementation will be discussed. Along with this, the Cell's results will be compared with the performance of an implementation on an Intel workstation. The comparative engineering effort required for each implementation will also be discussed.

4.1 Cell Processor Results

The implementation performs a beamforming operation on 192 sensing elements arranged in a circular array. It operates on a maximum of 8192 samples at a time, and uses the circular beamforming operation discussed in Section 2.1.3. The eventual output is a set of beamformed points in time in 416 radial directions.

The performance results for the Cell beamformer were measured on a PlayStation 3 running Linux and IBM Cell SDK 3.0. The Cell processor under Linux on the PS3 is limited to 6 SPEs for reasons described in Section 2.2.4.

Number of SPEs	Performance (GFlops)	Speedup
1	6.583	-
2	13.147	1.997
3	19.268	2.927
4	26.191	3.979
5	32.395	4.921
6	38.012	5.774

Table 4.1: Illustration of performance and speedup by varying number of Synergistic Processing Elements

4.1.1 Speedup

Scalability is very important when it comes to parallel computing applications. Scalability is the ability of a system to increase performance with the addition of more of some repeatable element. An important factor in scalability is the amount of speedup gained by increasing the number of computing elements devoted to some task. If the speedup increases linearly, meaning the addition of a computing element to a task increases the performance on that task by the same proportion as that addition, then the speedup is said to be linear. An application is said to have good scalability if it has linear or near linear speedup.

Speedup is generally limited by overhead. Overhead is the amount of time that a parallel computing system has to spend on tasks related to the parallelism that do not perform useful work, where useful work is defined as work performed essential to the result. Potential sources of overhead include shuffling data between computing elements and synchronization constructs that might be required, including barriers.

The Cell beamformer implementation has shown that it has very good scalability across varying numbers of SPEs. The speedup for 6 SPEs is 5.774, as shown in Table 4.1.

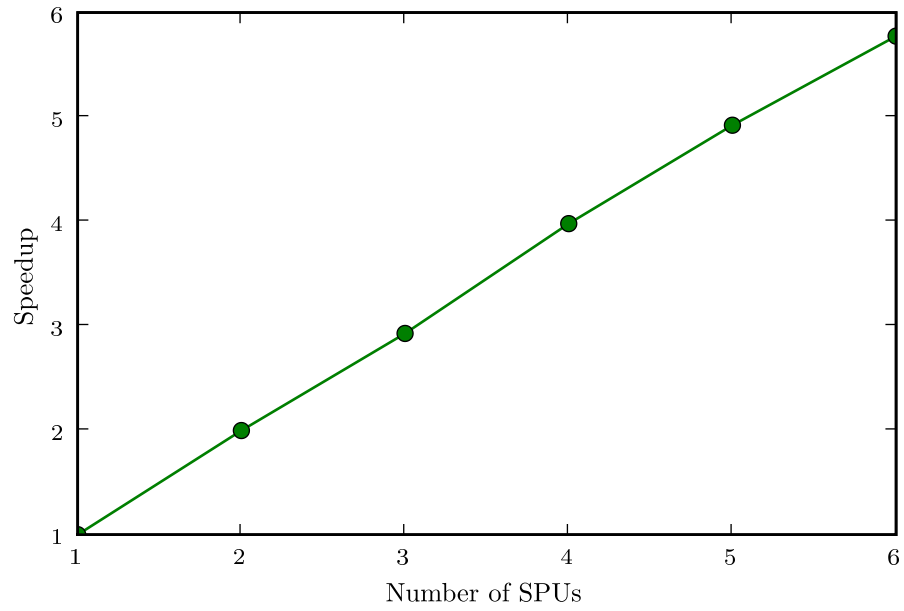


Figure 4.1: Speedup obtained for varying the number of Synergistic Processing Elements in the beamformer

4.1.2 Calculation versus Data Transfer Time

An important aspect of this implementation is its ability to maintain a stream of data going into the calculation pipeline. Ideally, at a given moment, each SPE is doing calculations on data rather than blocking on I/O. It's important then, to verify that the amount of time required for data transfer for each stage of the implementation requires less time than the computation. Figure 4.2 - 4.5 show the amount of time required for the calculation and data transfer segments of each part for varying numbers of SPEs. The calculation segment of the implementation is the same as the regular implementation, except that the DMA commands are disabled. The data transfer segment disables all of the calculations required for the beamformer calculation, although there is still some computation involved in calculating the addresses to DMA to and from.

