12-2009

# Accelerated CTIS Using the Cell Processor

Thaddeus James Thompson
*University of Tennessee - Knoxville*

To the Graduate Council:

I am submitting herewith a thesis written by Thaddeus James Thompson entitled "Accelerated CTIS Using the Cell Processor." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael Vose, Major Professor

We have read this thesis and recommend its acceptance:

James Plank, Michael Berry

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Thaddeus James Thompson entitled "Accelerated CTIS Using the Cell Processor." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science.

Michael Vose, Major Professor

We have read this thesis
and recommend its acceptance:

James Plank

Michael Berry

Accepted for the Council:

Carolyn R. Hodges
Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

# Accelerated CTIS Using the Cell Processor

A Thesis Presented for the
Master of Science Degree
The University of Tennessee, Knoxville

Thaddeus James Thompson
December, 2009

## Dedication

To my wife Robyn, and my mom.

# Abstract

The Computed Tomography Imaging Spectrometer (CTIS) is a device capable of simultaneously acquiring imagery from multiple bands of the electromagnetic spectrum. Due to the method of data collection from this system, a processing intensive reconstruction phase is required to resolve the image output. This paper evaluates a parallelized implementation of the Vose-Horton CTIS reconstruction algorithm using the Cell processor. In addition to demonstrating the feasibility of a mixed precision implementation, it is shown that use of the parallel processing capabilities of the Cell may provide a significant reduction in reconstruction time.

# Contents

# List of Tables

# List of Figures

# Abbreviations

**ART** Algebraic Reconstruction Technique

**CBEA** Cell Broadband Engine Architecture

**CTIS** Computed Tomography Imaging Spectrometer

**DMA** Direct Memory Access

**EIB** Element Interconnect Bus

**FFT** Fast Fourier Transform

**FLOPS** Floating Point Operations per Second

**GPU** Graphics Processing Unit

**MART** Multiplicative Algebraic Reconstruction Technique

**MERT** Mixed Expectation Reconstruction Technique

**MFC** Memory Flow Controller

**PPE** Power Processing Element

**PS3** PlayStation® 3

**SIMD** Single Instruction Multiple Data

**SPE** Synergistic Processing Element

**SPU** Synergistic Processing Unit

**TLB** Translation Lookaside Buffer

# Chapter 1

# Introduction

The use of images taken at various light wavelengths, termed multispectral or hyperspectral imagery, has found numerous applications within the last few years. Because each wavelength of light may react differently with various materials, analyzing images at multiple spectral wavelengths can allow for better feature extraction. This capability has application in everything from identification of astronomical phenomena [1] to target classification in smart missiles [2] to biological analysis at the cellular level [3]. While several methods exist to collect multispectral imagery, Computed Tomography Imaging Spectrometry (CTIS) has recently gained traction as a viable technique. The advantage of CTIS is that unlike most earlier methods, multiple spectral images may be acquired simultaneously. The trade off for this capability is that collected data can require significant computational effort to reconstruct into the output images. Figure 1.1 illustrates an object imaged with three spectral layers.

Because of the advantages CTIS offers for collecting multispectral imagery, a focus of recent research has been on optimizing the speed of reconstruction algorithms and their implementations. In 2006 Vose and Horton proposed a reconstruction technique exploiting the spatial shift invariance of the CTIS system and providing a substantial decrease in reconstruction time [4]. Hagen et. al. explore a related approach in their 2007 paper [5], while Sethaphong s parallelized implementations of CTIS reconstruction algorithms in his 2008 master's thesis. The publication date of the Vose-Horton solution was too late for Sethaphong to implement it for his thesis, but he comments that it needed to be rewritten for a multiprocessor system in order evaluate its suitability for achieving near real time performance [6]. That implementation will be the focus of this work.

Currently there exist a variety of options for implementing high speed data intensive computations. Symmetric multi-core processors are becoming common on the desktop and clusters of computers are easily assembled on even a modest budget. Additionally, special purpose hardware such as graphics processing units, digital signal processors, and field programmable gate arrays are seeing wider use as dedicated application accelerators. One promising architecture designed for application acceleration is the Cell Broadband Engine Architecture. The Cell processor provides a heterogeneous computing platform with a single general purpose processor driving multiple high speed vector processors across a fast interconnection bus. This architecture makes the Cell well suited to demanding scientific computations [7]. These features led to the selection of the Cell processor as the target platform for our implementation of the Vose Horton solver.

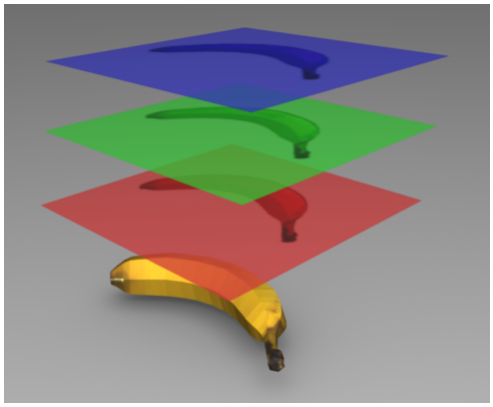


Figure 1.1: A data cube of three spectral wavelengths

# Chapter 2

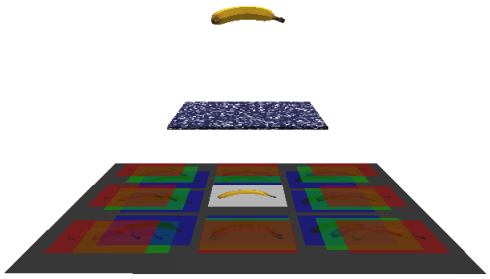
# Computed Tomography Imaging Spectrometry

An imaging spectrometer acquires data from multiple bands of the electromagnetic spectrum which can be presented as a three dimensional data cube. Each voxel within the cube can be interpreted as an intensity value at (x,y, $\lambda$) where x and y are the two dimensional coordinates of the image and $\lambda$ is the wavelength. Thus, each "slice" of the cube represents a 2-D spectral image of the target object. In traditional imaging spectrometers this was accomplished through a variety of techniques such as taking multiple images of an object through different filters with distinct wavelength transmission properties. By employing a scanning mechanism, the three dimensional data cube can be created with one layer imaged per sweep. This works well for imaging static objects, however it is unusable for dynamic scenes where the target object may move between scans.

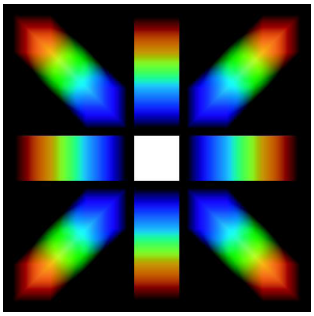
The computed tomography imaging spectrometer (CTIS) was presented as an alternative to scanning spectrometers [8], [9]. The CTIS employs a diffraction grating which disperses the incident light and creates multiple projections of the target object on a focal plane. Because the multiple projections include both spectral and spatial information, the three dimensional spectral cube can be reconstructed using computed tomography techniques. The advantage of this method is that because the entire spectral data cube is projected, only one exposure is needed for data acquisition. This capability, also referred to as flash spectrometry, reduces the acquisition time compared to scanning spectrometers and allows the CTIS to image dynamic scenes. More recently, improvements in the diffraction grating, increased computing power, and larger focal plane array elements have significantly increased the capabilities of CTIS systems [10], [11]. Figure 2.1(a) shows a conceptualized object projected through a grating disperser onto a focal plane. Figure 2.1(b) demonstrates the distribution of white light on the focal plane. The center image contains the full spectrum while the surrounding images show the dispersed spectrum.

## 2.1 CTIS Image Reconstruction

While CTISs gain an edge over scanning spectrometers in data acquisition time, a price is paid in the computation time needed for data reconstruction. The CTIS process is modeled as a linear system which admits the use of standard linear algebra techniques in its analysis. In the literature it is common to let the 2D image from the focal plane be represented by the



(a) Object, Dispersion Grating, and FPA



(b) Spectral Dispersion
Source: Hagen et. al. [5]

Figure 2.1: Grating dispersing an image onto a focal plane.

vector **g** and the 3D spectral cube be represented by the vector **f**. The matrix $H$ represents the transfer function of the CTIS system and relates the two data vectors to each other as

$$g = Hf$$

Each column of $H$ corresponds to the response of the CTIS system to single wavelength band at a given position, that is, a single voxel in the data cube. $H$ is usually acquired during calibration of the system [10]. Since the output of the system is $g$ and the data of interest is $f$, the key question to be answered is: how do we efficiently recover $f$?

When $H$ is relatively small, the inverse or Moore-Penrose inverse of $H$ can often be used to solve for $g$ directly. However, for large format CTIS systems the transfer matrix may have a row and column count in the tens or hundreds of thousands making the computation of the inverse infeasible with modern methods.

In 1970, Gordon et. al. presented methods for the reconstruction of 3D objects from electron micrographs and X-ray images they termed the Algebraic Reconstruction Techniques (ART) [12]. In their original work they present the additive and multiplicative reconstruction techniques and note that the multiplicative method gives a higher entropy solution than that of the additive algorithm. Since then, the Multiplicative Algebraic Reconstruction Technique (MART) has become a common tool in the field of computed tomography for 3D image reconstruction [8], [3], [13]. The essential formulation of MART is

$$f^{k+1} = f^k \frac{H^T g}{H^T H f^k}$$

where $f^k$ is the $k^{th}$ estimate of the object cube vector.

In order to deal with both the photon noise and system noise present in implemented CTIS systems, Garcia and Dereniak published the Mixed Expectation Reconstruction Technique (MERT) [14]. In their extended version of the reconstruction approach, they consider the problem

$$g = Hf + n1 + n2$$

where $n1$ is photon noise and $n2$ is system noise. Their iterative reconstruction technique is fundamentally similar to MART while incrementally compensating for system noise in the recovered image.

## 2.2 The Vose-Horton Algorithm

In 2006 Vose and Horton proposed a heuristic image reconstruction technique which offers a substantial benefit in reduced computation time for systems in which the transfer matrix has a special form [4]. The paper includes an explicit solution to the noise removal formulation given in [14] along with an iterative algorithm for image recovery. The primary focus of this paper is on the implementation and acceleration of the heuristic recovery algorithm (hereafter referred to as the VH solver). The noise removal technique is a fairly straightforward manipulation of the output $g$, ancillary to the reconstruction algorithm, and not discussed further here. In order to effectively present the accelerated implementation of the reconstruction algorithm, a review of the mathematical model is useful. Accordingly, these next two sections draw heavily from [4].

### 2.2.1 Strategy

The VH solver obtains an approximation to the problem

$$\vec{x} = H\vec{f}$$

where $H$ is the $n \times m$ transfer matrix of the CTIS process, $x$ is the length $n$ output vector of the system after noise removal, and $f$ is the length $m$ vector representing the object image. As noted before, as the size of the system and matrix increases, recovering the image quickly becomes computationally difficult.

The strategy used by the algorithm to deal with this limitation is based on two assumptions

1. $H$ can be partitioned into $w$ blocks

   $$H = (H_0|\ldots|H_{w-1})$$

   where each $H_k$ is partitioned into the same number of rectangular circulant blocks

   $$H_k = (T_{k,0}|\ldots|T_{k,\alpha-1})$$

2. Each $H_k$ is a shift-invariant system with respect to $T_{k,0}$. Specifically, from the Vose-Horton paper:

   Assume each $T_{k,l}$ $n \times a$ (thus $H$ has $m = a\alpha w$ columns) and

   $$T_{k,l} = R_g^l T_{k,0}$$

   where $g \geq a$, $n \geq \alpha g$ and $R_g$ is the $n \times n$ circulant matrix

   $$(R_g)_{i,j} = [g = i - j \bmod n]$$

   Here the notation [*expression*] denotes 1 if *expression* is true, and 0 otherwise.

Through the use of a masking matrix, $H_k$ may be embedded into a circulant matrix $C_k$. The VH paper defines the $g \times a$ matrix Q

$$Q = \left( \frac{I_a}{0} \right)$$

where $I_k$ is the $k \times k$ identity matrix. The $n \times a\alpha$ masking matrix E is then defined as

$$E = \left( \frac{I_a \otimes Q}{0} \right) \tag{2.1}$$

where $\otimes$ is the Kronecker product. By *assumption 2* above, the operation $H_k\vec{w}$ is equivalent to $C_k E\vec{w}$ for any $a\alpha \times 1$ vector $\vec{w}$ [4]. Since the effect of the multiplication $E\vec{w}$ is only to expand $\vec{w}$ to an $n \times 1$ vector, it is trivially implemented as a mapping function. For example, the implementation presented in this paper uses a bit vector to define the expansion (see appendix C for more details). This leaves us with the multiplication of a vector by the circulant matrix $C_k$.

Recall that a circulant matrix, $C$, is an $n \times n$ matrix of the form

$$C = \begin{bmatrix} c_0 & c_1 & \cdots & c_{n-1} \\ c_{n-1} & c_0 & \cdots & c_{n-2} \\ \vdots & \vdots & \ddots & \vdots \\ c_1 & c_2 & \cdots & c_0 \end{bmatrix}$$

Because each column of $C$ is a cyclic permutation of the first, the entirety of information contained in the matrix is represented in the first column. This is nice because matrix-vector multiplication of the form

$$\vec{x} = C\vec{v}$$

can be solved as a cyclic convolution of vectors [15]

$$\vec{x} = \vec{c} * \vec{v}$$

where $\vec{c}$ is the first column of C. This is handy because application of the circular convolution theorem [16] allows us calculate the result by component-wise multiplication in Fourier space

$$\vec{x} = \mathcal{F}^{-1}(\mathcal{F}(\vec{c})\mathcal{F}(\vec{v}))$$

where $\mathcal{F}$ and $\mathcal{F}^{-1}$ are the discrete Fourier transform and its inverse

$$\mathcal{F}(\vec{x})_i = \sum_{j=0}^{n-1} x_j e^{-\frac{-2\pi\sqrt{-1}}{n} ij}$$

$$\mathcal{F}^{-1}(\vec{x})_i = \frac{1}{n} \sum_{j=0}^{n-1} x_j e^{\frac{-2\pi\sqrt{-1}}{n} ij}$$

This of course is *especially* nice since we have fast techniques for computing the Fourier transform [17]. By working with circulant matrices the VH algorithm is able to implement its solution with loglinear time FFTs and vector multiplications rather than with the more complex, quadratic time matrix-vector multiplications.