The double-buffering allows the implementation to perform calculations during the time required for DMA transfers. This is shown in the figures by comparing the time required for calculation and the time required for both calculation and DMA transfer. For each part and the aggregate of all parts, they are very close to the same.

Interesting to note is that the time to DMA data does not vary much with the number of SPEs. This is evidence of both the efficiency of the EIB and the relative bottleneck of the path to main memory. The EIB does not require all eight SPEs to be transferring data simultaneously to saturate the path to main memory, and only seems to require one or two. Fortunately for this implementation, there is plenty of memory bandwidth to keep computation pipeline supplied for up to 6 SPEs. It also appears that even with 8 SPEs, the memory bandwidth would be able to keep the pipeline supplied.

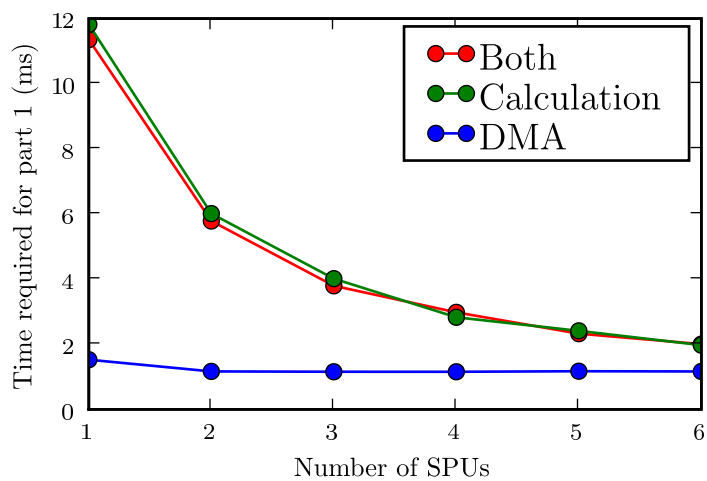


Figure 4.2: Time required for calculation and data transfer sections for a varying number of Synergistic Processing Elements for Stage 1 of the implementation.

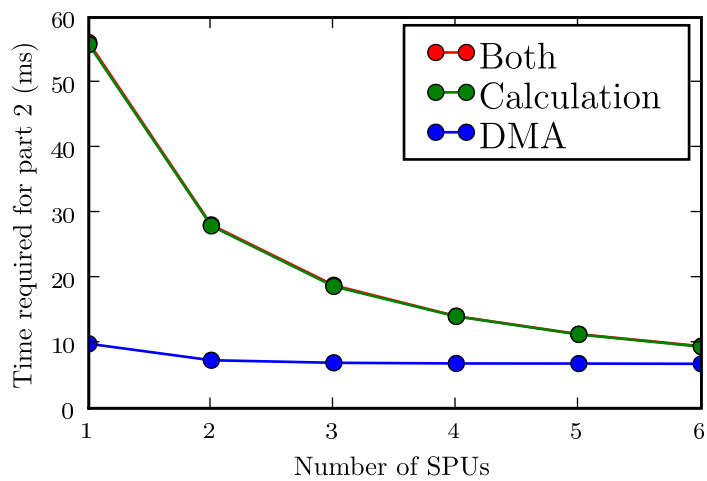


Figure 4.3: Time required for calculation and data transfer sections for a varying number of Synergistic Processing Elements for Stage 2 of the implementation.