### 2.2.2 Solution

As the reconstruction problem $x = Hf$ is an overdetermined system of linear equations, the VH solver uses linear least squares to approach the solution. Using the circulant matrices defined in section 2.2.1, the partitioned matrix $\mathcal{H}$ is defined as

$$\mathcal{H} = (C_0|\cdots|C_{w-1})$$

where

$$\vec{x} = Hf = \mathcal{H}\left(I_w \otimes E\right)\vec{f}$$

The normal equation

$$\begin{aligned} \vec{y} &= \left(I_w \otimes E\right)\vec{f} \\ \vec{h} &= \mathcal{H}^T x = \mathcal{H}^T\mathcal{H}\vec{y} \end{aligned} \tag{2.2}$$

with the constraint

$$\vec{y} = \left(I_w \otimes E\right)\left(I_w \otimes E^T\right)\vec{y} \tag{2.3}$$

is presented as an alternative to $x = Hf$ since

$$\vec{f} = \left(I_w \otimes E^T\right)\vec{y}$$

The two issues with this approach are

1. Solving (2.2) is problematic since $\mathcal{H}^T\mathcal{H}$ may be ill-conditioned (singular).
2. A solution $y$ satisfying (2.2) might not satisfy (2.3).

The first issue is addressed by working with the regularized problem

$$\vec{h} = \left(\mu I + \mathcal{H}^T\mathcal{H}\right)y \tag{2.4}$$

The introduction of the $\mu I$ term forces the resultant matrix to be non-singular. The VH paper then derives the inverse matrix through application of the Sherman-Morrison-Woodbury formula [18]

$$(A+UV)^{-1} = A^{-1} - \left[A^{-1}U\left(1+VA^{-1}U\right)^{-1}VA^{-1}\right]$$

The Sherman-Morrison formula is commonly used when the inverse of a matrix A is known, and one wants to make some change to A without losing its inverse. The Woodbury formula given above is the extension of the Sherman-Morrison formula to handle multiple correction terms. In general, the expectation is that $U$ and $V$ are much smaller than $A$ and therefore the inverse $\left(1+VA^{-1}U\right)^{-1}$ is easier to compute. Since the inverse of $\mu I$ is trivial, the approach taken is to adjust it with $\mathcal{H}^T$ and $\mathcal{H}$. By letting $A = \mu I$, $U = \mathcal{H}^T$, and $V = \mathcal{H}$, application of the Sherman-Morrison-Woodbury formula gives the solution

$$\vec{y} = \left(\mu^{-1}I - \mu^{-2}\mathcal{H}^T\left(I + \mu^{-1}\sum C_k C_k^T\right)^{-1}\mathcal{H}\right)\vec{h}$$

The VH paper notes that this is a nice result since, as hoped, the matrix

$$I + \mu^{-1}\sum C_k C_k^T$$

is circulant and therefore trivially invertible. Thus an approximate solution to (2.2) is achievable which resolves the first issue.

The second issue, namely that the solution $y$ must satisfy both (2.2) and (2.3) is handled as follows. Define the $nw \times n$ matrix $P$ as

$$P = \mu^{-1}\mathcal{H}^T \left( I + \mu^{-1} \sum C_k C_k^T \right)^{-1/2} \tag{2.5}$$

Define the $nw \times nw$ matrix $M$ as

$$M = \left( \mu I + \mathcal{H}^T \mathcal{H} \right)^{-1} = \mu^{-1} I - PP^T \tag{2.6}$$

and note that

$$M\mathcal{H}^T\mathcal{H} = \mu PP^T \tag{2.7}$$

Next, let

$$\begin{aligned} Z &= \operatorname{diag}\left( \left( I_w \otimes E \right) 1_m \right) \\ Z' &= I - Z \end{aligned}$$

where $1_m$ is the $m \times 1$ vector all of whose entries are 1. Using these definitions, define the constrained series

$$\begin{aligned} \vec{y_0} &= ZM\vec{h} & \vec{y_{i+1}} &= \vec{y_i} + ZM\mathcal{H}^T\mathcal{H}\vec{v_i} \\ \vec{v_0} &= Z'M\vec{h} & \vec{v_{i+1}} &= Z'M\mathcal{H}^T\mathcal{H}\vec{v_i} \end{aligned}$$

From 2.7 and these recursive equations it follows that

$$\begin{aligned} \vec{y_\infty} &= \lim_{i \to \infty} \vec{y_i} \\ &= Z \left( I + \mu PP^T \left\{ I - \mu \left( Z'P \right) \left( Z'P \right)^T \right\}^{-1} Z' \right) M\vec{h} \end{aligned}$$

Application of the Woodbury formula to the inverse in the this expression gives

$$\vec{y_\infty} = Z \left( I + \mu PP^T Z' \left\{ I + P \left( \mu^{-1} I - P^T Z' P \right)^{-1} P^T Z' \right\} \right) M\vec{h} \tag{2.8}$$

Solving for $\vec{y_\infty}$ using the above equation adds the constraints necessary to ensure that the solution meets the requirements of (2.3). Applying the strategy for implementing circulant matrix/vector multiplication laid out in section 2.2.1, the most straight forward approach to (2.8) is to start with $\vec{h}$ and multiply from right to left. The one expression not easily computed has the form $\left( \mu^{-1} I - P^T Z' P \right)^{-1} u$ This problem is dealt with through the use of a conjugate gradient method to approximately solve for $\vec{v}$ in the equation

$$\vec{u} = \left( \mu^{-1} I - P^T Z' P \right) \vec{v}$$

This conjugate gradient method is terminated at the first local minimum of error, or at some predefined number of iterations.

Using the machinery developed in (2.8), the we can proceed to heuristically solve for $\vec{f}$ in $\vec{x} = H\vec{f}$ through iterative refinement (of which several are possible, see [4] for an alternative method). The first approximation of $\vec{f}$, $\vec{f_0}$ is

$$\vec{f_0} = \left(I_w \otimes E^T\right) \vec{y_\infty}$$

Subsequent iterations solve for $\vec{y_\infty}$ with respect to the problem

$$x - H\eta_i \vec{f_i} = H\vec{f}$$

where

$$\vec{r} = H\vec{f_i}$$
$$\eta_i = (\vec{r}^T\vec{r})^{-1}(\vec{r}^T x)$$

The residual is then used to update the estimate

$$\vec{f_{i+1}} = \vec{f_i} + \left(I_w \otimes E^T\right) \vec{y_\infty}$$

Implementing an accelerated version of this algorithm for recovering $f$ is now our primary goal.

# Chapter 3

# The Cell Processor

As mentioned, one of the challenges facing the application of CTIS to large imaging problems is the computationally intense reconstruction process. Using modern desktop computers, CTIS image reconstruction algorithms can run for minutes or hours. While this may be perfectly acceptable for non time-critical applications, when near real time response is needed it becomes desirable to reduce this period to seconds or milliseconds. While algorithmic improvements will undoubtedly have the highest impact on reconstruction time, the implementation of these algorithms on modern hardware can also give a significant potential benefit. Due to its powerful architecture, the Cell processor provides a promising tool to the acceleration of this process.

The focus of this paper is the implementation and acceleration of the Vose-Horton algorithm on the Cell processor. This chapter presents a short review of the Cell Broadband Engine Architecture (CBEA).

## 3.1 Design

The architecture for the Cell processor is the result of a collaboration between Sony, Toshiba, and IBM beginning in 2001 aimed at creating an embedded platform for delivering rich multimedia applications. In order to meet the cost and power budget while delivering high computational throughput the team designing the Cell decided to use a heterogeneous multiprocessor chip with a coherent memory bus [19], [20]. Figure 3.1 illustrates the high level relationship between the various components.

### 3.1.1 The PowerPC Processing Element

The main processor in the Cell architecture it the PowerPC Processing Element (PPE) which is based on IBM's 64-bit PowerPC architecture. This architectural inheritance allows the execution of code built for the PowerPC without modification. It is equipped with a conventional two level cache and an AltiVec SIMD execution unit. The primary purpose of the PPE is to run an operating system and handle the control and logistics of application execution on the Cell.

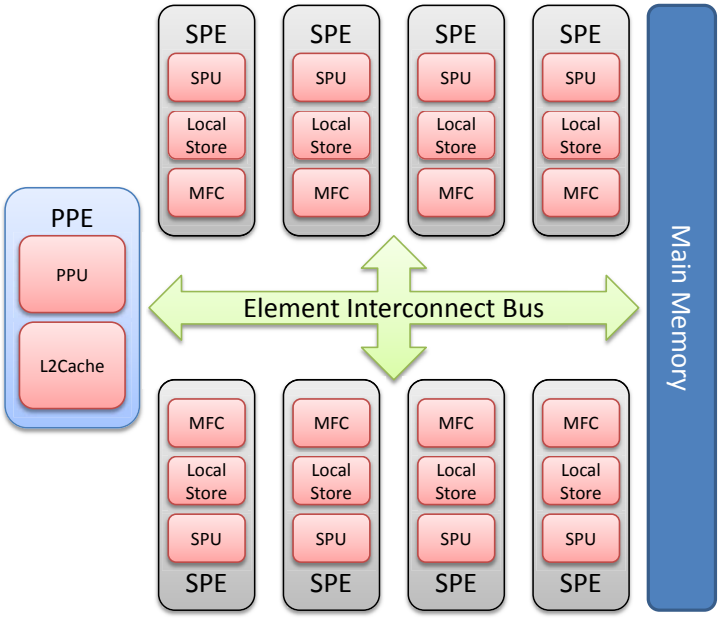



Figure 3.1: The Cell Broadband Architecture

### 3.1.2 The Synergistic Processing Elements

While the PPE is a fairly standard processor, the real workhorses of the Cell processor are a set of computational units called synergistic processing elements (SPE). The two primary components of an SPE are the synergistic processing unit (SPU) and the memory flow controller (MFC). The SPU is a 128-bit RISC processor with a SIMD based instruction set and a unified memory area called the local store. The MFC handles DMA transfers of data and instructions between main memory and the local store of an SPE. Due to the lack of a cache and a simplified execution pipeline, the timing of instruction executing on an SPE is deterministic and allows for very good software scheduling.

The first generation SPEs are geared specifically toward accelerating single precision floating point operations with a fully pipelined implementation being able to schedule one floating point vector operation per cycle. Double precision operations on the other hand are partially pipelined with a 13 clock cycle latency. The result is that double precision operations on the SPEs are significantly slower than single precision operations.

While the SPEs provide a very attractive platform for accelerated floating point computations it is important to note that they deviate slightly from the IEEE 754 standard. The most notable difference is that in single precision the SPE only supports the *round towards zero* rounding mode. The impact of this is that single precision computations on the SPE tend to gain error faster than equivalent operations on a fully IEEE 754 compliant platform.

### 3.1.3 The Element Interconnect Bus

One of the biggest challenges of parallel processing is that of keeping multiple processors "fed" with instruction and data from memory. The element interconnect bus (EIB) was implemented to help meet this challenge and forms the backbone of the Cell and connects the

PPE, SPEs, main memory controller, and other I/O interfaces. The EIB currently consists of four 16-byte wide data rings each of which transfers 128 bytes at a time with an internal maximum bandwidth of 96 bytes per processor clock cycle [21].

## 3.2 Programming

The end product of the innovative hardware design of the Cell processor is a powerful processing platform. This lead to its deployment in the Roadrunner supercomputer which topped the list of fastest supercomputers in the world at the time of this writing in 2009. The next challenge was to create an application development framework for the Cell. While there are various options available, the most common general approach is to use the IBM Cell SDK [22].

The IBM Cell SDK implements the application execution environment for the Cell on top of the Linux operating system. Kernel modules expose an interface to the SPEs through the SPE filesystem which allow them to be utilized from userspace [23]. Additionally, an API for SPE task management and communication is provided through the libspe library. Because the SPEs have a different instruction set architecture from the PPE, they also require a separate cross compiler which is supplied in the SDK.

One attractive platform for evaluating the Cell processor is Sony's PlayStation® 3, which contains a Cell processor at its core. Information about setting up a PlayStation® 3 as a development system are included in appendix B.

# Chapter 4

# Implementation

This chapter describes the implementation of the CTIS solver as well as defining the metrics used to measure the speed and accuracy of the solution. The output of the CTIS reconstruction algorithm, $\hat{f}$, should approximate the value of the vector f in the equation

$$x = Hf$$

Since f has been synthetically generated in order to test the algorithm, it can be directly compared with $\hat{f}$.

## 4.1 Metrics

Before describing the implementation and acceleration of the algorithm, it is necessary to define the metrics used to measure the speed and accuracy of the solution.

### 4.1.1 Accuracy

For the development of various parts of the CTIS solver solution, most of the operations under study are vector functions. Accordingly, the accuracy metrics defined here are used for the comparison of two vectors, x which is the known solution, and $\hat{x}$ which is the approximation under test. Both vectors are of length N.

The per element error of the estimate can be represented as either and absolute or relative quantity. The absolute error is defined as the difference between the values

$$\mid x_i - \hat{x}_i \mid$$

Alternatively, the per element relative error gives the deviation of the approximation from the exact value scaled by the exact value

$$\frac{\mid x_i - \hat{x}_i \mid}{\mid x_i \mid}$$

When taken in context of a vector, the average of either the absolute or relative error (denoted as $r$) is simply the mean

$$\frac{1}{N} \sum_{i=0}^{N-1} r$$

The maximum of either the absolute or relative error is defined as the uniform norm, also known as the supremum norm, also known as the infinity norm, also known as the Chebyshev norm

$$\|r\|_\infty$$

### 4.1.2 Speed

#### Time

When elapsed time is given, it is calculated by recording the output of the Time Base register on the PPE [24] before and after a computation. For computations accelerated on the Cell, the time to allocate the workloads to the SPEs is included in this measurement.

#### FLOPS

Since elapsed time may be too convenient a metric to use for performance, FLoating point Operations Per Second (FLOPS) are also stated for certain measurements. Within the context of this paper, a floating point operation is defined as one logical or arithmetic operation between two real numbers, regardless of the precision. Efforts have been made to distinguish between single and double precision operations where possible. Operations on complex numbers are calculated in terms of the number of real operations required to obtain the complex result. Some examples are

| Operation | Data Type | FLOP(s) |
|---|---|---|
| x+y | float | 1 |
| x*a+y | double | 2 |
| a+b | double complex | 2 |
| a*b | float complex | 6 |

## 4.2 Baseline Implementation

Development of the Vose-Horton solver was handled in four phases:

1. Prototyping
2. Simplification
3. Implementation in C
4. Acceleration

### 4.2.1 Prototyping

The initial prototyping of the solution was handled in Matlab$^{\mathrm{TM}}$, by directly translating equation (2.8)

$$\vec{y_\infty} = Z\left(I + \mu P P^T Z' \left\{I + P\left(\mu^{-1} I - P^T Z' P\right)^{-1} P^T Z'\right\}\right) M\vec{h}$$

into code

```
yLim = Z*(eye(n*w)+mu*P*P'*Zprime*(eye(n*w) + P *
    (((mu^-1)*eye(n)-P'*Zprime*P)^-1)*P'*Zprime))*M*sH'*x;
```

At this stage, all operations were implemented on full matrices. To rework the implementation as intended, the vector-matrix multiplications were then replaced by vector multiplications in the Fourier domain. Recall that this is possible due to the constraint that the operand matrices are circulant and are therefore diagonalized by the discrete Fourier transform.

### 4.2.2 Simplification

During the next stage, functions were consolidated with a deliberate effort to batch computations and avoid switches to the Fourier domain as much as possible.

### 4.2.3 Implementation in C

After the simplified algorithm was debugged and verified, it was ported to C and implemented to run on the PowerPC unit of the Cell processor. The elements required to implement the algorithm were the FFT and a variety of vector operations. Several options were explored, and it was decided to use the existing CELL optimized FFTW library and to construct a library of vector operations. While IBM has made a port of the BLAS available, only a few of the required functions had been optimized to use the SPEs [25].

#### FFTW

The Fastest Fourier Transform in the West (FFTW) library is widely regarded to be one of the fastest portable FFT implementations available [17]. Similar to the technique used in the ATLAS software package [26], the FFTW library experimentally evaluates the performance of various algorithms and parameters in order to determine the fastest FFT configuration for a given problem set and machine architecture. The IBM Austin Research Laboratory has contributed code to the project allowing it to use the SPEs to accelerate the FFT operations. While recent research has demonstrated improved performance FFT codes for certain problem sizes [27] [28], at the time of this writing the FFTW library was still one of the only codes known to the author to support large transforms (ie. $N > 2^{20}$).

#### BLASX

In order to implement the needed vector operations, the somewhat whimsically named *BLAS-eXtended* library was written from scratch. Similar naming conventions were adapted from the real BLAS library in order help code readability. Table 4.1 contains the functions defined by the BLASX library.

For the purpose of interpreting the results of accelerating the BLASX operations, it is useful to note that the precision specified in the function listing refers to the *transfer* precision of the data. Various factors have a bearing on the *computational* precision of an operation, including the processor architecture and the implementation of the function. Of particular note is that the zmultcM and zsxpyM functions have different input and output precisions.

Table 4.1: BLASX Functions

| **Function** | **Precision** | **Operations** | **Formula** |
|---|---|---|---|
| scopy | Single | 1 | $\vec{z} = \vec{x}$ |
| sscale | Single | 1 | $\vec{z} = \alpha\vec{x}$ |
| saxpy | Single | 2 | $\vec{z} = \alpha\vec{x} + \vec{y}$ |
| ssxpy | Single | 2 | $\vec{z} = \alpha\vec{x} - \vec{y}$ |
| sdot | Single | 2 | $z = \vec{x} \cdot \vec{y}$ |
| ccopy | Single Complex | 1 | $\vec{z} = \vec{x}$ |
| cscale | Single Complex | 6 | $\vec{z} = \alpha\vec{x}$ |
| caxpy | Single Complex | 8 | $\vec{z} = \alpha\vec{x} + \vec{y}$ |
| csxpy | Single Complex | 8 | $\vec{z} = \alpha\vec{x} - \vec{y}$ |
| cmult | Single Complex | 6 | $\vec{z} = \vec{x}\vec{y}^T$ |
| cmultc | Single Complex | 6 | $\vec{z} = \bar{\vec{x}}\vec{y}^T$ |
| zmultcM | Mixed | 6 | $\vec{z} = \bar{\vec{x}}\vec{y}^T$ |
| zsxpyM | Mixed | 8 | $\vec{z} = \alpha\vec{x} - \vec{y}$ |
| dcopy | Double | 1 | $\vec{z} = \vec{x}$ |
| dscale | Double | 1 | $\vec{z} = \alpha\vec{x}$ |
| daxpy | Double | 2 | $\vec{z} = \alpha\vec{x} + \vec{y}$ |
| dsxpy | Double | 2 | $\vec{z} = \alpha\vec{x} - \vec{y}$ |
| ddot | Double | 2 | $z = \vec{x} \cdot \vec{y}$ |
| zcopy | Double Complex | 1 | $\vec{z} = \vec{x}$ |
| zscale | Double Complex | 6 | $\vec{z} = \alpha\vec{x}$ |
| zaxpy | Double Complex | 8 | $\vec{z} = \alpha\vec{x} + \vec{y}$ |
| zsxpy | Double Complex | 8 | $\vec{z} = \alpha\vec{x} - \vec{y}$ |
| zmult | Double Complex | 6 | $\vec{z} = \vec{x}\vec{y}^T$ |
| zmultc | Double Complex | 6 | $\vec{z} = \bar{\vec{x}}\vec{y}^T$ |

### Mixed Precision

Mixed precision refers to a technique where part of a computation is performed in double precision, and other parts are handled in single precision [29]. Because the SPEs of the Cell processor are able to perform single precision operations much faster, there is a motivation to use it whenever possible. Additionally, data loaded or stored in single precision required half the amount of memory bandwidth compared with double precision. Both of these factors led to the investigation of a mixed precision implementation of the VH solver to enhance performance.

When determining the feasibility of performing a computation with reduced precision, it is necessary to identify the "numerically sensitive" regions of the algorithm where the system is most susceptible to error. Upon analysis, the VH solver contains two numerically sensitive regions: the dot product operation and $M\vec{h}$.

It's fairly intuitive to see that the dot product, $\hat{x}^T\hat{y}$, may quickly grow large with respect to the L2-norm of $\hat{x}$ and $\hat{y}$. Partly due to the non-standard rounding mode in the SPEs, a

single precision calculation of this value quickly loses accuracy. To help alleviate this, the input vectors and output value for this operation in the BLASX library are single precision while the multiplication and accumulation are performed in double precision. To handle problems with larger parameters, more sophisticated accumulation strategies exist which offer better numeric stability. The interested reader is referred to [30], [31], [32], and [33] for more information.

In addition to the dot product operation, the subtraction operation in $M\vec{h}$ is also numerically sensitive. Recall from equation 2.6 that

$$M = \mu^{-1}I - PP^T$$

When multiplied the products $(\mu^{-1}I)h$ and $(PP^T)h$ have similar values in the first few significant digits. While a characterization of this sensitivity was not undertaken, handling the operation $M\vec{h}$ in extended precision leads to much better convergence.

For more details with regards to the implementation, a pseudocode presentation of the VH solver may be found in appendix C.

# Chapter 5

# Acceleration with the Cell Processor

Up to this point our efforts have focused on minimizing the vector operations and floating point requirements needed to implement the VH solver. We now turn our attention to the use of the Cell processor to accelerate the algorithm. This section lays out the approach and details the incremental refinement of the Cell implementation.

## 5.1 Approach

As already described, the implementation of the VH solver requires only a handful of vector operations and Fourier transforms. As the FFTW library already offers an accelerated Fourier transform implementation for the Cell processor, the remaining task is to implement the vector operations contained in the BLASX library. In order to gauge the effects of various acceleration techniques, an approach of incremental improvement and benchmarking was adopted. The first iteration of the BLASX library is as a naive implementation on the PPE and serves as the starting baseline. At each iteration, the performance of the library is evaluated using metrics defined in section 4.1. The computation speed is compared to that of the previous iteration to derive an *acceleration* statistic

$$\text{Acceleration} = \frac{\text{GFLOPS}_i}{\text{GFLOPS}_{i-1}} - 1$$

where $i$ is the current iteration.

### 5.1.1 Parameters

Since we are primarily interested in operations involving fairly large vectors, $N = 2^{21}$ elements was used as the problem size for all benchmarks in this section. The contents of the test vectors were generated from the *drand48* standard C library call multiplied by 256, resulting in a uniform distribution of values in the interval [0, 256]. Each benchmark was executed 10 times with the average time and error values reported. Code was generated with the GNU GCC compiler version 4.1.1, and optimization level "-O3" was used for all benchmarks. The hardware used was a Sony PlayStation® 3 with a maximum of six SPEs allocated for use. See appendix B for a more detailed hardware description.

## 5.2 Acceleration of the BLASX Library

### 5.2.1 Simple PPE Baseline

The baseline version of the BLASX library is a naive standard C implementation running on the PPE. It should be noted that while the SIMD Altivec instruction set is available for the PPE, this version of the library does not take advantage of it. The timing benchmark is given in table A.1. Note that all tables referenced in this chapter appear in appendix A.

### 5.2.2 Deployment to the SPEs

The next phase in accelerating the BLASX library consisted of moving the computations from the PPE to the SPEs. Distributing work to the processing elements implements the following strategy:

1. The load is broken into 16kB work blocks.
2. Work blocks are distributed evenly to all available SPEs.
3. While the SPEs are running, the PPE processes any remaining elements which could not fill a work block.
4. The PPE waits for the SPEs to complete and then returns from the function.

The performance benchmark comparing the PPE and SPE accelerated implementations is given in table A.2. The increased performance is solely due to parallel processing of the vectors by the SPEs; the actual computation kernel for each function was ported unchanged. The error measurements recorded present the absolute error between the PPE and SPU floating point implementations. These error values remain unchanged in later optimization phases and thus are only presented here.

### 5.2.3 Huge Pages

Because of the fairly large amount of data being pushed back and forth between main memory, virtual address translation overhead can quickly add up. To help alleviate this bottleneck, most modern CPUs implement a Translation Lookaside Buffer or TLB which is able to cache page table entries and speed up address translation.

The size of memory pages being used by an application can have a fairly dramatic impact on its performance based on its working set size [34]. The base page size on the PowerPC platform is 4k, which implies that transferring just one double precision vector from our test set of $2^{21}$ elements will require 4096 memory pages to be hit. The size of the TLB is implementation dependent, but is usually kept fairly small in order to be fast, and is almost certainly smaller than 4096. The Linux kernel provides a mechanism for utilizing larger pages which helps to reduce the number of TLB misses when using a large working set of memory. In this section the BLASX library has been configured to take advantage of 16MB huge pages.

As can be observed from table A.3, the use of huge pages provides a fairly modest performance increase in this case.

### 5.2.4 Vectorization and Unrolling

After reducing memory latency somewhat, we now turn our attention to accelerating the arithmetic operations on the SPEs through vectorization and loop unrolling. Because the SPE only supports quadword loads and stores, scalar code can have a high degree of latency. For a typical scalar operation, a quadword must be loaded, the element shifted into place, operated on, shifted back, and then stored. To take advantage of the vector capabilities of the SPE, a set of C-language extensions have been made available which map directly to SIMD instructions. Using these intrinsics with quadword vectors allows for a much more efficient data handling scheme.

Another useful technique implemented during this stage was loop unrolling. Using the large register file on the SPEs with unrolled loops helps to reduce dependencies and increase the utilization of the dual issue pipeline. A more detailed treatment of these techniques is provided in Appendix E. The acceleration due to vectorization and loop unrolling on the SPEs is given in table A.4.

### 5.2.5 Double Buffering

A challenge for multi processor systems in general, and the Cell in particular, is to keep the computational units fed with a stream of data from memory. The CBEA architecture was specifically designed to be a streaming computation processor with the high bandwidth Element Interconnect Bus providing low latency pipe for data transfer. Because the SPEs lack a cache however, to retrieve a block of data from memory they must initiate a DMA transfer and then wait for it to complete. For compute bound programs operating on streaming data such as some of our BLASX functions, this introduces a latency every time the SPU idles waiting for a data transfer to complete. A buffering scheme overlapping DMA transfers with processing allows for a greater throughput on the SPE. In this optimization phase, a double buffering scheme is implemented, the results of which are presented in table A.5.

As expected, the most compute intensive operations receive the largest benefit of double buffering, while routines already at full bandwidth usage gain nothing. Another aspect worth notice in this benchmark is that, perhaps counter intuitively, the double precision addition and subtraction routines show a slight *decrease* in average performance. A possible explanation for this decrease is that, double buffering has increased memory contention between the SPEs. While double buffering could be selectively disabled for these particular problems, another approach is investigated to alleviate this contention is presented in the next section.

### 5.2.6 Bandwidth Optimization

Conventional wisdom for maximizing memory throughput with the Cell processor is to use as many SPEs with as many asynchronous DMA requests queued as possible. In some situations this can causes an overall throughput reduction when the SPEs are able to overwork the memory subsystem. In workloads such as the functions in the BLASX library which are I/O bound, reducing the number of SPEs assigned to a task can actually result in a speed increase by reducing memory access contention. This phenomenon is examined more closely in [35].

In this section, the number of SPEs assigned to a particular workload has been experimentally tuned to the fewest required to maintain previous performance numbers. The results of this characterization are given in table A.6.

In most of the functions where two input vectors are transferred, reducing the number of SPEs results in an overall throughput increase. In the one exception, zsxpyM, the function is CPU bound and reducing the worker SPEs reduces its throughput. This also sheds light on the apparent speed decrease observed in the daxpy and dsxpy functions when double buffering increased memory bandwidth pressure.