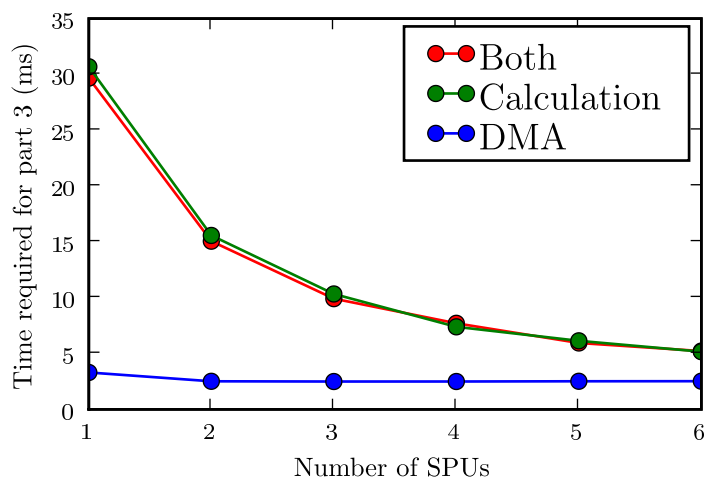


Figure 4.4: Time required for calculation and data transfer sections for a varying number of Synergistic Processing Elements for Stage 3 of the implementation.

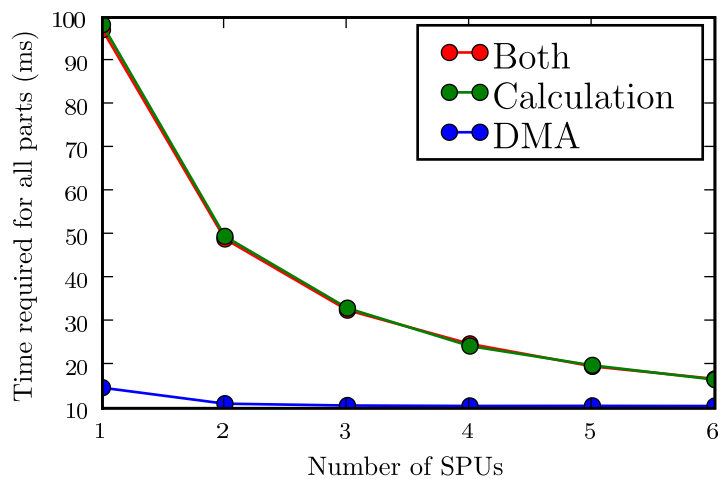


Figure 4.5: Time required for calculation and data transfer sections for a varying number of SPEs for all stages aggregated.

Number of Cores	Performance (GFlops)	Speedup
1	1.55	-
2	2.84	1.83
3	3.37	2.17
4	4.06	2.61

Table 4.2: Illustration of performance and speedup by varying number of CPU cores utilized in the Intel-based beamformer implementation

4.2 Comparison with an Intel-based Implementation

In order to provide a comparison for the given beamformer workload, an implementation suited to an x86-64-based architecture is contrasted. The system is a dual-processor configuration of the dual-core Intel Xeon 5148 processor, for a total of four cores, each clocked at 2.33 GHz. Each of the two processors had 4096 kB of L2 cache. The Front Side Bus (FSB) speed is 1.33 GTransfers/s, where each transfer is 8 bytes [10]. This implementation uses FFTW 3.1.3alpha3 for the FFTs and a set of custom-written functions for the matrix transposes and complex multiplies. Each operation was optimized using SSE vector instructions where appropriate. Parallelism was implemented at every step of the calculation using FFTW’s built-in pthreads implementation or OpenMP. The performance was measured by repeating a beamform calculation 100 times and then finding the average execution time. The number of operations required for a beamform was then divided by this number to determine the average number of GFlops obtained for each number of cores. The results are summarized in Table 4.2 and Figure 4.6.