An interesting aspect of this benchmark is that after optimization, the sdot operation is still slightly slower than ddot. This appears somewhat counterintuitive as ddot is transferring double precision vectors while sdot is only transferring single precision vectors, though both are performing double precision arithmetic. However, we can deduce that they are both CPU bound as reducing the number of SPEs results in a speed decrease. The additional time used by sdot is required for the process of shuffling the incoming single precision vectors into double precision registers for computation.

### 5.2.7 Summary

A summary of the effect of the various acceleration techniques presented in this chapter is displayed in figure 5.1.

A fair question to ask at this point is: How does the performance of our library compare to the theoretical maximum? Despite the very fast floating point computation speed of the SPUs, our speed benchmarks in this chapter show only a fraction of the theoretical peak performance of the Cell processor. In research conducted at the Innovative Computing Laboratory at the University of Tennessee, Buttari et. al. estimate that 24 operations must be performed on each single precision value in order to hide the communications cost of moving it from main memory [36]. Because each of these BLASX functions perform only a few operations, they are almost all entirely memory bound.

Because the BLASX functions are memory bound, we shall take as our theoretical best performance the time it takes to move the problem data from memory to the SPEs and back. Referring to appendix B, we find that the peak memory bandwidth on the PlayStation® 3 is 25.6 GB/sec. Taking into account the total number of bytes transferred to and from main memory for each problem, we can estimate how long a round trip would take if there were no CPU processing overhead. These results are compiled in table A.7.

Most of the BLASX function average about 80% peak bandwidth usage. Because they use mixed precision operands, the utilization comparison for the zmultcM and zsxpyM functions is slightly optimistic. One anomaly in this analysis which is difficult to explain is that the ddot function which shows a 93% utilization. Other functions in the library such as dcopy and dscale transfer the same amount of data, with the difference that they read and write a vector while ddot performs two reads.

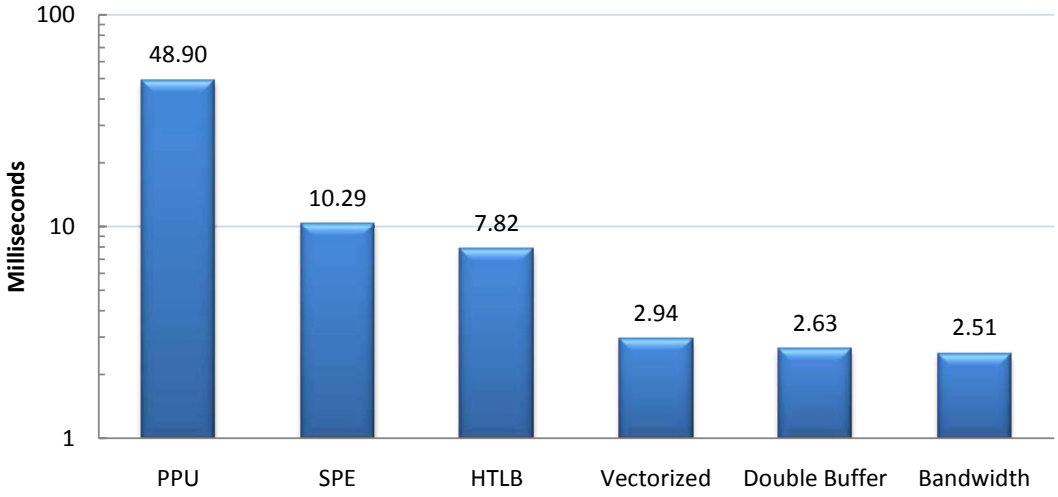



Figure 5.1: Acceleration of average BLASX routine execution time

# Chapter 6

# Results

As described in appendix C, the VH solver is implemented in terms of Fourier transforms and vector operations. With Cell parallelized versions of these functions in place, we now evaluate the performance of the accelerated implementation.

## 6.1 Setup

The parameters used for the solver in this section are given in table 6.1.

Table 6.1: VH Solver Parameters

| | |
|---|---|
| n | 524288 |
| w | 3 |
| $\alpha$ | 225 |
| a | 243 |
| g | 250 |
| $\mu$ | 0.1 |
| CG_ITERATIONS | 15 |
| CG_EPSILON | 0.01 |
| VH_ITERATIONS | 4 |
| VH_EPSILON | 0.1 |

The problem size considered here is somewhat smaller than that presented in [4] due to limited memory space available on the PS3. In keeping with the precedent set in the Vose-Horton paper, a single image was used as the target output in order to demonstrate the qualitative aspects of the reconstruction. The image (shown in figure 6.3) was broken into three "wavelength layers": red, green, and blue. Each element in a layer is a single integer value between 0 and 255 resulting in a composite image color depth of 24 bits. To simulate the operation of a CTIS system the image was multiplied by an H matrix with the resulting vector used as the input to the VH solver. The error estimates were computed by comparing the reconstructed image at each iteration with the original image.

Quantifying the performance of the solver is complicated by the fact that several factors affect the accuracy of the result at any given stage. Because the VH solver is an iterative

heuristic, the time to converge at a target accuracy is also impacted by these factors. Among them are:

- Problem dimensions.
- Range of values in the transfer matrix and input vector.
- Sparseness of the transfer matrix.
- Number of iterations the embedded conjugate gradient may execute.
- Computation precision employed.

As an example, bounding the number of iterations the embedded conjugate gradient may execute allows the trade off of speed versus accuracy for the computed result.

## 6.2 Reconstruction

The results of several implementation variants of the VH solver are presented here for comparison. Each variant is run through 4 iterations of the solver. At each iteration the following information is collected:

**VH Iters** - Iteration of the solver.

**CG Iters** - Number of iterations the embedded conjugate gradient routine executed.

**AbsAvg** - Average absolute error per pixel.

**AbsMax** - Maximum absolute error per pixel.

**RelAvg** - Average relative error per pixel.

**RelMax** - Maximum relative error per pixel.

**Sec** - Seconds in wall time the iteration ran.

The error estimates are generated by comparing the current reconstruction to the original data set.

### 6.2.1 Double Precision

The double precision implementation of the VH solver serves as an reference baseline. The algorithm is run entirely in double precision on the PPE with unaccelerated FFTW and BLASX libraries. Figure 6.2 shows the execution time for four iterations.

Table 6.2: Reconstruction: Double Precision

| VH Iters | CG Iters | AbsAvg | AbsMax | RelAvg | RelMax | Sec |
|---|---|---|---|---|---|---|
| 1 | 14 | 9.56e-01 | 7.09e+00 | 1.22e-01 | 6.23e+00 | 9.98 |
| 2 | 7 | 4.59e-01 | 1.92e+00 | 2.89e-02 | 1.22e+00 | 6.07 |
| 3 | 1 | 4.57e-01 | 1.89e+00 | 2.75e-02 | 1.17e+00 | 2.72 |
| 4 | 1 | 4.56e-01 | 1.86e+00 | 2.65e-02 | 1.11e+00 | 2.72 |
| **Total** | | | | | | **21.48** |

### 6.2.2 Mixed Precision

This variant of the solver implements the mixed precision strategy described in section 4.2.3. Again, all computation is handled on the PPE with unaccelerated libraries. Results are shown in table 6.3.

In this instance it appears that a reduction in the computational precision of the solver results in an *increased* convergence rate for the algorithm. As this is somewhat counterintuitive, it should be reiterated that the problem parameters, test data, and algorithm are identical to that of the double precision solver. The only parameter difference is that the region marked "Extended Precision" in the heuristic kernel (described in section C.2) is implemented in double precision while all other computation is carried out in single precision. The number of conjugate gradient iterations executed is different due to the constraint that the conjugate gradient routine exit at the first local minimum of residual error.

Efforts to localize the difference between the double precision implementation (which we shall refer to as $D$ for the remainder of this section) and the mixed precision implementation, $M$, met with little success. Because the conjugate gradient method ran with a different characteristic between $D$ and $M$, it seemed a logical first place to look. Tracing execution through the first iteration of the solver shows the input vectors to the conjugate gradient to be very close in both $D$ and $M$ (the element-wise maximum difference was 0.73). Figure 6.1 shows the difference of the residual error for each iteration of the conjugate gradient algorithm. $M$'s single precision conjugate gradient stops with a lower residual error than $D$'s. However, as noted in the 1954 paper introducing the conjugate gradient [37] the residual is not expected to be monotonically decreasing, so this difference in behavior may only be related to the difference in input. Also, when the single precision implementation of the conjugate gradient was plugged into $D$, it converged identically to the double precision version with the exception of slight additional rounding error. This leads to the conclusion that the difference in behavior of the conjugate gradient between $D$ and $M$ is an artifact of the difference of their input vector, however slight it may be.

Outside of the conjugate gradient, the next most likely candidate for introducing a measurable perturbation of the estimate vector in our implementation is the Fourier transform. However the double precision implementation should provide better approximations as seen in our later implementation where the FFT operation is moved to the lower precision SPEs and the accuracy is degraded. The reason for why $M$ converges to a better approximation than $D$ in fewer iterations with this parameter set remains undetermined. This behavior is not always consistent as experiments run with other parameters show that in some cases $M$

Table 6.3: Reconstruction: Mixed Precision

| VH Iters | CG Iters | AbsAvg | AbsMax | RelAvg | RelMax | Sec |
|---|---|---|---|---|---|---|
| 1 | 15 | 4.31e-01 | 3.43e+00 | 5.98e-02 | 3.43e+00 | 6.80 |
| 2 | 15 | 1.44e-01 | 3.98e-01 | 3.00e-03 | 1.02e-01 | 6.79 |
| 3 | 1 | 1.44e-01 | 3.91e-01 | 2.88e-03 | 9.33e-02 | 1.82 |
| 4 | 1 | 1.44e-01 | 3.90e-01 | 2.78e-03 | 8.70e-02 | 1.82 |
| **Total** | | | | | | **17.22** |

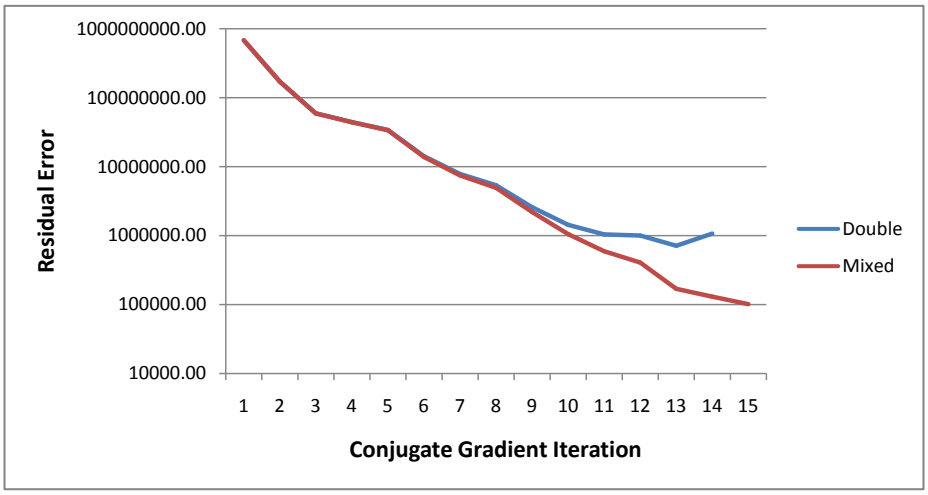



Figure 6.1: Conjugate Gradient Convergence

converges more slowly than $D$.

### 6.2.3 BLASX Accelerated

The BLASX variant implements the mixed precision solver with the accelerated BLASX library; FFT operations are still performed on the PPE. All 6 SPEs of the Cell processor on the PS3 were made available for use by the BLASX library. Results are given in table 6.4.

As expected, the computational output is very close to that of the unaccelerated mixed precision variant. The computational imprecision introduced by the Cell's SPE floating point implementation is minor and can best be seen in comparing the maximum relative errors of the two variants.

### 6.2.4 FFT Accelerated

This variant implements the mixed precision solver with the Cell accelerated FFTW library; the BLASX library implementation is run on the PPE. All 6 SPEs of the Cell processor on the PS3 were made available for use by the FFTW library. Figure 6.5 shows the benchmark results.

Table 6.4: Reconstruction: BLASX Accelerated

| VH Iters | CG Iters | AbsAvg | AbsMax | RelAvg | RelMax | Sec |
|---|---|---|---|---|---|---|
| 1 | 15 | 4.31e-01 | 3.43e+00 | 5.98e-02 | 3.43e+00 | 5.20 |
| 2 | 15 | 1.44e-01 | 4.01e-01 | 3.01e-03 | 1.05e-01 | 5.19 |
| 3 | 1 | 1.44e-01 | 3.93e-01 | 2.88e-03 | 9.71e-02 | 1.33 |
| 4 | 1 | 1.44e-01 | 3.90e-01 | 2.78e-03 | 9.00e-02 | 1.33 |
| **Total** | | | | | | **13.04** |

Table 6.5: Reconstruction: FFT Accelerated

| VH Iters | CG Iters | AbsAvg | AbsMax | RelAvg | RelMax | Sec |
|---|---|---|---|---|---|---|
| 1 | 14 | 7.42e-01 | 5.75e+00 | 9.36e-02 | 4.74e+00 | 3.34 |
| 2 | 7 | 3.67e-01 | 1.94e+00 | 2.78e-02 | 1.17e+00 | 2.07 |
| 3 | 1 | 3.64e-01 | 1.89e+00 | 2.64e-02 | 1.14e+00 | 0.98 |
| 4 | 1 | 3.62e-01 | 1.84e+00 | 2.54e-02 | 1.10e+00 | 0.98 |
| **Total** | | | | | | **7.38** |

While the FFT accelerated variant shows the best performance, the decrease in accuracy is noticeable. According to the implementors, the primary source of this decrease is the rounding mode of the SPEs which always round towards zero. The L2 norm of the relative roundoff error using this mode is approximately 4 to 8 times larger than then when using a round to even scheme [38].

### 6.2.5 BLASX+FFT

The next obvious step in accelerating the VH solver is to use the accelerated versions of both the BLASX and FFTW libraries. Table 6.6 records the effect of using the accelerated FFTW and BLASX libraries with both libraries given access to all 6 SPEs as above.

As might be anticipated, the performance is abysmal. Both libraries make use of all 6 allocated SPEs requiring the operating system to perform context switching between them. Because preemptive context switching on the SPEs is expensive, most of the time used in this variant is spend switching the libraries out. See section 7.1.2 for a more detailed discussion of this issue.

### 6.2.6 BLASX+FFT Split

The last variant we present is similar to the previous in that both the BLASX and FFTW libraries use their accelerated implementations. However, instead of being allocated all 6 SPEs the FFTW library is assigned 4 while the BLASX library is given 2. By splitting the number of SPEs allocated to the libraries context switching is eliminated and they no longer compete for resource. The results are given in table 6.7.

Table 6.6: Reconstruction: BLASX+FFT

| VH Iters | CG Iters | AbsAvg | AbsMax | RelAvg | RelMax | Sec |
|---|---|---|---|---|---|---|
| 1 | 14 | 7.42e-01 | 5.75e+00 | 9.36e-02 | 4.74e+00 | 15.64 |
| 2 | 7 | 3.67e-01 | 1.94e+00 | 2.78e-02 | 1.17e+00 | 10.07 |
| 3 | 1 | 3.64e-01 | 1.89e+00 | 2.64e-02 | 1.14e+00 | 5.27 |
| 4 | 1 | 3.62e-01 | 1.84e+00 | 2.54e-02 | 1.10e+00 | 5.27 |
| **Total** | | | | | | **36.25** |

Table 6.7: Reconstruction: BLASX+FFT Split

| VH Iters | CG Iters | AbsAvg | AbsMax | RelAvg | RelMax | Sec |
|---|---|---|---|---|---|---|
| 1 | 14 | 7.42e-01 | 5.75e+00 | 9.36e-02 | 4.74e+00 | 1.93 |
| 2 | 7 | 3.67e-01 | 1.94e+00 | 2.78e-02 | 1.17e+00 | 1.17 |
| 3 | 1 | 3.64e-01 | 1.89e+00 | 2.64e-02 | 1.14e+00 | 0.52 |
| 4 | 1 | 3.62e-01 | 1.84e+00 | 2.54e-02 | 1.10e+00 | 0.52 |
| **Total** | | | | | | **4.14** |

As expected, the accuracy measurements track those of the FFTW accelerated implementation leading to the conclusion that the convergence behavior is in this case is bounded by the accuracy of that library. This variant represents the fastest implementation of the VH solver that we have constructed.

## 6.3 Summary

Figure 6.2 displays the comparative speeds for 4 iterations of each variant of the VH solver that we benchmarked. Figure 6.3 shows the original image with the estimated reconstruction after passes 1 and 4 of the BLASX+FFTW Split variant.

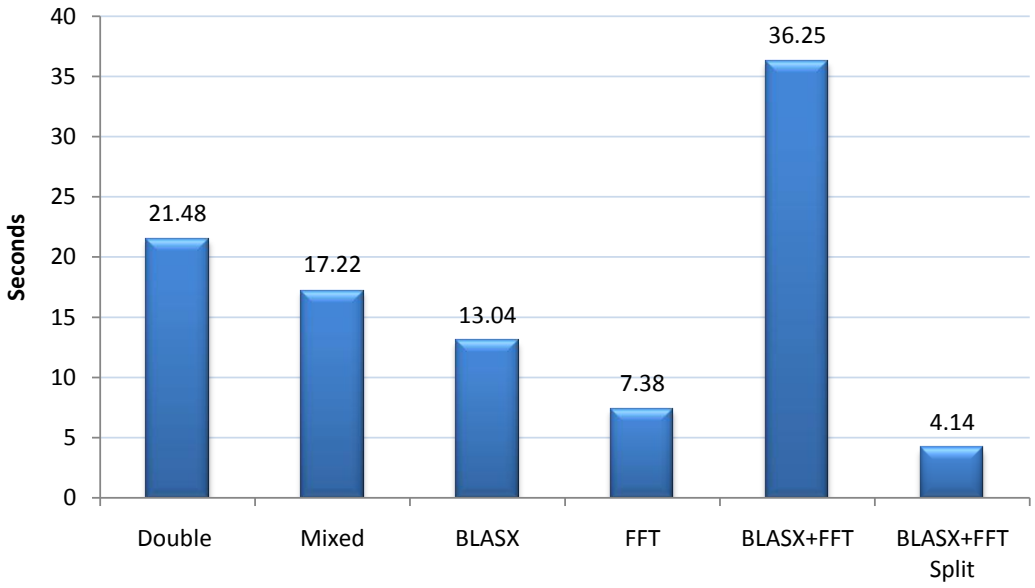
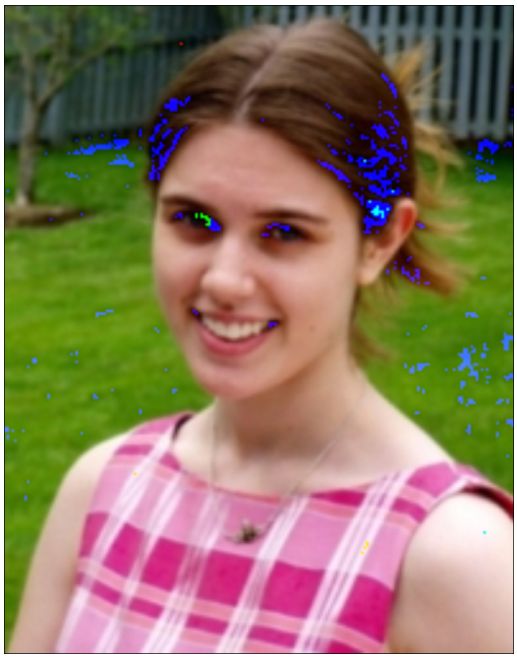



Figure 6.2: CTIS Reconstruction Performance - 4 passes

(a) Original Image



(b) Reconstruction after 1 pass

(c) Reconstruction after 4 passes

Figure 6.3: Sample image reconstruction with BLASX+FFT Split

# Chapter 7

# Conclusions and Future Work

## 7.1 Future Work

Though this paper demonstrates a 5 fold speed increase by parallelizing the VH solver, several optimization paths lay open for future research and development to increase the algorithm's performance even further. Reducing the number of Fourier transforms required in the computation, integrating the Cell libraries, and tuning the implementation to the problem characteristics could all have a positive performance impact.

### 7.1.1 Reducing Fourier transforms

Perhaps both the strength and weakness of the Vose-Horton solver to the CTIS reconstruction problem is its heavy use of the Fourier transform. It is a strength in that using an FFT makes very expensive vector/matrix multiplications much cheaper. The disadvantage is that the VH solver makes frequent use of them in the reconstruction. The nature of the Fourier transform requires the entire vector to be processed. Thus, every invocation of an FFT creates a data dependency on prior operations resulting in a computational pipeline stall every time it is invoked, limiting parallelism. Further, large vectors cause the memory subsystem to quickly becomes the primary bottleneck, since each FFT operation requires a round trip to main memory.

Finding a way to consolidate the time spent in Fourier space (or vice-a-versa) will certainly result in a more efficient algorithm. Because this would greatly benefit the solver implementation regardless of the platform on which it was run, it is perhaps the most attractive for future optimization.

### 7.1.2 Handling the Cell Library Problem

While the CBE architecture delivers significant performance advantages as an embedded platform, the lack of a widespread standard library interface makes it difficult to use as an application platform. As shown in section 6.2.5, two or more standalone libraries written to utilize the SPEs directly will interact poorly with each other. A major reason for this is that preemptively switching context on the SPE is an expensive operation requiring more than 258 KB of storage space per SPE [21]. With such a high preemptive multitasking overhead it becomes critical for libraries to implement a cooperative multitasking interface in order to utilize shared resources efficiently. Two potential candidates for this framework are

IBM's Accelerated Library Framework (ALF) [39] and Sony's Multicore Application Runtime System MARS. While these efforts provide a point in the right direction, at the time of this writing neither one has gained widespread community adoption, and not even IBM's libraries implement their ALF interface.

The approach taken to resolve the issue in this paper was to split the resources (SPEs) between the libraries. However, better resource utilization could be achieved by porting the libraries to one of the afore mentioned frameworks or integrating them manually.

### 7.1.3 Tuning the Implementation

The implementation presented in this paper was tuned to display the performance of the algorithm processing a moderate sized problem on the Cell processor in a PlayStation® 3. While we have demonstrated a modest performance increase on this platform, an implementation may achieve better performance when tuned to the characteristics of the CTIS system for which it is intended. Problems with smaller sized vectors or better range bounds on the reconstruction matrix may achieve acceptable results by reducing the precision requirements on the FFT or mixed precision region used in this thesis. Also, as previously mentioned there are currently FFT codes available for the Cell with faster performance with smaller vector sizes [27] [28]. Problems characterized by larger vectors will require greater amounts of memory, and special handling for roundoff error.

## 7.2 Conclusion

In summary, this paper presents a parallelized implementation of the Vose-Horton CTIS reconstruction technique on the Cell processor which shows an approximate five fold performance increase over a serial implementation. Additionally, the analysis demonstrates the feasibility and performance benefit of implementing the solver with mixed precision arithmetic. Finally, the results have shown the algorithm to be essentially memory bound and possible avenues of research to improve performance further have been presented.

# Bibliography

# Bibliography


[1] J. F. Scholl, E. K. Hege, M. Hart, D. OConnell, , and E. L. Dereniak, "Flash hyperspectral imaging of non-stellar astronomical objects," in *Mathematics of Data/Image Pattern Recognition, Compression, and Encryption with Applications XI* (M. S. Schmalz, G. X. Ritter, J. Barrera, and J. T. Astola, eds.), 2008.

[2] B. Karaçali and W. Snyder, "Automatic target detection using multispectral imaging," in *Applied Imagery Pattern Recognition Workshop*, pp. 55– 59, 2002.

[3] B. Ford, M. Descour, and R. Lynch, "Large-image-format computed tomography imaging spectrometer for fluorescence microscopy," *Optics Express*, vol. 9, pp. 444–453, 2001.

[4] M. Vose and M. Horton, "A heuristic technique for ctis image-reconstruction," *Applied Optics*, vol. 46, no. 26, pp. 6498–6503, 2007.

[5] N. Hagen, E. L. Dereniak, and D. T. Sass, "Fourier methods of improving reconstruction speed for ctis imaging spectrometers," vol. 6661 of *Imaging Spectrometry XII*, Proceedings of SPIE, 2007.

[6] L. Sethaphong, "Large format ctis in real time: Parallelized algorithms and preconditioning initializers," Master's thesis, Vanderbilt University, 2008.

[7] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "Scientific computing kernels on the cell processor," *International Journal of Parallel Programming*, vol. 35, pp. 263–298, June 2007.

[8] T. Okamoto and I. Yamaguchi, "Simultaneous acquisition of spectral image information," *Optics Letters*, vol. 16, pp. 1277–1279, 1991.

[9] T. Okamoto, A. Takahashi, and I. Yamaguchi, "Simultaneous acquisition of spectral and spatial intensity distribution," *Applied Spectroscopy*, vol. 47, pp. 1198–1202, 1993.

[10] M. Descour and E. Dereniak, "Computed-tomography imaging spectrometer: experimental calibration and reconstruction results," *Applied Optics*, vol. 34, pp. 4817–4826, 1995.

[11] M. R. Descour, C. E. Volin, E. L. Dereniak, T. M. Gleeson, M. F. Hopkins, D. W. Wilson, , and P. D. Maker, "Demonstration of a computed-tomography imaging spectrometer using a computer-generated hologram disperser," *Applied Optics*, vol. 36, pp. 3694–3698, 1997.

[12] R. Gordon, R. Bender, and G. Herman, "Algebraic reconstruction techniques (art) for three-dimensional electron microscopy and x-ray photography," *Journal of Theoretical Biology*, vol. 29, pp. 471–481, 1970.

[13] D. Mishra, J. P. Longtin, R. P. Singh, and V. Prasad, "Performance evaluation of iterative tomography algorithms for incomplete projection data," *Applied Optics*, vol. 43, no. 7, pp. 1522–1532, 2004.

[14] J. P. Garcia and E. L. Dereniak, "Mixed-expectation image-reconstruction technique," *Applied Optics*, vol. 38, pp. 3745–3748, 1999.

[15] G. H. Golub and C. F. V. Loan, *Matrix Computation.* Johns Hopkins Studies in Mathematical Sciences, 3rd. ed., 1996.

[16] E. O. Brigham, *The Fast Fourier Transform.* Prentice-Hall, Inc., 1974.

[17] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[18] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 3rd. ed., 2007.

[19] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "A novel simd architecture for the cell heterogeneous chip-multiprocessor," 2005.

[20] M. Gschwind, "The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor," tech. rep., IBM Research Division Thomas J. Watson Research Center, 2006.

[21] International Business Machines Corporation and Sony Computer Entertainment Incorporated and Toshiba Corporation, *Cell Broadband Engine Programming Handbook*, 1.1 ed., April 2007.

[22] I. B. M. Corporation, "Cell broadband engine resource center." http://www.ibm.com/developerworks/power/cell/.

[23] A. Arevalo, R. M. Matinata, M. Pandian, E. Peri, K. Ruby, F. Thomas, and C. Almond, *Programming the Cell Broadband Engine Architecture: Examples and Best Practices.* IBM Corp., 2008.

[24] IBM, *PowerPC Virtual Environment Architecture Book II*, 2.02 ed., January 2005.

[25] IBM, *Basic Linear Algebra Subprograms Library Programmers Guide and API Reference*, 3.1 ed., 2008.

[26] R. C. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, pp. 101–121, February 2005. http://www.cs.utsa.edu/ whaley/papers/spercw04.ps.

[27] D. Bader and V. Agarwal, "Fftc: Fastest fourier transform for the ibm cell broadband engine," vol. 4873, pp. 172–184, LNCS, 2007.

[28] S. Chellappa, F. Franchetti, and M. Püschel, "Computer generation of fast Fourier transforms for the cell broadband engine," in *International Conference on Supercomputing (ICS)*, 2009.