As demonstrated by these figures, the speedup shown by this Intel processor configuration was much inferior compared to the Cell. The reason for the lesser speedup can be attributed to the differences in the way that the memory is utilized between the two architectures. In the Intel implementation, calculations are done ei-

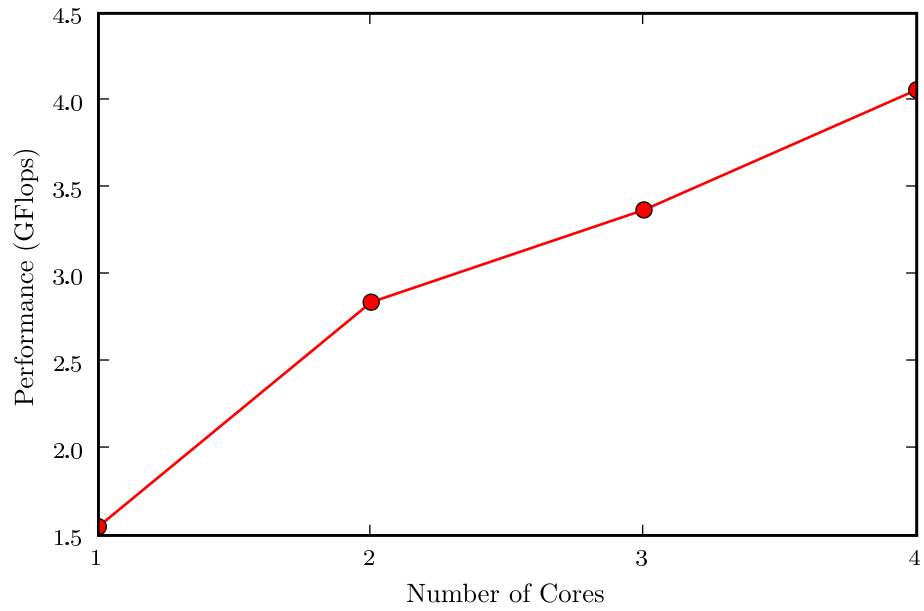


Figure 4.6: Speedup obtained for varying of CPU cores utilized in the Intel-based beamformer implementation

ther out of main memory or out of cache. In the case of operations that exhibit good spatial and temporal locality (like the FFT), good performance can be obtained. In the case of operations that do not exhibit this locality (matrix transposition and streaming complex multiplies being examples), the processor is limited by the available memory bandwidth. For this particular system, the memory bandwidth is 1.33 GT/s, with 8-byte transfers, for a total of 10.66 GB/s of memory bandwidth shared between all four cores. This means that the streaming complex multiply of 4-byte single-precision floating point numbers are limited to 2.66 GFlops in the case of bandwidth-limited operation, independent of the number of cores used.

Another factor in the poor speedup is the matrix transpose. In the Cell implementation, the matrix transpose is a negligible component of the calculation time, and therefore contributes negligibly to the loss of speedup. In the case of the Intel implementation, each matrix transpose contributes significantly to the calculation time without contributing any useful work to the calculation, while at the same time, the transpose is not efficiently parallelized because it is a memory intensive operation.

Another difference in the pure level of performance is the fact that the Cell is able to come much closer to its peak level of performance on real workloads. The theoretical peak performance of the 4 core Intel machine is $8 \text{ operations/cycle} \cdot 2.33 \text{ cycles/s} \cdot 4 \text{ cores} = 74.56 \text{ GFlops}$. The Intel beamformer implementation is only able to reach approximately 4 GFlops, 5% of this value. The peak performance of a Cell with 6 SPEs is $25.6 \text{ GFlops} \cdot 6 \text{ SPEs} = 153.6 \text{ GFlops}$. The Cell implementation reaches approximately 38 GFlops, or nearly 25% of the peak performance.

Chapter 5

Conclusion

This implementation has made it clear that the Cell is a very powerful platform for DSP-intensive workloads. It is capable of drastically outperforming implementations based on general-purpose architectures like those of Intel. However, the low-level programming that is necessary to fully take advantage of the Cell is very engineering-intensive, a model that may not be suitable for all applications. While an efficient implementation on an Intel system will require low-level programming at the assembly level in many cases, it's not necessary to explicitly specify memory movement among computing elements, like it is on the Cell.

The Cell architecture has proved very suitable for the sonar beamforming application to which it was applied in this report. However, the programming-intensive nature of parallel programming makes it unclear if the power of the Cell can be harnessed easily enough to make it a viable processor for a wide variety of applications.