[29] J. Kurzak and J. Dongarra, "Implementation of the mixed-precision high performance linpack benchmark on the cell processor," Technical Report UT-CS-06-580, University of Tennessee Knoxville, Department of Computer Science, Sept. 2006.

[30] I. J. Anderson, "A distillation algorithm for floating-point summation," *SIAM J. Sci. Comput*, vol. 20, pp. 1797–1806, 1999.

[31] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, vol. 2. Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 1997.

[32] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo, "Design, implementation and testing of extended and mixed precision blas," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 152–205, 2002.

[33] N. Yamanaka, T. Ogita, S. Rump, and S. Oishi, "A parallel algorithm for accurate dot product," *Parallel Computing*, vol. 34, no. 6-8, pp. 392 – 410, 2008.

[34] J. Navarro, *Transparent operating system support for superpages*. PhD thesis, Rice University, April 2004.

[35] T. Saidani, L. Lacassagne, S. Bouaziz, and T. M. Khan, *Parallel and Distributed Processing and Applications*, ch. Parallelization Strategies for the Points of Interests Algorithm on the Cell Processor, pp. 104–112. Springer Berlin / Heidelberg, 2007.

[36] A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra, and G. Boslica, "A rough guide to scientific computing on the playstation 3," Technical Report UT-CS-07-595, University of Tennessee Knoxville, Innovative Computing Laboratory, May 2007.

[37] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, 1952.

[38] M. Frigo and S. Johnson, *FFTW 3.2.2 Manual*. Massachusetts Institute of Technology, July 2009.

[39] International Business Machines Corporation, *Accelerated Library Framework Programmers Guide and API Reference*, 2008.

# Appendices

# Appendix A

# Tables

Table A.1: BLASX Performance: PPE Simple

| **Function** | **msec** | **GFLOPS** |
|---|---|---|
| scopy | 16.12 | 0.13 |
| sscale | 19.16 | 0.11 |
| saxpy | 28.59 | 0.15 |
| ssxpy | 28.13 | 0.15 |
| sdot | 23.30 | 0.18 |
| ccopy | 26.59 | 0.08 |
| cscale | 32.22 | 0.39 |
| caxpy | 55.21 | 0.30 |
| csxpy | 55.19 | 0.30 |
| cmult | 48.36 | 0.26 |
| cmultc | 60.26 | 0.21 |
| zmultcM | 57.16 | 0.22 |
| zsxpyM | 97.86 | 0.17 |
| dcopy | 23.44 | 0.09 |
| dscale | 27.56 | 0.08 |
| daxpy | 45.64 | 0.09 |
| dsxpy | 44.98 | 0.09 |
| ddot | 39.74 | 0.11 |
| zcopy | 44.19 | 0.05 |
| zscale | 46.75 | 0.27 |
| zaxpy | 89.38 | 0.19 |
| zsxpy | 89.37 | 0.19 |
| zmult | 83.94 | 0.15 |
| zmultc | 90.47 | 0.14 |
| **Total** | **1173.61** | **4.10** |

Table A.2: BLASX Performance: SPU Simple

| **Function** | **msec** | **GFLOPS** | **Average Error** | **Max Error** | **Acceleration** |
|---|---|---|---|---|---|
| scopy | 2.72 | 0.77 | 0.00E+00 | 0.00E+00 | 492% |
| sscale | 3.40 | 0.62 | 5.16E-04 | 1.95E-03 | 464% |
| saxpy | 4.54 | 0.92 | 1.24E-03 | 3.91E-03 | 513% |
| ssxpy | 4.52 | 0.93 | 5.86E-05 | 2.44E-04 | 520% |
| sdot | 4.74 | 0.89 | 0.00E+00 | 0.00E+00 | 394% |
| ccopy | 5.70 | 0.37 | 0.00E+00 | 0.00E+00 | 363% |
| cscale | 8.17 | 1.54 | 3.75E-03 | 1.10E-02 | 295% |
| caxpy | 14.47 | 1.16 | 2.57E-03 | 8.73E-03 | 287% |
| csxpy | 14.42 | 1.16 | 2.69E-03 | 9.77E-03 | 287% |
| cmult | 10.38 | 1.21 | 2.82E-03 | 1.10E-02 | 365% |
| cmultc | 10.41 | 1.21 | 2.76E-03 | 1.10E-02 | 476% |
| zmultcM | 16.88 | 0.75 | 0.00E+00 | 0.00E+00 | 241% |
| zsxpyM | 16.00 | 1.05 | 0.00E+00 | 0.00E+00 | 518% |
| dcopy | 4.22 | 0.50 | 0.00E+00 | 0.00E+00 | 456% |
| dscale | 5.43 | 0.39 | 0.00E+00 | 0.00E+00 | 388% |
| daxpy | 7.31 | 0.57 | 0.00E+00 | 0.00E+00 | 533% |
| dsxpy | 7.32 | 0.57 | 0.00E+00 | 0.00E+00 | 533% |
| ddot | 3.42 | 1.23 | 2.26E-10 | 4.73E-04 | 1018% |
| zcopy | 8.54 | 0.25 | 0.00E+00 | 0.00E+00 | 400% |
| zscale | 13.69 | 0.92 | 0.00E+00 | 0.00E+00 | 241% |
| zaxpy | 22.89 | 0.73 | 0.00E+00 | 0.00E+00 | 284% |
| zsxpy | 22.88 | 0.73 | 0.00E+00 | 0.00E+00 | 284% |
| zmult | 17.44 | 0.72 | 0.00E+00 | 0.00E+00 | 380% |
| zmultc | 17.46 | 0.72 | 0.00E+00 | 0.00E+00 | 414% |
| **Total** | **246.95** | **19.91** | | | |

Table A.3: BLASX Performance: SPU, HTLB

| **Function** | **msec** | **GFLOPS** | **Acceleration** |
|---|---|---|---|
| scopy | 2.11 | 0.99 | 29% |
| sscale | 2.76 | 0.76 | 23% |
| saxpy | 3.20 | 1.31 | 42% |
| ssxpy | 3.17 | 1.32 | 42% |
| sdot | 4.43 | 0.95 | 7% |
| ccopy | 4.22 | 0.50 | 35% |
| cscale | 6.84 | 1.84 | 19% |
| caxpy | 12.06 | 1.39 | 20% |
| csxpy | 12.11 | 1.38 | 19% |
| cmult | 7.96 | 1.58 | 31% |
| cmultc | 7.89 | 1.59 | 31% |
| zmultcM | 13.71 | 0.92 | 23% |
| zsxpyM | 11.84 | 1.42 | 35% |
| dcopy | 2.45 | 0.86 | 72% |
| dscale | 3.89 | 0.54 | 38% |
| daxpy | 4.96 | 0.85 | 49% |
| dsxpy | 4.96 | 0.85 | 49% |
| ddot | 2.66 | 1.58 | 28% |
| zcopy | 4.86 | 0.43 | 72% |
| zscale | 10.42 | 1.21 | 32% |
| zaxpy | 18.23 | 0.92 | 26% |
| zsxpy | 18.19 | 0.92 | 26% |
| zmult | 12.34 | 1.02 | 42% |
| zmultc | 12.47 | 1.01 | 40% |
| **Total** | **187.73** | **26.14** | |

Table A.4: BLASX Performance: SPU, HTLB, Vectorized

| **Function** | **msec** | **GFLOPS** | **Acceleration** |
|---|---|---|---|
| scopy | 0.78 | 2.67 | 170% |
| sscale | 0.78 | 2.68 | 253% |
| saxpy | 1.27 | 3.30 | 152% |
| ssxpy | 1.27 | 3.31 | 151% |
| sdot | 1.63 | 2.57 | 171% |
| ccopy | 1.55 | 1.35 | 170% |
| cscale | 1.55 | 8.14 | 342% |
| caxpy | 2.53 | 6.62 | 376% |
| csxpy | 2.53 | 6.62 | 380% |
| cmult | 2.53 | 4.97 | 215% |
| cmultc | 2.53 | 4.97 | 213% |
| zmultcM | 5.48 | 2.30 | 150% |
| zsxpyM | 6.11 | 2.74 | 93% |
| dcopy | 1.55 | 1.36 | 58% |
| dscale | 1.55 | 1.35 | 150% |
| daxpy | 2.53 | 1.66 | 95% |
| dsxpy | 2.53 | 1.66 | 95% |
| ddot | 1.36 | 3.07 | 94% |
| zcopy | 3.07 | 0.68 | 58% |
| zscale | 3.37 | 3.74 | 209% |
| zaxpy | 6.27 | 2.68 | 191% |
| zsxpy | 6.20 | 2.71 | 195% |
| zmult | 5.77 | 2.18 | 114% |
| zmultc | 5.78 | 2.18 | 116% |
| **Total** | **70.52** | **75.51** | |

Table A.5: BLASX Performance: SPU, HTLB, Vectorized, Double Buffer

| **Function** | **msec** | **GFLOPS** | **Acceleration** |
|---|---|---|---|
| scopy | 0.78 | 2.69 | 1% |
| sscale | 0.78 | 2.69 | 0% |
| saxpy | 1.27 | 3.30 | 0% |
| ssxpy | 1.27 | 3.30 | 0% |
| sdot | 1.40 | 3.00 | 17% |
| ccopy | 1.55 | 1.36 | 1% |
| cscale | 1.55 | 8.14 | 0% |
| caxpy | 2.53 | 6.64 | 0% |
| csxpy | 2.54 | 6.62 | 0% |
| cmult | 2.54 | 4.95 | 0% |
| cmultc | 2.54 | 4.96 | 0% |
| zmultcM | 4.16 | 3.02 | 31% |
| zsxpyM | 4.24 | 3.96 | 45% |
| dcopy | 1.54 | 1.36 | 0% |
| dscale | 1.55 | 1.35 | 0% |
| daxpy | 2.54 | 1.65 | -1% |
| dsxpy | 2.54 | 1.65 | -1% |
| ddot | 1.35 | 3.11 | 1% |
| zcopy | 3.06 | 0.68 | 0% |
| zscale | 3.08 | 4.09 | 9% |
| zaxpy | 5.06 | 3.32 | 24% |
| zsxpy | 5.07 | 3.31 | 22% |
| zmult | 5.06 | 2.48 | 14% |
| zmultc | 5.08 | 2.48 | 14% |
| **Total** | **63.08** | **80.11** | |

Table A.6: BLASX Performance: SPU, HTLB, Vectorized, Double Buffer, Bandwidth

| **Function** | **SPEs** | **msec** | **GFLOPS** | **Acceleration** |
|---|---|---|---|---|
| scopy | 2 | 0.78 | 2.69 | 0% |
| sscale | 3 | 0.78 | 2.69 | 0% |
| saxpy | 3 | 1.15 | 3.65 | 11% |
| ssxpy | 3 | 1.15 | 3.66 | 11% |
| sdot | 6 | 1.41 | 2.97 | -1% |
| ccopy | 2 | 1.54 | 1.36 | 0% |
| cscale | 3 | 1.54 | 8.18 | 0% |
| caxpy | 2 | 2.29 | 7.33 | 10% |
| csxpy | 2 | 2.29 | 7.31 | 10% |
| cmult | 2 | 2.29 | 5.50 | 11% |
| cmultc | 2 | 2.28 | 5.51 | 11% |
| zmultcM | 5 | 4.14 | 3.04 | 1% |
| zsxpyM | 6 | 4.23 | 3.96 | 0% |
| dcopy | 3 | 1.54 | 1.36 | 0% |
| dscale | 3 | 1.54 | 1.36 | 1% |
| daxpy | 3 | 2.26 | 1.86 | 13% |
| dsxpy | 3 | 2.27 | 1.85 | 12% |
| ddot | 6 | 1.35 | 3.11 | 0% |
| zcopy | 3 | 3.06 | 0.68 | 0% |
| zscale | 6 | 3.08 | 4.09 | 0% |
| zaxpy | 4 | 4.89 | 3.43 | 3% |
| zsxpy | 4 | 4.89 | 3.43 | 4% |
| zmult | 3 | 4.70 | 2.67 | 8% |
| zmultc | 3 | 4.70 | 2.68 | 8% |
| **Total** | | **60.16** | **84.37** | |

Table A.7: BLASX Performance Evaluation

| **Function** | **Type size** | **Count** | **Peak Time** | **BLASX Time** | **Utilization** |
|---|---|---|---|---|---|
| scopy | 4 | 2 | 0.62 | 0.78 | 80% |
| sscale | 4 | 2 | 0.62 | 0.78 | 80% |
| saxpy | 4 | 3 | 0.94 | 1.15 | 82% |
| ssxpy | 4 | 3 | 0.94 | 1.15 | 82% |
| sdot | 4 | 2 | 0.62 | 1.41 | 44% |
| ccopy | 8 | 2 | 1.25 | 1.54 | 81% |
| cscale | 8 | 2 | 1.25 | 1.54 | 81% |
| caxpy | 8 | 3 | 1.87 | 2.29 | 82% |
| csxpy | 8 | 3 | 1.87 | 2.29 | 82% |
| cmult | 8 | 3 | 1.87 | 2.29 | 82% |
| cmultc | 8 | 3 | 1.87 | 2.28 | 82% |
| zmultcM | 16 | 3 | 3.75 | 4.14 | 91% |
| zsxpyM | 16 | 3 | 3.75 | 4.23 | 89% |
| dcopy | 8 | 2 | 1.25 | 1.54 | 81% |
| dscale | 8 | 2 | 1.25 | 1.54 | 81% |
| daxpy | 8 | 3 | 1.87 | 2.26 | 83% |
| dsxpy | 8 | 3 | 1.87 | 2.27 | 83% |
| ddot | 8 | 2 | 1.25 | 1.35 | 93% |
| zcopy | 16 | 2 | 2.50 | 3.07 | 81% |
| zscale | 16 | 2 | 2.50 | 3.08 | 81% |
| zaxpy | 16 | 3 | 3.75 | 4.89 | 77% |
| zsxpy | 16 | 3 | 3.75 | 4.89 | 77% |
| zmult | 16 | 3 | 3.75 | 4.70 | 80% |
| zmultc | 16 | 3 | 3.75 | 4.70 | 80% |

# Appendix B

# Cell Processor on the PlayStation 3

One of the great features of Sony's PlayStation® 3 (PS3) entertainment system is that it contains a Cell processor at it's core. Combined with Sony's explicit support for running other operating systems on it, the price of a typical PS3 makes this platform attractive as an inexpensive alternative to other Cell based hardware.

## B.1 The Hardware

While very powerful as an entertainment console, one has to be mindful of the limitations of the PS3 when deciding whether or not to invest in it as a tool for developing or running Cell applications.

### B.1.1 CPU

The processor is a 3.2 GHz power processor core with 512KB L2 cache and seven active Synergistic Processing Elements. In order to increase production yield, Sony has elected to activate only seven of the eight SPEs. Additionally, when running Linux or any OS other than Sony's, one SPE is used to run a hypervisor which limits access to the GPU and other protected parts of the PS3 (presumably to make it more difficult to 'hack' the system). This reduces the number of SPEs available for general use to six.

### B.1.2 Memory

The system memory consists of 256 MB of Rambus Extreme Data Rate (XDR) RAM with a peak maximum bandwidth of 25.6 GB/s. When running Linux, the maximum available system memory is around 220 MB, presumably due to memory reserved for use by the above mentioned hypervisor.

### B.1.3 Network

The network interfaces consist of an 802.11 b/g wireless NIC, a 10/100/1000 Ethernet port, and a Bluetooth 2.0 transceiver.

### B.1.4 Storage

The PS3 comes installed with a standard 5400 RPM, 2.5 inch SATA hard drive, the capacity of which depends on the model. If more storage or disk performance is needed, it is simple to upgrade the PS3 with an off the shelf laptop hard drive. Step-by-step instructions and photos of this procedure can be found at multiple sites online.

### B.1.5 I/O

The I/O ports available depend on the model, but as of this writing (2009), all models come with at least 2 USB 2.0 ports. Some models additionally have memory card readers, but this capability has been removed in later series PS3s.

## B.2 Installing Linux on the PlayStation 3

While the Sony's firmware for the PS3 has nice gaming and media features, to get real work done you will need to install Linux. There are many distributions which support the PS3, but since Fedora is the one supported by IBM's Cell SDK, we will focus our efforts there. Before getting started, we will need to gather some required tools:

- A mouse and keyboard with USB interfaces.
- A USB storage device such as a thumb drive or hard drive.
- A DVD with the version of Fedora that you would like to load (note that the PowerPC version is required).

It is also recommended that you update the PS3 Sony OS to the latest version, and perform a full backup of any game and media files on the system before you begin.

### B.2.1 Partitioning the Hard Drive

The first step in getting Linux onto a PS3 is to create a partition on the hard drive to hold the OS. From the Sony OS go to: Settings → System Settings → Format Hard Drive. From there you can choose how to allocate the storage space, with the options being either 10 GB for the Sony OS, and the remainder for Linux, or vice-a-versa. If the primary purpose of the machine is Linux development, then it is probably wise to give it the bulk of the hard drive storage.

### B.2.2 Setting up a Bootloader

After partitioning the hard drive, the next step in the installation process is to install a bootloader capable of brining up a Linux installation. As of this writing, the two primary options are KBoot and Petitboot.

KBoot was the first bootloader to support the PS3, and has a console based interface. It is more sophisticated than some other bootloaders in that is loads a small Linux kernel and provides a basic running system. It then uses the kexec call to execute the full kernel

and boot the system. At the time of this writing, KBoot has been deprecated in favor of Petitboot.

Petitboot is a graphical bootloader designed specifically for the PS3, and is built on top of Kboot and the twin windowing system. It provides a graphical interface to select which system to load. While Petitboot is more graphically appealing and works well for most configurations, if you run into problems with a customized kernel it does not provide many recovery options.

After downloading the bootloader of your choice, you'll need to put it on your PS3. Simple rename the bootloader to other.bld, put it into a folder named ps3 on a USB drive, put it into the system and select "Load Other OS" from the system menu. Sony has also graciously provided step-by-step instructions on their website.

## B.3 Installing Linux

This part is actually fairly straightforward now that Fedora is well supported on the PS3. Put the Fedora DVD which you burned earlier into the drive. If using Petiteboot, you will be given the choice of which kernel to load. Use the 64-bit kernel as the PS3 won't run the 32-bit version. After answering the standard installation questions, you will be on on your way. Note that the default configuration of the hard drive is to use the Logical Volume Manager which can be a bit inconvenient when rebuilding the kernel. I recommend laying out the system with a standard boot, root, and swap partition.

## B.4 Pruning the System

Like many Linux distributions, Fedora is configured out of the box with a fairly general setup. As the PS3 has relatively little memory space for extra applications, I recommend stripping down the system as much as possible after installation. To automate this I run the following script post install:

```
#!/bin/sh

#
# To ensure that nothing is getting swapped out
# behind my back, turn off swap entirely.
#
sed -i 's/.*swap.*/#&/' /etc/fstab

#
# There's no need for SELINUX to be deployed on this
# research system.
#
sed -i 's/^SELINUX=.*/SELINUX=disabled/' /etc/selinux/config

#
# Turn off all but the essential startup daemons and services.
#
KEEP_STARTUP="network udev-post messagebus sshd"
```

```
STARTUP_PROGS=‘chkconfig --list | awk ’{print $1}’‘

# Turn everything off
for P in $STARTUP_PROGS; do
    chkconfig --level 0123456 $P off
done

# Reenable just the scripts we want to run on startup
for P in $KEEP_STARTUP; do
    chkconfig $P on
done

#
# Turn off all the extra ttys running on the console
# (I only need one for debugging)
#
mkdir -p /etc/event.d_disabled
for i in 2 3 4 5 6; do
    mv /etc/event.d/tty${i} /etc/event.d_disabled
done
```

## B.5 Updating the Linux Kernel

While the Fedora kernel provides a running system out of the box, a custom built kernel can reduce the operating system's memory footprint by compiling in only the drivers and systems needed for the PS3. At the time of this writing, the primary maintainer for the kernel patches for the PS3 is Geoff Levand. His kernel tree is accessible to be downloaded with git:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/geoff/ps3-linux.git ps3-linux
```

Once the kernel sources are downloaded, the kernel can be configured and made:

```
make mrproper ps3_defconfig menuconfig
```

At this point, any desired configuration changes can be made to the kernel. If you are not making changes to the kernel, then it is fairly safe to disable the kernel hacking and debugging options. Also, if you allowed the installer to setup the root drive on a logical volume when installing Linux, then be sure to enable LVM support in the kernel as well. It can be found in Generic Driver Options → Multiple devices driver support → Device mapper support.

After all configuration changes have been made and you are ready to build the kernel, run the following to build and install it:

```
make all modules_install install
```

After running the install, look in the `/boot` directory for the new kernel and initrd file. In order to configure Petitboot to load the new kernel, edit the `/etc/yaboot.conf` file, by copying the default configuration and replacing the kernel and initrd file names with the new ones. After saving `/etc/yaboot.conf`, reboot and select the new kernel from the bootloader menu.

## B.6 Configuring Huge Pages

After installing a Linux kernel capable of supporting huge pages, the next task is to reserve them so they can be used by applications. Once reserved in the kernel, these pages are accessible from user space through a filesystem interface. This can be accomplished with the following commands:

```
mkdir -p /huge
echo 20 > /proc/sys/vm/nr_hugepages
mount -t hugetlbfs nodev /huge
chown root:root /huge
chmod 777 /huge
```

For further information, there is a great article detailing the usage of huge pages as well as a script to allocate them automatically on startup at cellperformance.com.

## B.7 Installing the IBM SDK

The CELL platform SDK for Linux may be downloaded from the IBM website. There are a variety of packages and options, but for use on the PS3 the Fedora packages are most likely what you want. These can be downloaded to a directory on your PS3 along with the installer rpm and installed with the following commands:

```
rpm -ivh ./cell-install-3.1.0-0.0.noarch.rpm
/opt/cell/cellsdk --iso /root/sdk/ install
```

More information regarding installation of the SDK may be found online.

# Appendix C

# The VH Solver

This appendix describes the high level implementation of the Vose-Horton CTIS reconstruction algorithm described in section 2.2. While the actual implementation on the CELL processor is written in C, the solver is presented here in pseudocode to facilitate reading. The C code running on the PPE of the Cell follows this presentation fairly closely, with the acceleration being handled in the vector operations. In the following pseudocode, all operations involving vectors should be interpreted as component wise. The standard arithmetic operations have their usual function. Additional functions are the dot product denoted $(\cdot)$, the Fourier transform $\mathcal{F}()$ the inverse Fourier transform $\mathcal{F}^{-1}()$, and the complex conjugate denoted by an overline.

## C.1 Setup

| Vose Horton Solver Parameters |
|---|
| $x$: output vector of the CTIS process |
| $H$: transfer matrix |
| $n$: number of rows in $H$ |
| $w$: number of partitions of $H$ |
| $\alpha$: number of circulant blocks in each partition of $H$ |
| $a$: number of columns in each circulant block |
| $g$: shift invariance constraint from assumption 2 |
| $\mu$: regularization constant |
| VH_ITERATIONS: iteration bound for VH heuristic |
| VH_EPSILON: error bound for the VH heuristic |
| CG_ITERATIONS: iteration bound for embedded conjugate gradient |
| CG_EPSILON: error bound for the embedded conjugate gradient |

The VH solver is characterized by a number of input parameters, many of which are defined by the CTIS problem. The regularization constant $\mu$ introduced in equation 2.4 was selected to be 0.01 in the original Vose Horton paper. In practice, a larger value such as 0.1 may be desirable when handling large problems as it introduces less rounding error. The parameters VH_ITERATIONS and VH_EPSILON control the stopping conditions of the algorithm at a certain number of iterations or when the residual error is below a certain threshold.

Likewise, the CG_ITERATIONS and CG_EPSILON parameters control the embedded conjugate gradient allowing it to be terminated early.

Multiplication of the $H$ matrix is prominently featured throughout the reconstruction process, but due to the shortcuts taken by the VH algorithm these multiplications may be performed with vector operations in Fourier space. Accordingly, only the Fourier transform of it's special column vectors are needed which is advantageous since $H$ may be both large and sparse. These precomputed vectors are stored in the set $\mathbf{F}$. Another somewhat time consuming operation is the summation and square root operations from equation 2.5. Since this is a frequently used term in the solver, it is also precomputed offline and stored in the vector set $\mathbf{E}$. Finally, equation 2.8 required multiplication by the diagonalized matrices $Z$ and $Z'$, the effect of which expands or contracts the multiplied vector. This is implemented as a lookup table, $\mathbf{Z}$, which is precomputed and used directly to expand and contract vectors as needed. The C implementation uses a bitmask to conserve space.

Preprocessing of $H$

```
// Precompute F ;
for i = 0 to w do
    F_i = F(H_iαa) ;
end
// Precompute E ;
d = ((Σ_{i=0}^{w} F_i conj(F_i)) (n/μ) + n)^{1/2} ;
for i = 0 to w do
    E_i = d * conj(F_i) ;
end
// Create Z ;
i = 0 ;
for j = 0 to α do
    for k = 0 to a do
        Z_i = 1 ;
        i = i + 1 ;
    end
    for k = 0 to g − a do
        Z_i = 0 ;
        i = i + 1 ;
    end
end
for k = 0 to n − (α ∗ g) do
    Z_i = 0 ;
    i = i + 1 ;
end
```

## C.2 Implementation

The main loop of the solver uses the heuristic kernel to compute an estimate of the vector f, calculates the residual from the estimate, and then iterates until either the residual is below a certain threshold or the required number of iterations is reached.

**Algorithm 3**: VHSolver

```
f_estimate = 0 ;
vh_itr = 0 ;
residual = x ;
while vh_eps > VH_EPSILON and vh_itr < VH_ITERATIONS do
    f_estimate = HeuristicKernel(residual, f_estimate) ;
    residual = ComputeResidual(f_estimate) ;
    vh_eps = √(residual · residual) ;
    vh_itr = vh_itr + 1 ;
return f_estimate;
```

A majority of the computations of the VH algorithm uses four global "'scratch"' vectors, $v0$ through $v3$. While more workspace vectors could be used, this was the smallest number we were able to get away with. Reusing this workspace as much as possible keeps down the utilization of valuable RAM when dealing with larger problem sizes. The heuristic kernel implements the solution to equation 2.8.

**Procedure** `HeuristicKernel(`*residual, fEst*`)`

**input**: residual

$v0 = \mathcal{F}$(residual) ;

$v3 = 0$ ;

// Begin Extended Precision Region ;

// $\mathcal{H}^T x$ ;

**for** $i = 0$ **to** $w$ **do**

  $v2 = \overline{\mathbf{F}_i} * v0$ ;

  $v1_i = \overline{\mathbf{E}_i} * v2$ ;

  $v3 = v3 + v1_i$ ;

  $v1_i = \frac{1}{\mu} * v2$ ;

// $PP^T$ ;

**for** $i = 0$ **to** $w$ **do**

  $v2 = \mathbf{E}_i * v3$ ;

  $v2 = v1 - v2$ ;

  // End Extended Precision Region ;

  $v0_i = \mathcal{F}^{-1}(v2)$ ;

$v1 = 0$ ;

**for** $i = 0$ **to** $w$ **do**

  $v2 =$ `Zmask(`$v0_i$, $\mathbf{Z}$`)` ;

  $v2 = \mathcal{F}(v2)$ ;

  $v2 = \overline{\mathbf{E}_i} * v2$ ;

  $v1 = v1 + v2$ ;

$v1 = \mathcal{F}^{-1}(v1)$ ;

$v2 =$ `ConjugateGradient(`$v1$`)` ;

$v2 = \mathcal{F}(v2)$ ;

**for** $i = 0$ **to** $w$ **do**

  $v3 = \mathbf{E}_i$ * $v2$;

  $v3 = \mathcal{F}^{-1}(v3)$ ;

  $v1_i = v3 + v0$;

$v2 = 0$;

**for** $i = 0$ **to** $w$ **do**

  $v3 =$ `Zmask(`$v1$, $\mathbf{Z}$`)` ;

  $v3 = \mathcal{F}(v3)$ ;

  $v3 = \overline{\mathbf{E}_i} * v3$ ;

  $v2 = v2 + v3$ ;

**for** $i = 0$ **to** $w$ **do**

  $v3 = \mathbf{E}_i * v2$ ;

  $v3 = \mathcal{F}^{-1}(v3)$ ;

  $v0_i = v0_i + \mu * v3$ ;

f_estimate = `ZReduce(`$v2$, $v0$, $\mathbf{Z}$`)` ;

**return** f_estimate;

**Procedure** `Zmask(`*vIn, mask*`)`