Bibliography

- [1] Jaeyoung Choi, Jack J. Dongarra, and David W. Walker. Parallel matrix transpose algorithms on distributed memory concurrent computers. *Parallel Computing*, 21(9):1387–1405, 1995.
- [2] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.
- [3] Robert M. Corless, G.H. Gonnet, D.E.G Hare, D.J. Jeffrey, and D.E. Knuth. On the Lambert W function. *Advances in Computational Mathematics*, 5:329–359, 1996.
- [4] J.O. Eklundh. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, C-21(7):801–803, 1972.
- [5] D. R. Farrier, T.S. Durrani, and J. M. Nightingale. Fast beamforming techniques for circular arrays. *Journal of the Acoustical Society of America*, 58(4):920–922, 1975.
- [6] Freescale Semiconductor, Chandler, Arizona. *MRC6011 Reconfigurable Compute Fabric Product Brief Advance Information*, December 2004.

- [7] M. Frigo and S.G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, Feb. 2005.
- [8] Geoffrey C. Goldbogen. Prim: A fast matrix transpose method. *IEEE Transactions on Software Engineering*, SE-7(2):255–257, 1981.
- [9] IBM Corporation, Sony Computer Entertainment, Toshiba Corporation, Hopewell Junction, NY. *Cell Broadband Engine Programming Handbook*, April 2007.
- [10] Intel Corporation, Santa Clara, CA. *Dual-Core Intel Xeon Processor 5100 Series Datasheet*, August 2007.
- [11] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. May-June 2006.
- [12] Mike Krolak. Meet the experts: David Krolak on the Cell Broadband Engine EIB bus. <http://www-128.ibm.com/developerworks/power/library/pa-expert9/>, 2005. [Online; accessed 22-October-2007].
- [13] Ronald A. Mucci. A comparison of efficient beamforming algorithms. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 32(3):548–558, 1984.
- [14] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan L. Cox. Practical, transparent operating system support for superpages. In *USENIX: Operating Systems Design and Implementation*, 2002.
- [15] Roger G. Pridham and Ronald A. Mucci. A novel approach to digital beamforming. *Journal of the Acoustical Society of America*, 63(2):425–434, 1978.
- [16] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms and Applications*. Pearson Education, 1996.

- [17] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, Feb. 2005.
- [18] H.P. Raabe. Fast beamforming with circular receiving arrays. *IBM Journal of Research and Development*, July 1976.
- [19] H.K. Ramapriyan. A generalization of Eklundh’s algorithm for transposing large matrices. *IEEE Transactions on Computers*, C-24(12):1221–1226, 1975.
- [20] Mariko Sanchanta. Nintendo’s Wii takes console lead. <http://search.ft.com/ftArticle?id=070912011446&ct=0>, 2007. [Online; accessed 24-October-2007].
- [21] Anton Shilov. Demand for quad-core processors accelerating – Intel. <http://www.xbitlabs.com/news/cpu/display/20071017085352.html>, 2007. [Online; accessed 24-October-2007].
- [22] Texas Instruments, Inc., Houston, Texas. *TMS320C8x System-Level Synopsis*, September 1995.
- [23] Texas Instruments, Inc., Houston, Texas. *SMJ320C80 Digital Signal Processor (Rev. B) Datasheet*, June 2002.
- [24] Walter Welkowitz. Directional circular arrays of point sources. *Journal of the Acoustical Society of America*, 28(3):362–366, 1956.
- [25] Jack R. Williams. Fast beam-forming algorithm. *Journal of the Acoustic Society of America*, 44(5):1454–1455, 1978.

Vita

This is the Vita.

Permanent Address: 4204 Speedway #203
Austin, TX 78751

This report was typeset with $\text{\LaTeX} 2_{\epsilon}$ ¹ by the author.

¹ $\text{\LaTeX} 2_{\epsilon}$ is an extension of \LaTeX . \LaTeX is a collection of macros for \TeX . \TeX is a trademark of the American Mathematical Society. The macros used in formatting this report were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.