```
i = 0 ;
for j = 0 to w do
    for k = 0 to n do
        if mask_j is 1 then
            vOut_{k+(j*n)} = vIn_i ;
            i = i + 1 ;
        else
            vOut_{k+(j*n)} = 0
return vOut;
```

**Procedure** `ZReduce`

```
i = 0 ;
for j = 0 to w do
    for k = 0 to n do
        if Z_j is 1 then
            f_estimate_i = f_estimate_i + v0_{(j*n)+k} ;
            v2_{(j*n)+k} = f_estimate_i ;
            i = i + 1 ;
        else
            v2_{(j*n)+k} = 0 ;
```

**Procedure** `ConjugateGradient`

**input**: $v1$
cg_x $= 0$ ;
cg_r $= v1$;
cg_p $= v1$;
cg_eps $= (\text{cg\_r} \cdot \text{cg\_r})$ ;

**while** cg_eps $>$ CG_EPSILON **and** cg_itr $<$ CG_ITERATIONS **do**
    last_eps $=$ cg_eps;
    cg_v $= 0$ ;
    $v2 = \mathcal{F}(\text{cg\_p})$ ;
    **for** $i = 0$ **to** $w$ **do**
        $v3 = \mathbf{E}_i * v2$ ;
        $v3 = \mathcal{F}^{-1}(v3)$ ;
        $v3 = \texttt{Zmask}(v3,\ \mathbf{Z})$ ;
        $v3 = \mathcal{F}(v3)$ ;
        $v3 = \overline{\mathbf{E}_i} * v3$ ;
        cg_v $=$ cg_v $+ v3$ ;
    cg_v $= \mathcal{F}^{-1}(\text{cg\_v})$ ;
    cg_v $= \frac{1}{\mu} *$ cg_p $-$ cg_v ;
    ca $= \frac{\text{cg\_eps}}{(\text{cg\_p} \cdot \text{cg\_v})}$ ;
    cg_r $= -$ca $*$ cg_v $+$ cg_r ;
    dotR $= (\text{cg\_r} \cdot \text{cg\_r})$ ;
    `// Stop at the first local minimum of error` ;
    **if** last_eps $<$ dotR **then**
        **break**;
    cg_x $=$ cg_x $+$ ca $*$ cg_p ;
    ca $=$ dotR;
    cg_p $=$ cg_r $+ \frac{\text{ca}}{\text{cg\_eps}} *$ cg_p ;
    cg_eps $=$ ca;
    cg_itr $=$ cg_itr $+ 1$ ;

**return** cg_x

**Procedure** `ComputeResidual`

$v1 = 0$ ;
**for** $i = 0$ **to** $w$ **do**
    $v3 = \mathcal{F}(v2_i)$ ;
    $v3 = \mathbf{F}_i * v3$ ;
    $v3 = \mathcal{F}^{-1}(v3)$ ;
    $v1 = v1 + v3$;
$\mathsf{ls} = \frac{(v1 \cdot x)}{(v1 \cdot v1)}$ ;
i $= 0$ ;
**for** $j = 0$ **to** $w$ **do**
    **for** $k = 0$ **to** $n$ **do**
        **if** $\mathbf{Z}_j$ *is 1* **then**
            $v1_{j*n+k} = \mathsf{f\_estimate}_i * \mathsf{ls}$ ;
            $i = i + 1$ ;
        **else**
            $v1_{(j*n)+k} = 0$ ;
$v0 = 0$ ;
**for** $i = 0$ **to** $w$ **do**
    $v3 = \mathcal{F}(v1_i)$ ;
    $v3 = \mathbf{F}_i * v3$ ;
    $v3 = \mathcal{F}^{-1}(v3)$ ;
    $v0 = v0 + v3$;
$v0 = x - v0$ ;
**return** $v0$;

# Appendix D

# Building the CTIS Solver

The CELL implementation of the Vose-Horton CTIS solver may be obtained from
http://www.cs.utk.edu/~tthompso.

The library depends on the CELL SDK version 3.1 and the FFTW library. See section B.7 for instructions on installing the SDK. Additionally, there are a few prerequisite packages which can be installed with yum:

```
yum -y install glibc-devel.ppc64 wget.ppc make.ppc
```

After untarring the CTIS archive, the first item of business is to start the build of the FFTW library. The full CTIS solution requires the long-double, double, and single precision libraries to be built. The `setup_fftw.sh` script in the package will automatically download and configure the library. The blasx_benchmarks directory contains driver and test code to benchmark the accelerated operations in the BLASX library. They can be run as follows:

```
cd blasx\_benchmarks
make
./blasx\_bench -i 3 all 2097152
```

This command will run 3 iterations of all functions with an input length of 2097152 elements. The CTIS solver can also be run:

```
cd ctis
make
./run_full_test.sh data/conf_test.txt
```

The `run_full_test.sh` script executes a full test sequence of the CTIS solver:

1. `gen_test_data` is run to create a test vector f and matrix H.
2. `pre_matrix` pre-processes the matrix to create the eta and Fck vectors.
3. `multHF` multiplies the test matrix by the test vector using the pre-processed vectors.
4. `ctis_double` runs the double precision solver and outputs f_est.
5. `ctis_mixed` runs the mixed precision solver and outputs f_est.

The `conf_test.txt` file is the test configuration used for the CTIS solver benchmark presented in the paper. Note however that the paper used a different H matrix and input image.

# Appendix E

# SPU Task Optimization

While writing performance sensitive code to run on the Cell processors SPEs can be challenging, a nice feature of the platform is that instruction timing is deterministic. Since the SPU has no cache and a very simple pipeline, instruction timing can be analyzed statically with a fairly high degree of accuracy. The Cell Software Development Kit includes the `spu_timing` utility to help with this task. As of release 3.1 of the SDK, the timing utility is not installed by default, but is available as an RPM in the `CellSDK-Extras-Fedora_3.1.0.0.0.iso` file. After installing the tool, it can be found at `/opt/cell/sdk/usr/bin/spu_timing`. The `spu_timing` tool analyzes text assembly files and creates timing diagrams for them which can read to understand how an SPU will pipeline and execute a block of code. When using the SDK's build environment, the timing tool can be invoked automatically by putting the command `SPU_TIMING=1` in the makefile. Alternately, a timing profile can quickly be generated by specifying the tool in the environment with your make command, such as this:

```
SPU_TIMING=1 make foo.s
```

Where *foo* is the name of a code module (i.e. *foo.c*) in the application. The SPU static timing tool instruments an SPU assembly file with scheduling, timing, and instruction issue estimates assuming a linear execution of the program. The output of this tool, not including the optional running cycle count, is as follows:

**Column 1** — The first column indicates the pipeline, either 0 or 1, corresponding to even or odd pipeline, respectively, in which the instruction is issued.

**Column 2** — The second column indicates dual-issue status. A "D" in this column signifies successful dual-issue of the pair of instructions. A "d" in this column signifies dual-issue is possible, but would not occur due to operand dependencies (for example, operands being in flight). No text in this column indicates that dual-issue rules are not satisfied.

**Column 3** — always blank.

**Column 4 through 53** — The next 50 columns indicate clock cycles, "0123456789" repeated 5 times. A digit is displayed for every clock cycle the instruction executes. Therefore, a n-cycle instruction will display n digits. Operand dependency stalls are flagged by a dash ("-") for every cycle the instruction is expected to stall.

**Column 54 and beyond** — The original assembly input code.

As an example for analysis, consider the function *VectorSubtract*, which, as it's name implies, subtracts two vectors. This first code snippet is from a naive implementation in C:

```
void VectorSubtract(vector float *x, vector float *y, vector float *z, int N)
{
    int i;
    for(i=0; i < N; i++)
      z[i] = x[i] - y[i];
}
```

Running the SPU timing tool gives the following output:

```
VectorSubtract:
000093 0d                                          34          cgti    $2,$6,0
000098 1d 01                                       -----89     shlqbyi $8,$3,0
000099 0D                                                9     nop     $127
000099 1D 012                                            9     biz     $2,$lr
000100 0D 01                                                   il      $7,0
000100 1D 012345678901234                                      hbra    .L17,.L14
                                                              .L14:
000101 0d  12                                                  ai      $6,$6,-1
000102 1d  -234567                                             lqx     $2,$7,$4
000103 1     345678                                            lqx     $3,$7,$5
000104 0      4                                                nop     $127
000105 0       5                                               nop     $127
000109 0        ---901234                                      fs      $2,$2,$3
000115 1            -----567890                                stqx    $2,$7,$8
000116 0                  67                                   ai      $7,$7,16
                                                              .L17:
000117 1                   7890                                brnz    $6,.L14
000118 1                    8901                               bi      $lr
```

Inspecting the body of the loop (between .L14 and .L17) reveals a rather inefficient process. No instructions can be dual issued and there are long stalls on lines 109 and 115. Because this tight loop forms the core of our computation, these stalls greatly impede overall CPU throughput.

Next, let us examine the same function after vectorization and unrolling the loop:

```
void VectorSubtract(vector float *x, vector float *y, vector float *z, int N)
{
    int i;
    register vector float x0,x1,x2,x3,x4,x5,x6,x7;
    register vector float y0,y1,y2,y3,y4,y5,y6,y7;

    for(i=0; i < N; i += 8){
        x0 = x[i+0];
        x1 = x[i+1];
        x2 = x[i+2];
        x3 = x[i+3];
        x4 = x[i+4];
        x5 = x[i+5];
        x6 = x[i+6];
        x7 = x[i+7];
```

```
        y0 = y[i+0];
        y1 = y[i+1];
        y2 = y[i+2];
        y3 = y[i+3];
        y4 = y[i+4];
        y5 = y[i+5];
        y6 = y[i+6];
        y7 = y[i+7];

        z[i+0] = spu_sub(x0,y0);
        z[i+1] = spu_sub(x1,y1);
        z[i+2] = spu_sub(x2,y2);
        z[i+3] = spu_sub(x3,y3);
        z[i+4] = spu_sub(x4,y4);
        z[i+5] = spu_sub(x5,y5);
        z[i+6] = spu_sub(x6,y6);
        z[i+7] = spu_sub(x7,y7);
    }
}
```

The SPU timing output is now:

```
VectorSubtract:
000093 0D                                  34          cgti    $2,$6,0
000093 1D                                  3456        shlqbyi $24,$6,0
000098 0D                                   ----89     ori     $23,$3,0
000098 1D 01                                    89     shlqbyi $22,$4,0
000099 0D 0                                      9     ori     $21,$5,0
000099 1D 012                                    9     biz     $2,$lr
000100 0D 01                                           il      $20,0
000100 1D 012345678901234                              hbra    .L17,.L14
000101 0D  12                                          il      $19,0
000101 1D  1                                           lnop
                                                      .L14:
000103 0D   -34                                        a       $2,$22,$19
000103 1D    345678                                    lqx     $8,$19,$21
000104 0D     45                                       a       $3,$21,$19
000104 1D     456789                                   lqx     $10,$19,$22
000105 0D      56                                      ai      $20,$20,8
000105 1D      567890                                  lqd     $5,112($2)
000106 0D       67                                     a       $4,$19,$23
000106 1D       678901                                 lqd     $13,112($3)
000107 0D        78                                    cgt     $18,$24,$20
000107 1D        789012                                lqd     $6,16($2)
000108 1          890123                               lqd     $11,32($2)
000109 1           901234                              lqd     $12,48($2)
000110 0D           012345                             fs      $10,$10,$8
000110 1D           012345                             lqd     $7,64($2)
000111 1             123456                           lqd     $8,80($2)
000112 1              234567                          lqd     $9,96($2)
000113 0D              345678                         fs      $5,$5,$13
```

```
000113 1D              3                                    lnop
000114 1                456789                              lqd     $2,16($3)
000115 1                 567890                             lqd     $13,32($3)
000116 1                  678901                            lqd     $14,48($3)
000117 1                   789012                           lqd     $15,64($3)
000118 1                    890123                          lqd     $16,80($3)
000119 1                     901234                         lqd     $17,96($3)
000120 0D                    012345                         fs      $6,$6,$2
000120 1D                    012345                         stqx    $10,$19,$23
000121 0D                     123456                        fs      $11,$11,$13
000121 1D                     123456                        stqd    $5,112($4)
000122 0                       234567                       fs      $12,$12,$14
000123 0                        345678                      fs      $7,$7,$15
000124 0                         456789                     fs      $8,$8,$16
000125 0                          567890                    fs      $9,$9,$17
000126 0D                          67                       ai      $19,$19,128
000126 1D                          678901                   stqd    $6,16($4)
000127 1                            789012                  stqd    $11,32($4)
000128 1                             890123                 stqd    $12,48($4)
000129 1                              901234                stqd    $7,64($4)
000130 1                               012345               stqd    $8,80($4)
000131 1                                123456              stqd    $9,96($4)
                                                           .L17:
000132 1                                 2345               brnz    $18,.L14
000133 1                                  3456              bi      $lr
```

This version has a noticeably longer loop section even though it performs the same amount of work. However, with this version there is only one stall cycle at the top of the loop and many of the instructions are dual issued. When timed with the spu_decrementer the original implementation executes in 8280 clock cycles while this optimized implementation runs in slightly over 2000 cycles.

# Vita

Thad Thompson grew up in San Angelo Texas. He went to Angelo Christian School and graduated in 1997. Subsequently, he attended Angelo State University and received his BS in Computer Science in 2001. After moving to Tennessee he worked as an IT and software engineer while attending the University of Tennessee part time, receiving an MS in Computer Science in 2009. He currently lives in Knoxville with his wife Robyn, and can be contacted at thad.thompson@gmail.com.